Towards Verifying Nonlinear Integer Arithmetic

Paul Beame∗
Computer Science and Engineering
University of Washington
beame@cs.washington.edu

Vincent Liew∗
Computer Science and Engineering
University of Washington
vliew@cs.washington.edu

May 12, 2017

Abstract

We eliminate a key roadblock to efficient verification of nonlinear integer arithmetic using CDCL SAT solvers, by showing how to construct short resolution proofs for many properties of the most widely used multiplier circuits. Such short proofs were conjectured not to exist. More precisely, we give \( n^{O(1)} \) size regular resolution proofs for arbitrary degree 2 identities on array, diagonal, and Booth multipliers and \( n^{O(\log n)} \) size proofs for these identities on Wallace tree multipliers.

1 Introduction

Recent decades have seen remarkable advances in our ability to verify hardware. Methods for hardware verification based on Ordered Binary Decision Diagrams (OBDDs) developed in the 1980s for hardware equivalence testing [15] were extended in the 1990s to produce general methods for symbolic model checking [17] to verify complex correctness properties of designs. More recently, several orders of magnitude of improvements in the efficiency of SAT solvers have brought new vistas of verification of hardware and software within reach.

Nonetheless, there is an important area of formal verification where roadblocks that were identified in the 1980s still remain: verification of data paths within designs for Arithmetic Logic Units (ALUs), or indeed any verification problem in hardware or software that involves the detailed properties of nonlinear arithmetic. Natural examples of such verification problems in software include computations involving hashing or cryptographic constructions. At the highest level of abstraction, nonlinear arithmetic over the integers is undecidable, but the focus of these verification problems is on the decidable case of integers of bounded size, which is naturally described in the language of bit-vector arithmetic (see, e.g. [31, 29]).

In particular, a notorious open problem is that of verifying properties of integer multipliers in a way that both is general purpose and avoids exponential scaling in the bit-width. Bryant [10] showed that this is impossible using OBDDs since they require exponential size in the bit-width just to represent the middle bit of the output of a multiplier. This lower bound has been improved [10] and extended to include very tight exponential lower bounds for much more general diagrams than OBDDs, and FBDDs [35, 9] and general bounded-length branching programs [37]. With the

∗Research supported in part by NSF grant CCF-1524246.
flexibility of CNF formulas, efficient representation of multipliers is no longer a problem but, even with the advent of greatly improved SAT solvers, there has been no advance in verifying multipliers beyond exponential scaling.

One important technique for verifying software and hardware that includes multiplication has been to use methods of uninterpreted functions to handle multipliers (see [13, 31]) – essentially converting them to black boxes and hoping that there is no need to look inside to check the details. Another important technique has been to observe that it is often the case that one input to a multiplier is a known constant and hence the resulting computation involves linear, rather than nonlinear arithmetic. These approaches have been combined with theories of arithmetic (e.g. [11, 34, 41, 12]), including preprocessors that do some form of rewriting to eliminate nonlinear arithmetic, but these methods are not able, for example, to check the details of a multiplier implementation or handle nonlinearity.

Though the above approaches work in some contexts, they are very limited. The approach of verifying code with multiplication using uninterpreted functions is particularly problematic for hashing and cryptographic applications. For example, using uninterpreted functions in the actual hash function computation inherently can never consider the case that there is a hash collision, since it only can infer equality between terms with identical arguments. Concern about such applications is real: longstanding errors in multiplication in OpenSSL have recently come to light [33].

Recent presentations at verification conferences and workshops have highlighted the problem of verifying nonlinear arithmetic, and multipliers in particular, as one of the key gaps in our current verification methods [5, 6, 28, 8].

Since bit-vector arithmetic is not itself a representation in Boolean variables, in order to apply SAT solvers to verify the designs, one must convert implementations and specifications to CNF formulas based on specified bit-widths. The process by which one does this is called flattening [31], or more commonly bit-blasting. The resulting CNF formulas are then sent to the SAT solvers. While the resulting bit-blasted CNF formulas for a multiplier may grow quadratically with the bit-width, this growth is not a significant problem. On the other hand, a major stumbling block for handling even modest bit-widths is the fact that existing SAT solvers run on these formulas experience exponential blow-up as the bit-width increases. This is true even for the best of recent methods, e.g., Boolector [12], MathSAT [13], STP [26], Z3 [24], and Yices [23].

In verifying a multiplier circuit one could try to compare it to a reference circuit that is known to be correct. This introduces a chicken-and-egg problem: how do we know that the reference circuit is correct? Another approach to verifying a multiplier circuit is to check that it satisfies the right properties. A correct multiplier circuit must obey the multiplication identities for a commutative ring. If we check that each of these ring identities holds then the multiplier cannot have an error. This approach has the advantage that the specification of a multiplier circuit can be written a priori in terms of its natural properties, rather than in terms of an external reference circuit.

Empirically, however, modern SAT-solvers perform badly using either approach to problems of multiplier verification. Biere, in the text accompanying benchmarks on the ring identities submitted to the 2016 SAT Competition [7] writes that when given as CNF formulas, no known technique is capable of handling bit-width larger than 16 for commutativity or associativity of multiplication or bit-width 12 for distributivity of multiplication over addition. These observations lead to the question: is the difficulty inherent in these verification problems, or are modern SAT-solvers just using the wrong tools for the job?

Modern SAT-solvers are based on a paradigm called conflict-directed clause-learning
(CDCL) which can be seen as a way of breaking out of the backtracking search of traditional DPLL solvers. When these solvers confirm the validity of an identity (by not finding a counterexample), their traces yield resolution proofs of that identity. The size of such a proof is comparable to the running time of the solver; hence finding short resolution proofs of these identities is a necessary prerequisite for efficient verification via CDCL solvers. Although it is not known whether CDCL solvers are capable of efficiently simulating every resolution proof, all cases where short resolution proofs are known have also been shown to have short CDCL-style traces (e.g., [19, 18, 20]).

The extreme lack of success of general purpose solvers (in particular CDCL solvers) for verifying any non-trivial properties of bit-vector multiplication, recently led Biere to conjecture that there is a fundamental proof-theoretic obstacle to succeeding on such problems; namely, verifying ring identities for multiplication circuits, such as commutativity, requires resolution proofs that are exponential in the bit-width $n$.

We show that such a roadblock to efficient verification of nonlinear arithmetic does not exist by giving a general method for finding short resolution proofs for verifying any degree 2 identity for Boolean circuits consisting of bit-vector adders and multipliers. This method is based on reducing the multiplier verification to finding a resolution refutation of one of a number of narrow critical strips. We apply this method to a number of the most widely used multiplier circuits, yielding $n^{O(1)}$ size proofs for array, diagonal, and Booth multipliers, and $n^{O(\log n)}$ size proofs for Wallace tree multipliers.

These resolution proofs are of a special simple form: they are regular resolution proofs. Regular resolution proofs have been identified in theoretical models of CDCL solvers as one of the simplest kinds of proof that CDCL solvers naturally express. Indeed, experience to date has been that the addition of some heuristics to CDCL suffices to find short regular resolution proofs that we know exist. The new regular resolution proofs that we produce are a key step towards developing such heuristics for verifying general nonlinear arithmetic.

Related work SAT solver-based techniques used in conjunction with case splitting previously were shown to achieve some success for multiplier verification in the work of Andrade et al. improving on earlier work which combined SAT solver and OBDD-based ideas for multiplier verification among other applications; however, there was no general understanding of when such methods will succeed.

Recently, two alternative approaches to multiplier verification have been considered: Kojevnikov designed a mixed Boolean-algebraic solver, BASolver, that takes input CNF formulas in standard format. It uses algebraic rules on top of a DPLL solver. Though it can verify the equivalence of multipliers up to 32 bits in a reasonable time, in each instance it requires human input in order to find a suitable set of algebraic rules to help the solver. An alternative approach using Groebner basis algorithms has been considered. This is a purely algebraic approach based on polynomials. Since the language of polynomials allows one to explicitly write down the algebraic specification for an $n$-bit multiplier, the verification problem is conveniently that of checking that the multiplier circuit computes a polynomial equivalent to the multiplier specification.

---

1Some of these proofs are even more restricted ordered resolution proofs, also known as DP proofs, which are associated with the original Davis-Putnam procedure. In contrast to the Davis-Putnam procedure, which eliminates variables one-by-one keeping all possible resolvents, ordered resolution (or DP) proofs only keep some minimal subset of these resolvents needed to derive a contradiction.
that Groebner basis algorithms can be used to verify 64-bit multipliers in less than ten minutes and 128-bit multipliers in less than two hours. However, this requires that the multipliers be identified and treated entirely separately from the rest of the circuit or software. Unfortunately, for the non-algebraic parts of circuits, Groebner basis methods can only handle problems several orders of magnitude smaller than can be handled by CDCL SAT-solvers and it remains to be seen whether it is possible to combine these to obtain effective verification for a general purpose software with nonlinear arithmetic or circuits that contain a multiplier as just one component of their design. In contrast, CDCL SAT solvers are already very effective for the non-algebraic aspects of circuits and are well-suited to handling the combination of different components; our work shows that there is no inherent limitation preventing them from being effective for verification of general purpose nonlinear arithmetic.

Roadmap: In Section 3 we introduce and give constructions of some standard multipliers, in particular the array multiplier and the Wallace tree multiplier. We give polynomial size regular resolution proofs for degree 2 identities for circuits constructed using array, diagonal, and Booth multipliers in Section 4 and give quasipolynomial size regular resolution proofs for circuits constructed using Wallace tree multipliers in Section 5.

2 Notation and Preliminaries

We represent Boolean variables in lowercase and denote clauses by uppercase letters and think of them as sets of literals, for example $C = \{x, \bar{y}, z\}$. We will work with length $n$ bit-vectors of variables, denoted by $z = z_{n-1} \ldots z_1 z_0$.

We consider identities from the commutative ring of integers $\mathbb{Z}$. We use a set $\sigma = \sigma(x_0, x_1 \ldots x_n) = \{x_0 = b_0, x_1 = b_1 \ldots x_n = b_n\}$, where each $b_i \in \{0, 1\}$, to denote an assignment to the variables $x_0, x_1, \ldots x_n$.

Definition A commutative ring $(\mathcal{R}, \oplus, \otimes, 0, 1)$ consists of a nonempty set $\mathcal{R}$ with addition ($\oplus$) and multiplication ($\otimes$) operators that satisfy the following properties:
1. $(\mathcal{R}, \oplus)$ is associative and commutative and its identity element is 0.
2. For each $x \in \mathcal{R}$ there exists an additive inverse.
3. $(\mathcal{R}, \otimes)$ is associative and commutative and its identity element is $1 \neq 0$.
4. (distributivity) For all $x, y, z \in \mathcal{R}$, $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$.

A ring identity $L = R$ denotes a pair of expressions $L, R$ that can be transformed into each other using commutativity, distributivity and associativity.

Note that both verifying integer $\oplus$ circuits and verifying that $x \otimes 1 = x$ are easy in practice, so verifying an integer multiplier circuit $\otimes$ can be easily reduced to verifying its distributivity.

Definition A resolution proof consists of a sequence of clauses, each of which is either a clause of the input formula $\phi$, or follows from two prior clauses via the resolution rule which produces clause $C \lor D$ from clauses $C \land x$ and $D \lor \bar{x}$. We say that this inference resolves the clauses on $x$. The proof is a refutation of $\phi$ if it ends with the empty clause $\bot$. (With resolution we will use the terms “proof” and “refutation” interchangeably, since resolution provides proofs of unsatisfiability.)
We can naturally represent a resolution proof \( P \) as a directed acyclic graph (DAG) of fan-in 2, with \( \perp \) labelling the lone sink node. Tree resolution is the special subclass of resolution proofs where the DAG is a directed tree. Another restricted form of resolution is regular resolution: A resolution refutation is regular iff on any path in its DAG the inferences resolve on each variable at most once. The shortest tree resolution proofs are always regular. An ordered resolution refutation is a regular resolution refutation that has the further property that the order in which variables are resolved on along each path is consistent with a single total order of all variables. This is a very significant restriction and indeed the shortest tree resolution proofs do not necessarily have this property.

We will find it convenient to express our regular resolution proofs in the form of a branching program that solves the conflict clause search problem.

**Definition** Suppose that \( \phi \) is an unsatisfiable formula. Then every assignment \( \sigma \) to its variables conflicts with some clause in \( \phi \). The conflict clause search problem is to map any assignment to some corresponding conflicting clause.

**Definition** A branching program \( B \) on the Boolean variables \( X = \{ x_0, x_1, \ldots \} \) and output set \( \phi \) (typically a set of clauses in this paper) is a finite directed acyclic graph with a unique source node and sink nodes at its leaves, each leaf labeled by an element from \( \phi \). Each non-sink node is labeled by a variable from \( X \) and has two outgoing edges, one labeled 0 and the other labeled 1. An assignment \( \sigma \) activates an edge labeled \( b \in \{0, 1\} \) outgoing from a node labeled by the variable \( x_i \) if \( \sigma \) contains the assignment \( x_i = b \). If \( \sigma \) activates a path from the source to a sink labeled \( C \in \phi \), we say that the branching program \( B \) outputs \( C \).

A read-once branching program (also known as a Free Binary Decision Diagram, or FBDD) is a branching program where each variable is read at most once on any path from source to leaf. An Ordered Binary Decision Diagram (OBDD) is a special case of an FBDD in which the variables read along any path are consistent with a single total order.

The general case of the following theorem connecting regular resolution proofs and conflict clause search is due to Krajicek [30]; the special case connecting ordered resolution and OBDDs for the conflict clause search problem, which is a simple observation extending the original proof, does not seem to have been explicitly noted previously.

**Theorem 2.1.** Let \( \phi \) be an unsatisfiable formula. A regular resolution refutation \( R \) for \( \phi \) of size \( s \) corresponds to a size \( s \) read-once branching program that solves the conflict clause search problem for \( \phi \).

Suppose that \( B \) is a read-once branching program of size \( s \) solving the conflict clause search problem for \( \phi \). Then there is a regular resolution refutation for \( \phi \) of size \( s \).

Furthermore, if \( R \) is an ordered resolution refutation then the resulting branching program is an OBDD and if \( B \) is an OBDD then the resulting resolution refutation is an ordered resolution refutation.

**Proof.** Suppose that \( R \) is a regular resolution refutation of size \( s \) for \( \phi \). Each clause \( C \) appearing in \( R \) is a node of \( B \). If two clauses \( C_0 \lor x, C_1 \lor \bar{x} \) in \( R \) resolve on a variable \( x \) to produce the clause \( C \), then in the branching program \( B \) we branch from the node \( C \) on the variable \( x \) to reach \( C_0 \lor x \) on the \( x = 0 \) branch, and \( C_1 \lor \bar{x} \) on the \( x = 1 \) branch. The resulting branching program \( B \) solves
the conflict clause search problem for $\phi$ and has the same size as the refutation $R$. The fact that no variable is branched on more than once on any path is immediate from the definition; the fact that this results in an OBDD in the case of ordered resolution is also immediate.

In the other direction, we obtain a regular refutation $R$ from the specified read-once branching program $B$. We will label each node $v$ with the maximal clause $C_v$ that is falsified by every assignment reaching $v$. These clauses form the regular resolution refutation. If $v$ is a leaf then $C_v$ is the conflicting clause from $\phi$ found by $B$. If $B$ branches from node $v$ on a variable $x$ to nodes $v_0, v_1$, then in $R$ we resolve the clauses $C_{v_0}, C_{v_1}$ on $x$ to obtain $C_v$. Again, the number of clauses in the refutation $R$ is the same as the number of nodes in the branching program $B$. The fact that the resolution is regular follows immediately from the fact that the branching program is read-once; if the branching program is an OBDD then it is immediate that the resolution refutation is ordered.

In our proofs we write each clause as the assignment it forbids. For example we write the clause $\{x, \bar{y}\}$ as the assignment $\{x = 0, y = 1\}$. We will build up branching programs for conflict clause search in $\phi$ in terms of three types of action, shown in Figures 2, 3, 4. At a node labeled by an assignment $\sigma \not\ni z$, we branch on the variable $z$ by adding a child node with assignment $\sigma \cup \{z = 0\}$, connected by a 0-labeled edge, and another child node $\sigma \cup \{z = 1\}$, connected by a 1-labeled edge. In the case that one of these children has an assignment conflicting with a clause $C \in \phi$, we say that we propagated the assignment $\sigma$ to the other child’s assignment. Lastly, for a set of leaf nodes with assignments $\sigma_0, \sigma_1, \ldots$ we can merge their branches based on a common assignment $\sigma = \cap_i \sigma_i$ by replacing these nodes with a single node labeled by $\sigma$.

3 Multiplier Constructions

We describe our SAT instances as a set of constraints, where each constraint is a set of clauses. We build circuits out of adders that output, in binary, the sum of three input bits. An adder is encoded as follows:
Figure 4: Merging on the common assignment \( \{ b = 0 \} \).

**Figure 5**: 4-bit ripple-carry adder adding \( x, y \). Each box represents a full adder with incoming arrows and outgoing arrows representing inputs and outputs.

**Definition** Suppose \( a_0, a_1, a_2 \) are inputs to an adder \( A \). The outputs \( c, d \) of the adder \( A \) are encoded by the constraints:

\[
\begin{align*}
    d &= a_0 \oplus a_1 \oplus a_2 \\
    c &= \text{MAJ}(a_0, a_1, a_2)
\end{align*}
\]

We call \( c \) and \( d \) the *sum-bit* and *carry-bit* respectively. If an adder has two constant inputs 0 it acts as a *wire*. If it has precisely one constant input 0, we call it a *half adder*. If no inputs are constant, we call it a *full adder*.

Each circuit variable in our constructions has a *weight* of the form \( 2^i \). Each adder will take in three bits of the same weight \( 2^i \) and output a *sum-bit* of weight \( 2^i \) and a *carry-bit* of weight \( 2^{i+1} \). The adder’s definition ensures that the weighted sum of its input bits is the same as the weighted sum of its output bits. In the constructions that follow, we divide the adders up into columns so that the \( i \)-th column contains all the adders with inputs of weight \( 2^i \).

**Ripple-Carry Adder:**

A ripple-carry adder, shown in Figure 5, takes in two bitvectors \( x, y \) and outputs their sum in binary. In the \( i \)-th column, for \( i \leq n \), we place an adder \( A_i \) that takes the three variables \( c_{i-1}, x_i, y_i \) and outputs the adder’s carry variable and sum variable to \( c_i \) and \( o_i \) respectively. In the \( n+1 \)-st column we place a wire \( A_{n+1} \) taking \( c_n \) as input and outputting to \( o_{n+1} \). While the implementation is simple, it has depth \( n \).

**Carry-Lookahead Adder:**

A carry-lookahead adder (CLA) uses a tree structure to add two bitvectors \( x, y \) with only logarithmic depth. The 4-bit CLA computes, for each pair \( x_i, y_i \), the values

\[
\begin{align*}
    g_i^0 &= x_i y_i \\
    p_i^0 &= x_i \oplus y_i
\end{align*}
\]
where the superscript indicates the layer. Then, writing $c_i$ for the carry bit in the $i$-th digit, we have

$$c_{i+1} = g_i^0 + (p_i^0 c_i).$$

We can use this to derive the following equations, which we can use to compute each carry digit in parallel from the values $g_i, p_i$ and $c_0$:

$$

c_1 = g_0^0 + p_0^0 c_0
\quad c_2 = g_1^0 + g_0^0 p_1^0 + c_0 p_0^0 p_1^0
\quad c_3 = g_2^0 + g_1^0 p_2^0 + g_0^0 p_1^0 p_2^0 + c_0 p_0^0 p_1^0 p_2^0
\quad c_4 = g_3^0 + g_2^0 p_3^0 + g_1^0 p_2^0 p_3^0 + g_0^0 p_1^0 p_2^0 p_3^0 + c_0 p_0^0 p_1^0 p_2^0 p_3^0.
$$

These values are used to compute the outputs: $o_i = c_i \oplus x_i \oplus y_i$. It additionally computes the group propagate and group generate:

$$
p_1^1 = p_3^0 p_2^0 p_1^0
\quad g_1^1 = g_3^0 + g_2^0 p_3^0 + g_1^0 p_2^0 p_3^0 + g_0^0 p_1^0 p_2^0 p_3^0.
$$

We construct a 16-bit CLA with 2 layers, whose first half of is shown in Figure 6. At the zero-th layer we arrange four 4-bit CLAs, the $k$-th CLA taking inputs $x_i, y_i, i \in [4k, 4k + 3]$ and outputting to $p_i^0, g_i^0, i \in [4k, 4k + 3]$, where the superscript indicates the layer. We denote the $k$-th CLA group propagate and generate by $p_{4k}^1 g_{4k}^1$. Then the carries $c_4, c_8, c_{12}, \ldots$ can be computed by the equations

$$

c_4 = g_4^1 + p_4^1 c_0
\quad c_8 = g_4^1 + g_4^1 p_8^1 + c_0 p_4^1 p_8^1
\quad c_{12} = g_8^1 + g_4^1 p_8^1 + g_0^1 p_4^1 p_8^1 + c_0 p_0^1 p_4^1 p_8^1
\quad c_{16} = g_{12}^1 + g_8^1 p_{12}^1 + g_4^1 p_8^1 p_{12}^1 + g_0^1 p_4^1 p_8^1 p_{12}^1 + c_0 p_0^1 p_4^1 p_8^1 p_{12}^1.
$$

Notice that these equations are isomorphic to the previous equations for computing carries within each 4-bit CLA. We can reuse the same circuitry from the 4-bit CLA to compute these carries, as well as $p^2, g^2$, the group propagate and generate for the next layer.
We can repeat this process to construct larger CLAs, with each iteration able to handle four times the bitwidth.

All the multipliers we describe perform two phases of computation to compute $xy$ for length $n$ bitvectors $x, y$. The first phase is the same in each multiplier: the circuit computes a tableau of values $x_i \land y_j$ for each pair of input bits $x_i$ and $y_j$. In the second phase where the constructions differ, the circuit computes the weighted sum of the bits in the tableau.

**Array Multiplier:**

An $n$-bit array multiplier works by arranging $n$ ripple-carry adders in sequence in order to sum the $n$ rows of the tableau. This multiplier has a simple gridlike architecture that is compact and easy to lay out physically. It has depth linear in its bitwidth. In the first phase, an array multiplier computes each tableau variable $t_{ij}^C = x_i \land y_j$, with associated weight $2^{i+j}$.

Arrange a grid of full adders $A_{i,j}$, where $i, j \in [0, n]$, as shown in Figure 7. Adder $A_{i,j}$ occupies the $j$-th row and the $i+j$-th column, takes inputs $t_{i,j}, d_{i+i,j-1}, c_{i-1,j}$ (replacing nonexisting variables with the constant 0), and outputs the carry and sum bits $c_{i,j}$ and $d_{i,j}$. Finally, we add constraints equating the sum-bits $d_{0,0}, d_{0,1}, \ldots, d_{0,n-1}, d_{1,n-1}, \ldots, d_{n-1,n-1}$ with the corresponding output bits $o_0, o_1, \ldots, o_{2n-1}$.

**Diagonal Multiplier and Booth Multiplier:**

A diagonal multiplier uses a similar idea to the array multiplier. The difference is that the diagonal multiplier routes its carry bits to the next row instead of the same row as depicted in Figure 8.

A Booth multiplier uses a similar idea to the array multiplier, but uses a telescoping sum identity to skip consecutive digits in one multiplicand.

**Wallace Tree Multiplier:**

A Wallace tree multiplier takes a different approach to summing the tableau. Using carry-save adders (parallel 1-bit adders), it iteratively finds a new tableau with the same weighted sum as the previous tableau, but with 1/3 fewer rows. Upon reducing the original tableau to just two rows, it uses a carry-lookahead adder to obtain the final result. In contrast to the array multiplier, a Wallace tree multiplier is complicated to lay out physically, but has only logarithmic depth.
We construct a Wallace tree multiplier $C$ that multiplies the binary numbers $x, y$. We compute a tableau of partial products like in the array multiplier. We then go through $h \approx \log n$ steps to reduce the $n$-row starting tableau to an equivalent 2-row tableau. At this step we can use a carry-lookahead adder to compute the final sum.

We define tableau variables $t_{l,i,j}^C$ where $l$ is the layer of the tableau, $i$ is the index of the column containing the adder and $j$ is the row. We denote the set of tableau variables in a column by

$$\text{Col}^C(i) = \{t_{l,i,j}^C \text{ for all } l, j\},$$

and call the subset of a column within a layer $l$ a subcolumn, denoted by

$$\text{Col}^C(l, i) = \{t_{l,i,j}^C \text{ for all } j\}.$$  

In the zero-th layer, these tableau variables represent the partial products:

$$t_{0,i,j}^C = x_{i-j} \land y_j.$$  

We now specify how to construct layer $l + 1$ from layer $l$. For each column $i$ of layer $l$’s tableau containing three or more variables, we partition those variables into a number $p_{l,i}$ of sets $P_{l,i,j}^C$, each containing three variables (except possibly the last set). The partitions go in the following order:

$$P_{l,i,0}^C = \{t_{l,i,0}^C, t_{l,i-1,1}^C, t_{l,i-2,2}^C\},$$

$$P_{l,i,1}^C = \{t_{l,i-3,3}^C, t_{l,i-4,1}^C, t_{l,i-5,5}^C\},$$  

$$\ldots$$

Each partition $P_{l,i,j}$ is input to a full adder that is labeled by the partition $P_{l,i,j}$. We place the outputs of these adders row-by-row into the next layer $l + 1$. Precisely, for each fixed $j$, for each $i \in [0, 2n]$, we append adder $P_{l,i,j}$’s sum-bit to subcolumn $\text{Col}(l + 1, i)$. We then, for each $i$, append adder $P_{l,i,j}$’s carry-bit to subcolumn $\text{Col}(l + 1, i + 1)$. We do this for each $j$ in increasing order.

Each layer reduces the number of rows in the tableau from $N$ to $\lceil 2N/3 \rceil$. At the end of the tree, at the tableau for the last layer $h < \log_{3/2}(n) < 2 \log n$, we will reach a tableau with only two rows. The Wallace tree multiplier implements a $2n - 1$-bit carry-lookahead adder that sums the two rows, outputting the sum in the output bits $o_i^C$.

## 4 Efficient Proofs for Degree Two Array Multiplier Identities

We give polynomial-size resolution proofs that commutativity, distributivity, and $x(x+1) = x^2 + x$ hold for a correctly implemented array multiplier. We go on to give polynomial-size resolution proofs for general degree two identities.

**Proof Overview:** The main idea, common to our proofs for each circuit family including Wallace tree multipliers, is to start by branching according to the lowest order disagreeing output bit between the two circuits. In each of these branches, the subcircuit known as a critical strip consists of the constraints on a small number of columns behind the disagreeing bit. For a large enough choice of width this critical strip is unsatisfiable since the removed section of the tableau on the right does
not have enough total weight to cause the disagreeing output bit. It then remains to refute each critical strip.

Our proofs inside each critical strip cycle through three steps: branching on the values of input bits, propagating those values as far in the circuit as possible, then saving the resulting assignment to the variables on the boundary of the propagation. We call each of these boundaries a cut in the circuit. We can think of the branching program as scanning the input bits, saving its progress with assignments to small cuts.

These cuts are sets of variables that, under any assignment, split the strip into a satisfiable and an unsatisfiable region. If our branching program holds a cut assignment that was propagated from an earlier portion of the circuit, then this cut assignment is consistent with this earlier subcircuit. But since the critical strip as a whole is unsatisfiable, this cut assignment must be inconsistent with the rest of the circuit. Using these cuts, we isolate a small unsatisfiable region in the critical strip that is easily refuted.

One can view our proof as showing that the constraints within each strip form a graph of path-width $O(\log n)$ which, by [25], implies that there is a polynomial-size ordered resolution refutation of the strip. In the case of commutativity, our argument implies that the constraint graphs for the strips can be combined to yield a single constraint graph of pathwidth $O(\log n)$. For the other identities, the orderings on the strips are different and the resulting constraint graphs only have small branchwidth which, by [1], still implies that there are small regular resolution proofs of the other identities. Rather than simply invoke these general arguments, we give the details of the resolution proofs, along with more precise size bounds.

### 4.1 Efficient Resolution Proofs for Commutativity

**Definition** We define a SAT instance $\phi_{Array\ Comm}(n)$. The inputs are length $n$ bitvectors $x, y$. Using the construction from Section 3, we define array multipliers $L$ and $R$ computing, respectively, the products $xy$ and $yx$. In this construction, the tableau variables in multipliers $L$ and $R$ are defined by the constraints

$$t_{i,j}^L = x_i \land y_j,$$  
$$t_{i,j}^R = y_i \land x_j,$$

and in particular we can infer, through resolution, that $t_{i,j}^L = t_{j,i}^R$.

After specifying the subcircuits $L$ and $R$, we add a final subcircuit $E$, a set of inequality-constraints encoding that the two circuits disagree on some output bit:

$$e_i = (\neg o_i^L \land o_i^R) \lor (o_i^L \land \neg o_i^R) \quad \forall i \in [0, 2n - 1],$$

$$e_0 \lor e_1 \lor \ldots e_{2n-1}.$$

We give a small resolution proof for $\phi_{Array\ Comm}(n)$ in the form of a labeled OBDD $B$, as described in Theorem 2.1. The variable order for $B$ begins with the comparison bits in the order $e_0, e_1, \ldots$, followed by the output bits $o_0^R, o_1^R, \ldots$. Then $B$ reads the variables associated with adders $A_{i,j}^L, A_{j,i}^R$ in order of increasing $j$, reading each row right to left. Finally, $B$ reads the output bits $o_0^L, o_1^L, \ldots$, then the input bits $x, y$ in an arbitrary order.

At the root of $B$, we search for the first output bit that $L$ and $R$ disagree on by branching on the sequences of bits $e_k = 1, e_{k-1} = 0, \ldots e_0 = 0$ for each $k \in [0, 2n]$. We will show that on each branch we can prove that $\phi_{Array\ Comm}(n)$ is unsatisfiable using only the constraints from $L$ and $R$ on the variables inside columns $[k - \log n, k]$. 

Definition Let $\delta = \log n$. Let $\phi_{\text{Strip}}(k) \subset \phi_{\text{Comm}}^\text{Array}(n)$ hold the constraints containing a tableau variable $t^L_{i,j}$ or $t^R_{i,j}$ for $i + j \in [k - \delta, k]$. Further, add unit clauses to $\phi_{\text{Strip}}(k)$ that encode the assignment: $e_0 = 0, e_1 = 0, \ldots, e_{k-1} = 0, e_k = 1$ to the first $k$ bits of $e$. We call $\phi_{\text{Strip}}(k)$ a critical strip of $\phi_{\text{Comm}}^\text{Array}(n)$. We call the subset $\phi_{\text{Strip}}(k) \cap L$ the critical strip of circuit $L$ and likewise for circuit $R$.

Lemma 4.1. $\phi_{\text{Strip}}(k)$ is unsatisfiable for all $k$.

Proof. We interpret each critical strip as a circuit that outputs the weighted sum of the input variables in either circuit $L, R$. The assignment to $e$ demands that the difference between the critical strip outputs is precisely $2^k$. But by $t^L_{i,j} = t^R_{i,j}$, the weighted sum of the tableau variables is the same in both critical strips. The difference in the critical strip outputs is then bounded by the sum of the input carry bits to column $k - \delta$ in either strip. There are fewer than $n$ input carry bits for either critical strip, each of weight $2^{k-\delta} = 2^k/n$, therefore the difference in critical strip outputs is less than $2^k$, violating the assignment to $e$.

Observe that this proof only relied on the relation $t^L_{i,j} = t^R_{i,j}$ in the tableau variables. The additional requirement that the tableau variables came from an assignment to $x, y$ is unnecessary to refute $\phi_{\text{Strip}}(k)$.

Lemma 4.2. There is an $O(k^7 \log k)$-sized ordered resolution proof that $\phi_{\text{Strip}}(k)$ is unsatisfiable.

Proof. For simplicity we assume $k \leq n$; the case where $k > n$ is similar. We will also preprocess $\phi_{\text{Strip}}(k)$ by resolving on $x, y$ to obtain the tableau variable relations $t^R_{j,i} = t^L_{i,j}$, then replacing all the variables $t^R_{j,i}$ by $t^L_{i,j}$ in the clauses $\phi_{\text{Strip}}(k)$. Viewing the proof as a branching program, this amounts to querying $x, y$ at the end. We will not resolve on $x, y$ in the remainder of this proof.

We give this resolution proof in the form of a labeled read-once branching program $B$. We call the set of tableau variables of $L$ as well as the carry variables from column $k - \delta - 1$ of both $L$ and $R$, the input variables to this critical strip. Given an assignment to the input variables $\sigma_{\text{input}}$, we can propagate to the outputs of the critical strip. $L$ and $R$ can propagate this assignment to their outputs (with no additional branching).

The idea behind the branching program $B$ is to verify circuit $L$ by branching on input variables row-by-row, going from top-to-bottom, maintaining an assignment to a row of sum-variables. Since $t^L_{i,j} = t^R_{j,i}$, the tableau variables of circuit $R$ simultaneously get revealed from bottom to top. In circuit $R$ we maintain both a guess for its output values, and a row of sum-variables. From the proof of Lemma 4.1 if we have found that the outputs of $L$ and $R$ were computed correctly then they must violate one of the constraints $e_k = 0, \ldots, e_{k-\delta+1} = 0, e_{k-\delta} = 1$.

Definition Define $\text{Cut}(0)$ as the set of variables containing
\[ d^R_{0,i}, a^R_{i-1} \text{ for } i - 1 \in [k - \delta, k]. \]
For $j \in [1, k - \log k]$, we define $\text{Cut}(j)$ to be the set containing the variables:
\[ d^L_{i,j-1}, d^R_{j,i-1} \text{ for } i + j - 1 \in [k - \delta, k], \]
\[ c^R_{j-1,i} \text{ for } i + j - 1 \in [k - \delta, k - 1], \]
\[ o^R_i \text{ for } i \in [k - \delta, k]. \]
Lastly, for \( j \in [k - \delta, k] \), we define \( \text{Cut}(j) \) to be the set containing the variables, when the indices are in-range:

\[
\begin{align*}
\sigma_i^L & \quad \text{for } i \in [k - \delta, j - 1] \\
\sigma_i^R & \quad \text{for } i + j \in [k - \delta, k], \\
\sigma_{i+1,j-1}^R, \sigma_{j-1,i}^R & \quad \text{for } i + j - 1 \in [k - \delta, k - 1], \\
\sigma_i^R & \quad \text{for } i \in [k - \delta, k].
\end{align*}
\]

We will label each node of \( B \) by the pair \((\text{Cut}(j), \sigma)\) where \( \text{Cut}(j) \) keeps track of the previously seen cut.

**Initialization:** Throughout, we work in terms of the tableau variables in circuit \( L \), implicitly substituting \( t_{ij}^L \) for \( t_{ji}^R \). We begin at the root node of the read-once branching program \( B \), labeled \((\emptyset, \emptyset)\). For \( i \in [k - \delta, k] \) we branch on the variable \( \sigma_i^R \), then propagate to \( d_{0,i}^R \) using a clause from the constraint \( \sigma_i^R = d_{0,i}^R \). The surviving branches are those labeled by an assignment satisfying the constraints \( \sigma_i^R = d_{0,i}^R \). At this point we have reached nodes labeled \( \text{Cut}(0) \).

For each of the surviving branches, we branch on the tableau variables in the first row of \( L \):

\[
\begin{align*}
t_{i,0}^L & \quad \text{for } i \in [k - \delta, k],
\end{align*}
\]

Then we propagate to the variables, in sequence,

\[
\begin{align*}
d_{1,i}^R, c_{0,i}^R & \quad \text{for } i + 1 \in [k - \delta, k]
\end{align*}
\]
from Cut(1) (notice that this does not include the input carry-bit $c_{0,k-\delta-1}^R$). We then merge on Cut(1).

**Inductive Step:** We now describe the transition from Cut($j$) to Cut($j + 1$) for $j \in [1,k + 1]$. Suppose that the branching program $B$ has reached an assignment to Cut($j$). From these nodes we branch on the next, $j$-th row’s tableau variables

$$t_{i,j}^L \quad \text{for} \quad i + j \in [k - \delta,k]$$

and, when they exist, the pair of incoming input carry variables $c_{i,j}^L, c_{j-1,i}^R$ from column $k - \log k - 1$. We then propagate to the Cut($j + 1$) and $c^L$ variables in the sequence:

$$c_{i,j}^L, d_{i+1,j}^L \quad \text{for} \quad i + j + 1 \in [k - \log k, k]$$

in circuit $L$. If $j \in [k - \delta,k]$ then we also propagate to $o_{j-1}$. 

$$c_{j,i}^R, d_{j+1,i}^R \quad \text{for} \quad i + j + 1 \in [k - \log k, k]$$

in circuit $R$. After branching on the last variable in Cut($j + 1$) we start labeling nodes by Cut($j + 1$) and merge branches on their assignment to Cut($j + 1$). This completes the step from Cut($j$) to Cut($j + 1$).

We repeat this step until we have reached Cut($k + 1$). At this point we have an assignment to the critical strip output bits $o^L, o^R$. Furthermore, both output assignments were the result of, and therefore consistent with, propagating from a single assignment on the input variables $\sigma_{\text{inputs}}$. By the proof of Lemma 4.1 this implies that our assignment to $o^L, o^R$ conflicts with an inequality constraint.

**Size Bound:** We show that there are $O(k^7 \log k)$ nodes in $B$. Each Cut($j$) section of $B$ begins with an assignment to at most $4 \log k$ variables, so there are at most $k^4$ nodes labeled by an assignment to precisely Cut($j$). We branch on up to $\log(k) + 2$ input variables that propagate to the next cut’s variables. So each cut has a full binary tree of $2 \times (2^4k^2)$ nodes branching on different configurations of input variables. For each leaf of this tree, $B$ has a path of $O(\log k)$ nodes for propagating before the nodes get merged. Therefore each cut labels at most $O(k^6 \log k)$ nodes. There are $k + 1$ different cuts, thus $B$ has at most $O((k + 1)k^6 \log k) = O(k^7 \log k)$ nodes.

Since the tableau variables were actually partial products of $x$ and $y$, we can make this proof smaller by branching on the bits of $x, y$ to determine the tableau variables in a row, maintaining a sliding window of $\delta$ bits of $x$, yielding:

**Corollary 4.3.** $\phi_{\text{Strip}}(k)$ has an $O(k^6 \log k)$-size regular resolution refutation.

**Theorem 4.4.** Let $N = |\phi_{\text{Comm}}^{\text{Array}}| = O(n^2)$. There is an $O(N^{7/2} \log N)$ size regular resolution proof that $\phi_{\text{Comm}}^{\text{Array}}$ is unsatisfiable. There is an $O(N^{4} \log N)$ size ordered resolution proof that $\phi_{\text{Comm}}^{\text{Array}}$ is unsatisfiable.
Proof. We can now describe the overall branching program $B$ for $\phi_{\text{Comm}}(n)$. The branching program branches on the inequality-constraint assignments $\sigma_e(k) = \{e_k = 1, e_{k-1} = 0, \ldots, e_0 = 0\}$ for $k \in [0, 2n - 1]$. The $k$-th branch contains the clauses $\phi_{\text{Strip}}(k)$ so we can use the read-once branching program from either Corollary 4.3 or Lemma 4.2 (with each node augmented with the assignment $\sigma_e(k)$) to show that the branch is unsatisfiable. Corollary 4.3 yields the regular resolution proof and Lemma 4.2 yields the ordered resolution proof.

4.2 Efficient Resolution Proofs for Distributivity

Definition We define a SAT instance $\phi_{\text{Dist}}(n)$ to verify the distributivity property

$$x(y + z) = xy + xz$$

for an array multiplier in the natural way, using the constructions from Section 3. For the left hand expression we construct a ripple-carry adder $L_{y+z}$, outputting $(y + z)$, and array multiplier $L_{x(y+z)}$ outputting $x(y + z)$. For the right hand expression, we similarly define circuits $R_{xz}$, $R_{xy}$ and $R_{xy+xz}$.

We define $L = L_{y+z} \cup L_{x(y+z)}$ and $R = R_{xz} \cup R_{xy} \cup R_{xy+xz}$. We let $E$ contain the usual inequality constraints. The full distributivity instance is then $\phi_{\text{Dist}}(n) = L \cup R \cup E$.

We follow a similar strategy to the one used to refute $\phi_{\text{Comm}}$. We again divide the instance into critical strips.

Definition Define the constant $\delta = \log(2n)$. Define the subset $\phi_{\text{Strip}}(k) \subset \phi_{\text{Dist}}(n)$ to include, firstly, the full ripple-carry adder circuit $L_{y+z}$. Secondly, include the constraints containing one of the tableau variables $t_{i,j}^{L_{x(y+z)}}, t_{i,j}^{R_{xy}}, t_{i,j}^{R_{xz}}$ for $i + j \in [k - \delta, k]$. Thirdly, include the constraints on the carry-bits and sum-bits $c_i^{R_{xy+xz}}, d_i^{R_{xy+xz}}$ for $i \in [k - \delta, k]$. Lastly, add constraints to $\phi_{\text{Strip}}(k)$ that encode the assignment the bits: $e_k = 1, e_{k-1} = 0, \ldots, e_0 = 0$.

Lemma 4.5. $\phi_{\text{Strip}}(k)$ is unsatisfiable for all $k$

Proof. Similarly to the proof of Lemma 4.1, the critical strip for $L_{x(y+z)}$ holds tableau bits with the same weighted sum (modulo $2^{k+1}$) as those in $R_{xz}$ and $R_{xy}$ combined. The critical strip for $L_{x(y+z)}$ has at most $n$ input carry-bits of weight $2^{k-\delta}$. The critical strips of the $n$-bit multipliers $R_{xz}$ and $R_{xy}$ each have at most $n - 1$ input carry variables of weight $2^{k-\delta}$. The critical strip of the adder $R_{xy+xz}$ has one input carry variable, so the critical strip for $R$ has $2n - 1$ input carry-bits. Since we set the width of the strip at $\delta = \log(2n)$, it is unsatisfiable.

Lemma 4.6. For each $k$ there is an $O(n^4 \log n)$ size regular resolution proof that $\phi_{\text{Strip}}(k)$ is unsatisfiable.

Proof. We construct a labeled branching program $B$ that solves the conflict clause search problem for $\phi_{\text{Strip}}(k)$. We branch row-by-row in the critical strips, maintaining an assignment to cuts of variables in each multiplier. For each strip we will select a (different) variable ordering for $x, y, z$ that reveals the tableau variables row-by-row. Assume that $k < n$ for simplicity; the case where $k \geq n$ is similar.
Figure 10: Cut(2) for $\phi_{\text{Strip}}(4)$ consists of the blue variables.

For an array multiplier $C \in \{L_{x(y+z)}, R_{xz}, R_{xy}\}$ and $j \in [1, k - \delta]$ we define $\text{Cut}^C(j)$ to be the set of variables

$$d^C_{i,j-1} \text{ for } i + j - 1 \in [k - \delta, k],$$

and for $j \in [k - \delta + 1, k]$ we define $\text{Cut}^C(j)$ as the set of variables

$$d^C_{i,j-1} \text{ for } i + j - 1 \in [k - \delta, k],$$

$$o^C_i \text{ for } i \in [k - \delta, j - 2]\]$$

We define $\text{Cut}^{L_{y+z}}(j)$ to be the singleton set $\{c^L_{j-1}\}$. We also refer to a global cut, across the whole circuit: $\text{Cut}(j) = \bigcup_C \text{Cut}^C(j)$.

**Initialization: Getting to Cut(1)** At the root node $(\emptyset, \emptyset)$ of $B$, we branch on the circuit input variables $y_0, z_0$ and

$$x_i \text{ for } i \in [k - \delta, k].$$

We propagate these assignments to variables $c^L_{0}y+z$ and $o^L_{0}y+z$, giving us an assignment to $\text{Cut}^{L_{y+z}}(0)$. The assignment to $o^L_{0}y+z$, in turn, propagates to an assignment to the first row of tableau and sum variables from the Critical Strip for $L_{x(y+z)}$:

$$l^{L_{x(y+z)}}_{i,0}, d^{L_{x(y+z)}}_{i,0} \text{ for } i \in [k - \delta, k].$$

At this point we have an assignment to $\text{Cut}^{L_{x(y+z)}}(0)$.

We then propagate the input variable assignments through the multipliers $R_{xy}$ and $R_{xz}$:

$$l^{R_{xy}}_{i,0}, d^{R_{xy}}_{i,0} \text{ for } i \in [k - \delta, k],$$

$$l^{R_{xz}}_{i,0}, d^{R_{xz}}_{i,0} \text{ for } i \in [k - \delta, k],$$

obtaining assignments to $\text{Cut}^{R_{xy}}(0)$ and $\text{Cut}^{R_{xz}}(0)$, thus completing an assignment to Cut(0). At this point we merge nodes on assignment to Cut(0).
**Inductive Step:** Cut$(j)$ to Cut$(j + 1)$ Suppose we have merged branches and are at a node labeled with an assignment to Cut$(j)$. If this assignment contains a variable $d_{i,j}^C$, we propagate to $o_j^C$. We branch on input variables $x_{k-j}, y_j, z_j$. We then propagate these assignments to $c_{i,j}^{L_i+y+z}, o_j^{L_i+y+z}$, followed by the next row of tableau, carry, and sum variables in each multiplier:

$$c_{i-j-2, j+1}^C, c_{i-j-1, j+1}^C, d_{i-j-1, j+1}^C : i \in [k - \delta, k].$$

At this point we have reached an assignment to all of the variables in Cut$(j + 1)$ so we merge nodes based on Cut$(j + 1)$. We repeat this step until reaching an assignment to Cut$(k + 1)$, which consists of each multiplier’s output bitvector $o^C$.

**End:** Beyond Cut$(k + 1)$ Suppose we have reached Cut$(k + 1)$ and merged nodes. We branch on the input carry variable $c_{k-\delta-1}^{R_{xy+xz}}$, that goes into the Critical Strip of ripple-carry adder $R_{xy+xz}$. We can then propagate to the outputs $o^{R_{xy+xz}}$. We now have an assignment to both $o^{L_{z(y+z)}}, o^{R_{xy+xz}}$ that was propagated from one assignment to the input variables to the Critical Strip. By Lemma 4.5, this assignment conflicts with an Inequality Constraint from $E$.

**Size Bound:** There are $k + 1$ different global cuts Cut$(j)$. Each Cut$(j)$ section of $B$ begins with an assignment to at most $35 + 1$ variables. So each section Cut$(j)$ is initialized with at most $2^{35+1} = 8n^3$ branches. Each of these branches is a path with at most $10\delta$ queried variables and therefore at most $20\delta$ nodes. So there are at most $200\delta n^3$ nodes per cut and therefore at most $(k + 1)200\delta n^3 = O(n^4 \log n)$ nodes in $B$.

**Theorem 4.7.** Let $N = |\phi_{\text{Dist}}(n)| = O(n^2)$. There is an $O(N^{5/2} \log N)$ size resolution proof that $\phi_{\text{Dist}}(k)$ is unsatisfiable.

**Proof.** At the root of this proof there are $2n$ branches each holding an assignment to $e_k, \ldots, e_1, e_0$. We refute each branch using the $O(n^4 \log n)$ size proof from Lemma 4.6.

### 4.3 Efficient Resolution Proofs for $x(x + 1) = x^2 + x$

**Definition** We define a SAT instance $\phi_{x(x+1)}^\text{Array}(n)$. Circuit $L$ is composed of circuits $L_{(x+1)}$, consisting of a ripple-carry adder with inputs $x$ and $1$ and outputs $(x + 1)$, and $L_{x(x+1)}$, an array multiplier outputting the product $x(x + 1)$. Similarly, circuit $R$ is composed of circuits $R_{x^2}$ and $R_{x^2+x}$.

We let $E$ contain the usual Inequality-Constraints. The instance is then

$$\phi_{x(x+1)}^\text{Array}(n) = L \cup R \cup E.$$

While this identity looks like a special case of distributivity, its resolution proof is more complicated. This is because for distributivity: $x(y + z) = xy + xz$, the inputs to each multiplier were separate variables. If we try a similar strategy to scan the critical strip for the multiplier $R_{x^2}$ from top to bottom, we will read each $x_i$ twice. To avoid reading the same variable twice, we instead scan the critical strip from both ends, meeting in the middle.
Figure 11: An assignment to Cut(1) divides the critical strip $\phi_{\text{Strip}}(5)$ into a satisfiable and an unsatisfiable region.

**Definition** Define the constant $\delta = \log(2n - 1)$. Define the subset $\phi_{\text{Strip}}(k) \subset \phi_{\text{Array}}(n)$ to include the full ripple-carry adder circuit $L_{x+1}$. Also include the constraints containing one of the multiplier tableau variables $t_{i,j}^{L_{x+1}}$, $t_{i,j}^{R_{x^2}}$ for $i + j \in [k - \delta, k]$. Further include the constraints on the ripple-carry adder carry-bits and sum-bits $c_i^{R_{x^2+1}}, d_i^{R_{x^2+1}}$ for $i \in [k - \delta, k]$. Lastly, add constraints to $\phi_{\text{Strip}}(k)$ that encode the values of the bits: $e_k = 1, e_{k-1}, \ldots, e_0 = 0$.

We refer to the subcircuit $\phi_{\text{Strip}}(k) \cap C$ as the critical strip for $C$. Figure 11 shows an example of a critical strip.

**Lemma 4.8.** $\phi_{\text{Strip}}(k)$ is unsatisfiable for all $k$

**Proof.** The proof is the same as the proof for Lemma 4.5. \hfill \Box

**Definition** For either Array multiplier $C \in \{L_{x+1}, R_{x^2}\}$ and $j \in [1, (k - \delta)/2]$ we define $\text{Cut}^C(j)$ to be the set of variables

$$
d_{i,j-1}^C : \quad i + j - 1 \in [k - \delta, k], \quad \text{(upper cut)}
$$

$$
c_{i,j-1}^C, d_{i,j}^C : \quad i + j \in [k - \delta, k]. \quad \text{(lower cut)}
$$

We define $\text{Cut}^{L_{x+1}}(j)$ to be the set of variables

$$
\{x_i : \quad i \in [k - \delta - j + 1, k - j]\}.
$$

**Theorem 4.9.** There is a size $n^7 \log n$ regular resolution proof that $\phi_{\text{Strip}}(k)$ is unsatisfiable.
Proof. We give our proof in the form of a labeled read-once branching program $B$. We begin by branching on a guess for the critical strip outputs $o^{L_{x(x+1)}}$, $o^{R_{x+2}x}$. For the branches that don’t conflict with an Inequality-Constrain, we branch on a guess for the values $o_i^{R_{x^2},x_i} : i \in [k - \delta, k]$, then merge to erase the assignment to $o^{R_{x^2}x}$.

**Initialization: Getting to** Cut(1). We first observe that the carry variables in $L_{x+1}$ must be a sequence of 1s followed by 0s. If, on the contrary, we observe the assignments $c_i = 0$ and $c_j = 1$ for $i < j$, then we can find a conflict by scanning the portion of $L_{x+1}$ in columns $[i, j]$. So we can begin this proof by branching on the at most $n$ sequences of carry bits $c_0 = 1, \ldots, c_i = 1, c_{i+1} = 0, \ldots, c_k = 0$.

We branch on input variables $x_0$ and $x_k, x_{k-1}, \ldots, x_{k-\delta}$ and then propagate to an assignment to the upper and lower cuts in each circuit, then merge on the assignment to Cut(1).

From Cut($j$) to Cut($j+1$), we have two cases: the upper and lower cuts of Cut($j+1$) either intersect or they do not. In either case we branch on input variables $x_{j-1}, x_{k-\delta-j+1}$ and the input carry variables to rows $j$ and $k-\delta-j+1$. If the cuts do not intersect, we propagate to, then merge on, all the Cut($j+1$) variables. Otherwise, suppose that the upper and lower cuts of Cut($j+1$) intersect on $d_{ij}$. The upper and lower cuts of Cut($j$) either propagate to conflicting values of $d_{ij}$, in which case we have found a conflict, or they agree on the value of $d_{ij}$, in which case we delete column $i+j$ from our cuts.

**Size Bound** There are less than $n$ cuts. Each cut belongs to one of up to $n$ branches for the carry variables in $L_{x+1}$ and holds an assignment to at most $6 \log n$ variables so there are at most $n^7$ initial nodes for each cut. Each of these nodes propagates for $O(\log n)$ steps to get to the next cut, so our branching program has size $O(n^7 \log n)$.

We can now obtain a refutation for $\phi_{Array,x(x+1)}(n)$ by branching on sequences of variables in $e$ and using the refutation for $\phi_{Strip}(k)$ on each branch.

**Theorem 4.10.** There is a size $n^8 \log n$ regular resolution proof that the SAT instance $\phi_{Array,x(x+1)}(n)$ is unsatisfiable.

### 4.4 Efficient Resolution Proofs for Degree Two Identities

Let $\phi^{Array}_{L=R}(n)$ denote a SAT instance corresponding to verifying the ring identity $L = R$. Similarly define $\phi^{Diag}_{L=R}(n)$ and $\phi^{Booth}_{L=R}(n)$ denote the diagonal and Booth multiplier versions. With the insight from the earlier proofs in this section, we can prove the general theorem:

**Theorem 4.11.** For any degree two ring identity $L = R$, there are polynomial size regular refutations for $\phi^{Array}_{L=R}(n)$. 

Proof. (Sketch) We divide $\phi^\text{Array}_{L=R}(n)$ into unsatisfiable critical strips of width $\delta = \log mn$, where $m$ is the number of terms in the identity $L = R$. The ripple-carry adders that input to a multiplier remain intact, and for the rest we remove the columns outside the critical strip.

We begin by branching on guesses for the $\delta$ output bits from each multiplier and each truncated ripple-carry adder. In each multiplier we use a ”meet-in-the-middle” strategy, similar to the proof for $x(x + 1) = x^2 + x$. We read all the input bitvectors in parallel, each in the same order. This branch order for each input bitvector is $x_0, x_n, x_1, x_{n-1}, \ldots$. We can propagate these assignments to diagonal cuts in each multiplier that scan from the top and bottom edges towards the middle, and likewise for the intact ripple-carry adders. In each input bitvector we remember the assignment to just the most recently queried $2\delta$ variables. Because of the symmetry of this variable order, it is compatible with swapping the order of inputs to any multiplier, as well as multipliers squaring an input.

Corollary 4.12. For any degree two ring identity $L = R$, there are polynomial size regular resolution proofs for $\phi^\text{Diag}_{L=R}(n)$ and $\phi^\text{Booth}_{L=R}(n)$

Proof. The proofs for diagonal and Booth multipliers follow the same ideas as the proof for an array multiplier.

5 Wallace Tree Multipliers

Like the proofs for array multipliers, our proofs for Wallace tree multipliers divide the instance into critical strips. In fact, our proofs branch on the input tableau in the same row-by-row order in both array and Wallace tree multipliers. However the size of the resulting cuts is $O(\log^2 n)$ for Wallace tree multipliers rather than the $O(\log n)$ size cuts for array multipliers. These cuts result in quasipolynomial size regular resolution proofs.

When analyzing the cuts in a Wallace tree multiplier, we will find the following property useful:

Definition For layer $l$ of a Wallace tree multiplier, if for each $j \leq k$, the $j$-th row of adders

$$P_{i,j} \text{ for all } i$$

outputs only to rows $2j, 2j + 1$ of the next layer $l + 1$’s tableau, we say that layer $l$ is row-friendly up to its $k$-th row of adders. If layer $l$ is row-friendly up to its last, $h_l - 1$-th, row of adders, we say that layer $l$ is row-friendly.

Lemma 5.1. In a Wallace tree multiplier, each layer $l \in [0, h - 2]$ is row-friendly.

In terms of the dot diagram in Figure [12] this Lemma simply states that no two bits are connected with a line of slope greater than one.

In order to prove this Lemma, we will first prove the following smooth and singly-peaked properties of each layer of a Wallace tree multiplier. Define $\#\text{Col}(l,i)$ to be the number of variables in subcolumn $\text{Col}(l,i)$.

Definition We say that a layer $l$ is smooth if for all pairs of adjacent subcolumns $\text{Col}(l,i)$ and $\text{Col}(l,i - 1)$, we have

$$|\#\text{Col}(l,i) - \#\text{Col}(l,i - 1)| \leq 2.$$
Figure 12: Dot diagram showing a $9 \times 9$ Wallace tree multiplier. Blue dots represent carry-bits and green dots represent sum-bits. Dots connected by an edge are outputs to the same adder.
We say that a function $f : [0, N] \rightarrow \mathbb{Z}$ is *singly-peaked* if there exists an integer $k$ such that the following inequalities hold:

$$f(0) \leq f(1) \leq \ldots \leq f(k),$$

$$f(k) \geq f(k + 1) \geq \ldots \geq f(N).$$

For any $k$ satisfying these inequalities, we say that $f$ attains its *peak* at $k$.

We say a layer $l$ of a Wallace tree Multiplier is *singly-peaked* if the function $\# \text{Col}(l, i)$ of the variable $i$ is singly-peaked. If $\# \text{Col}(l, i)$ attains its peak at $k$, we also say that layer $l$ attains its peak at $k$.

**Proposition 5.2.** All layers of a Wallace tree multiplier are smooth.

**Proof.** It is clear that the initial layer is smooth. Assume, for induction, that layer $l - 1$ is smooth. From our construction, for layers $l > 0$ we have the recurrence

$$\# \text{Col}(l, i) = \left\lceil \frac{1}{3} \# \text{Col}(l - 1, i) \rightceil + \left\lceil \frac{1}{3} \# \text{Col}(l - 1, i - 1) \right\rceil$$

(1)

where the left term counts the number of sum-variables in $\text{Col}(l, i)$ and the right term counts the number of carry-variables. Therefore we can write

$$|\# \text{Col}(l, i) - \# \text{Col}(l, i - 1)| = \left| \left\lceil \frac{1}{3} \# \text{Col}(l - 1, i) \right\rceil - \left\lceil \frac{1}{3} \# \text{Col}(l - 1, i - 2) \right\rceil \right|.$$  

To bound this expression, we use that by the smoothness of layer $l - 1$:

$$|\# \text{Col}(l - 1, i) - \# \text{Col}(l - 1, i - 2)| \leq 4.$$  

This implies the bound

$$\left| \left\lceil \frac{1}{3} \# \text{Col}(l - 1, i) \right\rceil - \left\lceil \frac{1}{3} \# \text{Col}(l - 1, i - 2) \right\rceil \right| \leq 2.$$  

\[\square\]

**Proposition 5.3.** All layers of a Wallace tree multiplier are singly-peaked.

**Proof.** The initial layer is singly-peaked. Assume, for induction, that layer $l - 1$ is singly-peaked and attains a maximum at $k$. We show that layer $l$ is singly-peaked and attains its peak at $k$ or $k + 1$. Roughly, this follows because, from Equations (1), $\# \text{Col}(l, i)$ is a sum of two singly-peaked functions, one attaining its peak at $k$ and the other attaining its peak at $k + 1$.

Consider the case where $i < i' \leq k$. We will show that $\# \text{Col}(l, i) \leq \# \text{Col}(l, i')$. We have, from Equations (1) that

$$\# \text{Col}(l, i) = \left\lceil \frac{1}{3} \# \text{Col}(l - 1, i) \right\rceil + \left\lceil \frac{1}{3} \# \text{Col}(l - 1, i - 1) \right\rceil$$

(2)

$$\# \text{Col}(l, i') = \left\lceil \frac{1}{3} \# \text{Col}(l - 1, i') \right\rceil + \left\lceil \frac{1}{3} \# \text{Col}(l - 1, i' - 1) \right\rceil.$$
Since \( \# \text{Col}(l - 1, x) \) was singly-peaked in \( x \), attained its peak at \( k \), and \( i - 1 < i \leq i' - 1 < i' \leq k \), we have

\[
\# \text{Col}(l - 1, i) \leq \# \text{Col}(l - 1, i') \\
\# \text{Col}(l - 1, i - 1) \leq \# \text{Col}(l - 1, i' - 1),
\]

implying, through Equations 2, that \( \# \text{Col}(l, i) \leq \# \text{Col}(l, i') \).

Consider the other case where \( k + 1 \leq i < i' \). We will show that \( \# \text{Col}(l, i) \geq \# \text{Col}(l, i') \). Since \( \# \text{Col}(l - 1, x) \) is singly-peaked in \( x \), attained its peak at \( k \), and \( k \leq i - 1 < i \leq i' - 1 < i' \leq k \), we have

\[
\# \text{Col}(l - 1, i) \geq \# \text{Col}(l - 1, i') \\
\# \text{Col}(l - 1, i - 1) \geq \# \text{Col}(l - 1, i' - 1),
\]

implying, through Equations 2, that \( \# \text{Col}(l, i) \geq \# \text{Col}(l, i') \).

Lastly, either \( \# \text{Col}(l - 1, k) \geq \# \text{Col}(l - 1, k + 1) \) in which case \( \# \text{Col}(l, i) \) is singly-peaked, attaining its peak at \( k \), or \( \# \text{Col}(l - 1, k) < \# \text{Col}(l - 1, k + 1) \) and \( \# \text{Col}(l, i) \) attains its peak at \( k + 1 \).

Together, these two properties imply that the rows of adders in each layer are arranged in a shallow pyramid.

**Proposition 5.4.** In each layer \( l \) of a Wallace tree multiplier:

1. Each row \( j \) of adders occupies a contiguous interval of columns between \( \text{minCol}(j) \) to \( \text{maxCol}(j) \).

2. Each row \( j \) of adders is arranged from the middle-out with full adders in the middle, then half adders, then wires at the ends.

3. For row \( j + 1 > 0 \), we have the strict inequalities

\[
\text{minCol}(j) < \text{minCol}(j + 1) \quad (3) \\
\text{maxCol}(j) > \text{maxCol}(j + 1). \quad (4)
\]

**Proof.** Properties (1) and (2) follow from Proposition 5.3: layer \( j \) is singly-peaked.

(Proof of 3.) The (non-strict) inequalities

\[
\text{minCol}(j) \leq \text{minCol}(j + 1) \quad (5) \\
\text{maxCol}(j) \geq \text{maxCol}(j + 1). \quad (6)
\]

follow from layer \( j \) being singly-peaked. The strictness of these inequalities follows from Proposition 5.2 which states that layer \( j \) is smooth. For contradiction, suppose that

\[
\text{minCol}(j) = \text{minCol}(j + 1).
\]

Then we have at least two more adders in \( \# \text{Col}(\text{minCol}(j)) \) compared to the previous column. These are either full adders or half adders, therefore

\[
\# \text{Col}(\text{minCol}(j)) - \# \text{Col}(\text{minCol}(j) - 1) \geq 4
\]

contradicting the smoothness of layer \( j \).
Having established this structure on the arrangement of adders in each layer, we prove Lemma 5.1. Recall the definition and statement of Lemma:

**Definition** For layer $l$, if for all $j \leq k$ the $j$-th row of adders

$$P_{l,i,j} \text{ for all } i$$

outputs only to rows $2j, 2j + 1$ of the next layer $l + 1$’s tableau, we say that layer $l$ is *row-friendly* up to its $k$-th row of adders. If layer $l$ is row-friendly up to its last $h_l - 1$-th row of adders, we say that layer $l$ is *row-friendly*.

**Lemma 5.5.** In a Wallace tree multiplier, each layer $l \in [0, h - 2]$ is row-friendly.

**Proof of Lemma 5.5.** Fix a layer $l$. We show that if $l$ satisfies the properties in Proposition 5.4 then it is row-friendly. We will use induction on the rows of adders in $l$.

**Definition** An interval of columns from $[i, i']$ is flat if they all contain the same number of variables.

It is clear from the Wallace tree multiplier construction that the zero-th row of adders

$$P_{l,i,0} \text{ for all } i \in [\minCol(0), \maxCol(0)]$$

outputs only to rows 0 and 1 so it is row-friendly. Furthermore, from our construction and 5.4 after placing this row of adder outputs in layer $l + 1$, the resulting columns are flat from $i \in [\minCol(0) + 1, \maxCol(0)]$.

Assume, as an induction hypothesis, $l$ was row-friendly up to row $j$ and furthermore, that after the outputs for the $j$-th row of adders have been placed, layer $l + 1$ is flat from columns

$$i \in [\minCol(j) + 1, \maxCol(j)].$$

By the third property of 5.4 we can restate this, saying that layer $l + 1$ is flat from columns

$$i \in [\minCol(j + 1), \maxCol(j + 1) + 1].$$

Since we added $j$ rows of adders to these columns, each column contains $2j$ variables. These are precisely the columns in which when we place the outputs for the $j + 1$-th row of adders. They occupy rows $2j$ and $2j + 1$ so that $l$ is row-friendly up to row $j + 1$. Additionally, we observe that if we append the $j + 1$-th row of adder outputs to the flat interval of columns

$$[\minCol(j + 1), \maxCol(j + 1) + 1],$$

the resulting columns are flat from

$$[\minCol(j + 1) + 1, \maxCol(j + 1)].$$

$\Box$
5.1 Efficient Proofs for Commutativity

**Definition** We define a SAT instance $\phi_{\text{Wall}}^{\text{Comm}}(n)$. The inputs to the multipliers are $n$-bit integers $x, y$. Using the construction from Section 3, we define Wallace tree multipliers $L, R$, computing $xy$, and $R$, computing $yx$ (reversing the order of multiplier inputs).

After specifying the circuits $L$ and $R$, we add a circuit $E$, of of inequality-constraints encoding that the two circuits disagree on some output bit.

**Definition** Define the constant $\delta = \log 2n + \log \log n$. Let $\phi_{\text{Strip}}(k) \subset \phi_{\text{Wall}}^{\text{Comm}}(n)$ be the subset of constraints containing a tableau variable $t^{L}_{i,j}$ or $t^{R}_{i,j}$ for $i \in [k-\delta, k]$, as well as the full CLA at the end of the Wallace tree multiplier. Further, add unit clauses to $\phi_{\text{Strip}}(k)$ that encode the values: $e_0 = 0, e_1 = 0, \ldots, e_{k-1} = 0, e_k = 1$.

**Lemma 5.6.** $\phi_{\text{Strip}}(k)$ is unsatisfiable for all $k$.

*Proof.* We reason similarly to the proof of Lemma 4.1. Again, we interpret the critical strip as a circuit that computes the weighted sum, in both $L$ and $R$, of the tableau variables and carry variables from column $k - \delta - 1$. Let $\text{out}^C$ be the value of the binary number $\delta^C, \delta^C_{k-1}, \ldots, \delta^C_{k-\delta}$. The subcircuit $\phi_{\text{Strip}}(k)$ consists of the clauses from adders in columns $[k - \delta, k]$ from each layer’s tableau, in both circuits $L$ and $R$. Each of these strips has less than $n/3$ unconstrained tableau bits in column $k - \delta$ that were outputs of adders from the removed column $k - \delta - 1$. Since there are less than $2\log n$ layers, the critical strip for the entire Wallace tree multiplier has, in total, less than $2n/3 \log n$ unrestricted carry bits of weight $2^{k-\delta-1}$ going into the least-indexed column of the strip.

Also, the newly unconstrained inputs to the final CLA from the removed columns can contribute a total weight of at most $2^{k-\delta-1}$ to the final output. Since we set $\delta = \log 2n + \log \log n$, the total difference between the final outputs is at most $2^{k-\delta-1}(2n/3 \log n + 1) < 2^k$. \hfill $\square$

**Lemma 5.7.** There is a regular resolution proof of size $2^{O(\log^2 n)}$ that $\phi_{\text{Strip}}(k)$ is unsatisfiable.

*Proof.* The idea of this proof is to read the initial layer of the critical strip row-by-row. If we have assigned all of the inputs to some adder, we propagate to the adder’s output bits. In this way we will compute output bits in both circuits coming from one input to $x$ and $y$. Furthermore, we will only keep track of a constant number of variables in each subcolumn. From the proof of 5.6, the result will contradict one of the inequality-constraints from $\phi_{\text{Wall}}^{\text{Comm}}(n)$.

Before describing the branching program $B$, we preprocess the constraints to obtain the equalities

$$t^L_{0,i,j} = t^R_{0,i,j-1}.$$ 

We define $B$ using an algorithm that makes use of subroutines $\text{PropagatePair}(P^L_{0,i,j}, P^R_{0,i,j'})$, which handles propagations in the initial layer, and $\text{PropagateAdder}(P^C_{i,j})$, which handles propagations in the subsequent layers. For efficiency, these algorithms order the propagations in a depth-first fashion, with the lowest-order bits first.

We partition the zero-th layer tableau variables into Windows, each corresponding to a row of adders:

$$\text{Window}^L(i) = \bigcup_i P^L_{0,i,j},$$

$$\text{Window}^R(i) = \bigcup_i P^R_{0,i,p0,i-j-1}.$$ 

We label nodes of $B$ according to the last fully read window of circuit $L$. 

---

*Page 25*
**Algorithm 1** PropagateAdder($P^C_{l,i,j}$) where $C \in \{L, R\}$

**Require:** At a node labeled by assignment $\sigma$ where $\sigma$ contains an assignment to $P^C_{l,i,j}$.
- Propagate to the sum-variable $s$ and the carry-variable $c$ of $P^C_{l,i,j}$.
- Merge branches based on forgetting the assignment to $P^C_{l,i,j}$.
- if Adder $P^C_{l+1,k-\delta,j'}$ is fully assigned except for a set, $S$, of at most two unrestricted carry bits then
  - Branch on unrestricted carry bits in $S$
  - PropagateAdder($P^C_{l+1,k-\delta,j'}$)
endif
- if Propagating to $s$ filled an assignment to adder $P^C_{l+1,i,j'}$ then
  - PropagateAdder($P^C_{l+1,i,j'}$)
endif
- if Propagating to $c$ filled an assignment to adder $P^C_{l+1,i+1,j''}$ then
  - PropagateAdder($P^C_{l+1,i+1,j''}$)
endif

**Algorithm 2** PropagatePair($P^L_{0,i,j}, P^R_{0,i,j'}$)

**Require:** At a node labeled by assignment $\sigma$ containing an assignment to the adders $P^L_{0,i,j}, P^R_{0,i,j'}$.
- Propagate to the sum and carry variables $s, c$ of the adder $P^L_{0,i,j}$.
- Merge branches based on forgetting the assignment to $P^L_{0,i,j} \cap P^R_{0,i,j'}$, the tableau variables contained in both adders.
- for each adder $P$ that became fully assigned except for a (possibly empty) set $S$, of at most two unrestricted carry bits, from least to most significant column, do
  - PropagateAdder($P$)
end for
Figure 13: Showing the overlap of inputs to the zero-th layer adders in $L$ and $R$.

Observe that we can propagate an assignment from the initial tableau variables $\text{Window}_L(0)$ to $\text{Window}_R(0)$. Similarly, as shown in Figure 13, we can propagate an assignment on $\text{Window}_L(j - 1) \cup \text{Window}_R(j)$ to $\text{Window}_R(j)$. Thus, reading the windows from $L$ in the sequence $\text{Window}_L(0), \text{Window}_L(1), \ldots$ reveals the windows from $R$ in tandem. We show how to compute outputs $o^L_i$ consistent with the assignments to the sequence of windows $\text{Window}_L(j)$.

**Algorithm 3** Branching Program Algorithm for $\phi_{\text{Strip}}(k)$.

```
Begin at root node (Window$_L(-1), \emptyset$).
for $j = 0, 1, \ldots, p_0, n - 1$ do
    for $i = k - \delta, \ldots, k$ do
        Branch on the variables in adder $P^L_{0,i,j}$.
        Propagate to the tableau variables in adder $P^R_{0,i,p_0,i-j-1}$.
        PropagatePair($P^L_{0,i,j}, P^R_{0,i,p_0,i-j-1}$).
    end for
end for
```

Each surviving branch, after running Algorithm 3, contains a full assignment to the final layer $h$ of tableau variables in the strip that is consistent with the revealed input variables. It remains to describe how to efficiently propagate this assignment to the final outputs through the ending CLA. The method is more complicated than with a ripple-carry adder because each CLA carry variable inside the critical strip can possibly depend on calculated values from outside the critical strip.

From Lemma 5.6 any assignment to the tableau variables to the right of the strips $L$ and $R$ will produce final outputs that conflict with an inequality-constraint. So we just need to compute each carry within the critical strip consistently with any such assignment. In either circuit $C$ we will sequentially branch on the CLA inputs $t^C_{h,i,0}, t^C_{h,i,1}$ to the right of the critical strip, in order of increasing column. For each pair outside the critical strip, we compute the corresponding propagate and generate values $p^C_{h,i,0}, g^C_{h,i,0}$, then merge to forget $t^C_{h,i,0}, t^C_{h,i,1}$. When we have an assignment to the carry $c$, four generate, and four propagate variables that input to some sub-CLA $CLA_l$ at layer $l$, we propagate to the output group-propagate and group-generate going into a larger sub-CLA $CLA_{l+1}$ on layer $l + 1$. If the input carry $c$ was the most significant carry of the sub-CLA $CLA_l$ then we merge to forget the inputs to $CLA_l$ except for the least significant carry, computed at a
layer after \( l \). In this way we ensure that each layer holds an assignment to at most a constant number of variables so that overall we keep track of an assignment to \( O(\log n) \) variables. After branching on each pair \( t^C_{h,i,0}, t^C_{h,i,1} \) in the above fashion, we will arrive at an assignment to \( O(\log n) \) propagate, generate and carry variables that can be used to compute each carry, and then each output, within the strip. An intermediate stage in this procedure is shown in Figure 14.

After running the above procedure, we have an assignment to the outputs of both critical strips. By Lemma 5.6, this assignment violates an inequality-constraint in \( E \).

**Size Bound:** We claim that each node in \( B \) is labeled by an assignment to a constant number of variables per subcolumn in the critical strip. To do this, we show that the adders \( P^L_{l,i,j} \) become assigned in order of increasing \( j \), row-by-row. We can show that the adders \( P^R_{l,i,j} \) become assigned in order of decreasing \( i \) by a symmetric argument.

It is clear that the initial layer’s adders are assigned to in row order. Suppose that layer \( l \) was assigned in row order. We claim that for two either half or full adders \( P_{l+1,i,j}, P_{l+1,i',j'} \) with \( j' > j \), the adder \( P_{l+1,i',j'} \) from the later row does not become fully assigned before the earlier row’s adder \( P_{l+1,i,j} \). Recall that by Lemma 5.1, the \( j \)-th row of adders in layer \( l \) outputs to precisely rows \( 2j \) and \( 2j + 1 \) of tableau variables in layer \( l + 1 \). The claim then follows because we read layer \( l \)’s adders row-by-row.

The claim implies that during the run of Algorithm 3 the variables within each subcolumn \( \text{Col}^C(l,i) \) are read in order of increasing row \( j \). In our algorithm, in between reading adders from the initial layer, we make as many propagations as possible. So in particular, for adders \( P_{l,i,j}, P_{l,i,j+1} \) in a subcolumn, we will have propagated from and forgotten the variables in \( P_{l,i,j} \) before \( P_{l,i,j+1} \) is fully assigned. Therefore no node in our branching program \( B \) assigns more than five variables to any subcolumn of \( L \). Symmetrically, we can argue that no node assigns more than two variables to any subcolumn of \( R \).

Now we can bound the number of nodes in \( B \). The portion of \( B \) that propagates through the ending CLA is polynomial in size since each node only holds an assignment to \( O(\log n) \) variables. In the Wallace tree portion of the circuit, each node holds an assignment to at most \( 4h\delta \leq 4(2 \log n)(\log 2n + \log \log n) \leq 16 \log^2 n \) variables. Since \( B \) was read-once and there are \( O(n^2) \) variables in the representation of the Wallace tree multiplier, each path in \( B \) has length
Furthermore, $B$ is oblivious since the steps of algorithm 3 only rely on the set of variables that are assigned. So for a node assignment, there are $O(n^2)$ possible distinct sets of at most $20 \log^2 n$ variables each. Therefore the number of nodes in $B$ is upper bounded by

$$O(n^2)2^{16\log^2 n} = 2^{O(\log^2 n)}.$$ 

A tighter analysis can reduce the constant in the exponent giving the upper bound

$$2^{3\log^2 n + 3}.$$ 

**Theorem 5.8.** There is a regular resolution proof of size $2^{O(\log^2 n)}$ that $\phi_{\text{Comm}}^\text{Wall}(n)$ is unsatisfiable.

**Proof.** Like before, we initially branch on the assignments $\sigma_e(k) = \{e_0 = 0, e_1 = 0, \ldots e_k = 1\}$ for $k \in [0, 2n - 1]$. The $k$-th branch contains the clauses $\phi_{\text{Strip}}(k)$ so we can use the Read-Once branching program from Lemma 5.7 (with each node augmented with the assignment $\sigma_e(k)$) to show that the branch is unsatisfiable.

The size of this proof is $2n 2^{O(\log^2 n)} = 2^{O(\log^2 n)}$. 

**5.2 Efficient Proofs for Distributivity**

Like with array multipliers, our proof of commutativity for Wallace tree multipliers used an algorithm to efficiently propagate an assignment to the input variables of $L$’s critical strip to its outputs. We will similarly use this algorithm to verify the distributivity of Wallace tree multipliers.

**Definition** Define a SAT instance $\phi_{\text{Dist}}^\text{Wall}(n)$ encoding the identity $x(y + z) = xy + xz$ in the usual way, with subcircuits $L_{x(y+z)}$, $L_{x(y+z)}$ forming circuit $L$, $R_{xy}$, $R_{xz}$, $R_{xy+xz}$ forming circuit $R$ and inequality-constraints $E$.

**Theorem 5.9.** There is a regular resolution proof of size $2^{O(\log^2 n)}$ that $\phi_{\text{Dist}}^\text{Wall}(n)$ is unsatisfiable.

**Proof.** (Sketch) We sketch the proofs for distributivity, as they are simpler than the proofs for commutativity. We define critical strips in the usual way for each multiplier. There are less than $2(2n+1)/3 \log(2n+1)$ unconstrained carry bits incoming to the critical strip for the $n+1$-bit multiplier $L_{x(y+z)}$, and less than $4n/3 \log n$ incoming to the critical strip for $R$. Hence if our critical strip has width $\delta = \log(4n/3 \log n) = \log(4n/3) + \log \log n$, it will be unsatisfiable.

To prove that a critical strip $\phi_{\text{Strip}}(k)$ from $\phi_{\text{Dist}}^\text{Wall}(n)$ is unsatisfiable we branch on the input variables in the same row-by-row order as in the proof of Theorem 4.7 to reveal the initial tableaus of each multiplier row-by-row. Each time all of the inputs to an adder have been assigned, we propagate to its outputs and erase the assignment to its input tableau variables. Upon reaching an assignment to the last layer’s tableau, we use the same method from the proof of commutativity to propagate this assignment through the CLA to the outputs.

Because we revealed the initial tableaus row-by-row and a Wallace tree multiplier is row-friendly, each subcolumn gets assigned from top to bottom. So we never hold an assignment to more than two variables in a subcolumn. That is, we never hold an assignment to more than $3\times 4\delta \log n = O(\log^2 n)$ variables across the three multipliers. Thus our branching program has $2^{O(\log^2 n)}$ nodes. A tighter analysis can reduce the constant in the exponent giving the upper bound $2^{4.5 \log^2 n + 2}$. 

29
Using the same ordering on the input variables and ideas from the proof of Theorem 4.11, we can prove the analogous result for Wallace tree multipliers.

**Theorem 5.10.** For any degree two ring identity $L = R$, there are quasipolynomial size regular refutations for $\phi_{L=R}^{\text{Wall}}(n)$.

6 Proving Equivalence Between Multipliers

Given any two $n$-bit multiplier circuits $\otimes_1$ and $\otimes_2$ we can define a Boolean formula $\phi_{\otimes_1=\otimes_2}$ encoding the (negation of the) identity $x \otimes_1 y = x \otimes_2 y$ between length $n$ bitvectors $x$ and $y$.

If they are both correct and compute using the typical tableau for multipliers then, as before, we can split $\phi_{\otimes_1=\otimes_2}$ into unsatisfiable critical strips. We can scan down both strips row-by-row, as in the proofs for commutativity and distributivity. If we have reached the outputs of both multipliers without finding an error, these outputs will disagree with the inequality-constraints for the critical strip. For our examples this method yields polynomial-size proofs if neither is a Wallace tree multiplier, and quasi-polynomial size proofs otherwise.

On the other hand, if one multiplier is incorrect and the other is not, then the proof search will yield a satisfying assignment in the appropriate critical strip.

In the more general case where a multiplier does not use the typical tableau, one can label each internal gate by the index of the smallest output bit to which it is connected and focus on comparing subcircuits labeled by $O(\log n)$ consecutive output bits, as we do with critical strips. The complexity of this equivalence checking will depend somewhat on the similarity of the circuits involved.

7 Discussion

Despite significant advances in SAT solvers, one of their key persisting weaknesses has been in equivalence checking of arithmetic circuits. This pointed towards the conjecture that the corresponding resolution proofs are exponentially large; if true, this would have been a fundamental obstacle putting nonlinear arithmetic out of reach for any CDCL SAT solver. We have shown that no such obstacle exists by giving the first small resolution proofs for verifying any degree two ring identity for the most common multiplier designs. Given the historical success of CDCL SAT solvers for finding specific proofs, our results suggest a new path towards verifying nonlinear arithmetic.

We introduced a method of dividing each instance into narrow, but still unsatisfiable, critical strips that is sufficiently general to yield short proofs for a wide variety of popular multiplier designs. The proof size upper bounds we derived were conservative; we did not try to optimize the parameters. A more important direction than doing so is to find the right guiding information to add, either to the formulas derived from the circuits or to CDCL SAT solver heuristics, to help them find such short proofs.

It also remains open to find a small resolution proof verifying the last ring property, associativity $(xy)z = x(yz)$. Our critical strip idea alone does not seem to work: while we can divide the outer multipliers into narrow critical strips, the $yz$ or $xy$ multipliers remain intact. These critical strips do not seem to have small cuts. Finding efficient proofs of associativity, combined with our results for degree two identities, could yield small proofs of any general ring identity.
References

[1] Michael Alekhnovich and Alexander A. Razborov. Satisfiability, branch-width and tseitin tautologies. In 43rd Symposium on Foundations of Computer Science (FOCS 2002), Proceedings, pages 593–603, Vancouver, BC, Canada, November 2002. IEEE Computer Society.

[2] Gunnar Andersson, Per Bjesse, Byron Cook, and Ziyad Hanna. A proof engine approach to solving combinational design automation problems. In Proceedings of the 39th Design Automation Conference, DAC 2002, pages 725–730, New Orleans, LA, USA, June 2002. ACM.

[3] Fabrício Vivas Andrade, Márcia C. M. Oliveira, Antônio Otávio Fernandes, and Claudionor José Nunes Coelho Jr. Sat-based equivalence checking based on circuit partitioning and special approaches for conflict clause reuse. In Patrick Girard, Andrzej Krasniewski, Elena Gramatová, Adam Pawlak, and Tomasz Garbolino, editors, Proceedings of the 10th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2007), pages 397–402, Kraków, Poland, April 2007. IEEE Computer Society.

[4] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. J. Artif. Intell. Res. (JAIR), 22:319–351, 2004.

[5] Armin Biere. Challenges in bit-precise reasoning. In Formal Methods in Computer-Aided Design, FMCAD 2014, page 3, Lausanne, Switzerland, October 2014.

[6] Armin Biere. Where does SAT not work? In BIRS Workshop on Theory and Applications of Applied SAT Solving, January 2014. http://www.birs.ca/events/2014/5-day-workshops/14w5101/videos/watch/201401201634-Biere.html.

[7] Armin Biere. Collection of Combinational Arithmetic Miters Submitted to the SAT Competition 2016. In Tomáš Balys, Marijn Heule, and Matti Järvisalo, editors, Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions, volume B-2016-1 of Department of Computer Science Series of Publications B, pages 65–66. University of Helsinki, 2016.

[8] Armin Biere. Weaknesses of CDCL solvers. In Fields Institute Workshop on Theoretical Foundations of SAT Solving, August 2016. http://www.fields.utoronto.ca/talks/weaknesses-cdcl-solvers.

[9] B. Bollig and P. Woelfel. A read-once branching program lower bound of $\Omega(2^{n/4})$ for integer multiplication using universal hashing. In Proceedings of the Thirty-Third Annual ACM Symposium on the Theory of Computing, pages 419–424, Hersonissos, Crete, Greece, July 2001.

[10] Beate Bollig. Larger lower bounds on the OBDD complexity of integer multiplication. Inf. Comput., 209(3):333–343, 2011.

[11] Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In Proceedings of the ASPDAC 2002 / VLSI Design 2002, pages 741–746, Bangalore, India, January 2002.

[12] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, pages 174–177, 2009.
[13] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT(\{BV\}) solver for hard industrial verification problems. In Proceedings, Computer Aided Verification, 19th International Conference, CAV 2007, pages 547–560, Berlin, Germany, July 2007.

[14] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4SMT solver. In Proceedings, Computer Aided Verification, 20th International Conference, CAV 2008, pages 299–303, 2008.

[15] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers, 35(8):677–691, 1986.

[16] Randal E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. IEEE Trans. Comput., 40(2):205–213, 1991.

[17] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits, 13(4):401–424, 1994.

[18] Samuel R. Buss and Maria Luisa Bonet. An improved separation of regular resolution from pool resolution and clause learning. In Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT 2012), volume 7313 of Lecture Notes in Computer Science, pages 244–57, Trento, Italy, June 2012.

[19] Samuel R. Buss, Jan Hoffmann, and Jan Johannsen. Resolution trees with lemmas: Resolution refinements that characterize DLL algorithms with clause learning. Logical Methods in Computer Science, 4(4), 2008.

[20] Samuel R. Buss and Leszek Kołodziejczyk. Small stone in pool. Logical Methods in Computer Science, 10(2), 2014.

[21] M. Davis and H. Putnam. A computing procedure for quantification theory. Communications of the ACM, 7:201–215, 1960.

[22] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. Commun. ACM, 5(7):394–397, 1962.

[23] Leonardo Mendonça de Moura. System description: Yices 0.1. Technical report, Computer Science Laboratory, SRI International, 2005.

[24] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In Proceedings, Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, pages 337–340, 2008.

[25] Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In Eric Horvitz and Finn Verner Jensen, editors, UAI '96: Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence, pages 211–219, Portland, OR, USA, August 1996. Morgan Kaufmann.
[26] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Proceedings, Computer Aided Verification, 19th International Conference, CAV 2007, pages 519–531, Berlin, Germany, July 2007.

[27] Edward Hirsch, Dmitry Itsykson, Arist Kojevnikov, Alexander Kulikov, and Sergey Nikolenko. Report on the mixed boolean-algebraic solver. Technical report, Laboratory of Mathematical Logic of St. Petersburg Department of Steklov Institute of Mathematics, 2005.

[28] Priyank Kalla. Formal verification of arithmetic datapaths using algebraic geometry and symbolic computation. In Proceedings, Formal Methods in Computer-Aided Design, FMCAD, page 2, Austin, TX, September 2015.

[29] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of fixed-size bit-vector logics. Theory Comput. Syst., 59(2):323–376, 2016.

[30] J. Krajíček. Bounded Arithmetic, Propositional Logic and Complexity Theory. Cambridge University Press, 1996.

[31] Daniel Kroening and Ofer Strichman. Decision Procedures: An Algorithmic Point of View. Springer, 2008.

[32] João P. Marques-Silva, Ines Lynce, and Sharad Malik. CDCL solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, Handbook of Satisfiability, chapter 4, pages 131–154. IOS Press, 2009.

[33] Openssl.org. Openssl bug cve-2016-7055, 2016.

[34] Ganapathy Parthasarathy, Madhu K. Iyer, Kwang-Ting Cheng, and Li-C. Wang. An efficient finite-domain constraint solver for circuits. In Proceedings of the 41th Design Automation Conference, DAC, pages 212–217, 2004.

[35] S Ponzio. A lower bound for integer multiplication with read-once branching programs. In Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing, pages 130–139, Las Vegas, NV, May 1995.

[36] Sherief Reda and A. Salem. Combinational equivalence checking using boolean satisfiability and binary decision diagrams. In Wolfgang Nebel and Ahmed Jerraya, editors, Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2001, pages 122–126, Munich, Germany, March 2001. IEEE Computer Society.

[37] M. Sauerhoff and P. Woelfel. Time-space tradeoff lower bounds for integer multiplication and graphs of arithmetic functions. In Proceedings of the Thirty-Fifth Annual ACM Symposium on the Theory of Computing, pages 186–195, San Diega, CA, June 2003.

[38] Amr A. R. Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken, and Rolf Drechsler. Formal verification of integer multipliers by combining gröbner basis with logic reduction. In Luca Fanucci and Jürgen Teich, editors, 2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, pages 1048–1053, Dresden, Germany, March 2016. IEEE.