Integer Reasoning Modulo Different Constants in SMT

Elizaveta Pertseva[1], Alex Ozdemir[1], Shankara Pailoor[2], Alp Bassa[2], Sorawee Porncharoenwase[4*], Işil Dillig[2,3], and Clark Barrett[1]

[1] Stanford University (pertseva@stanford.edu)
[2] Veridise
[3] The University of Texas at Austin
[4] Amazon Web Services

Abstract. This paper presents a new refutation procedure for multi-modular systems of integer constraints that commonly arise when verifying cryptographic protocols. These systems, involving polynomial equalities and disequalities modulo different constants, are challenging for existing solvers due to their inability to exploit multimodular structure. To address this issue, our method partitions constraints by modulus and uses lifting and lowering techniques to share information across subsystems, supported by algebraic tools like weighted Gröbner bases. Our experiments show that the proposed method outperforms existing state-of-the-art solvers in verifying cryptographic implementations related to Montgomery arithmetic and zero-knowledge proofs.

1 Introduction

Throughout history, cryptosystems have been defined using modular arithmetic. The first symmetric cipher—the Caesar cipher—used arithmetic modulo the alphabet size. The first public key exchange (Diffie-Hellman [24]) used arithmetic modulo a large prime. The first digital signature (RSA [72]) used arithmetic modulo a large biprime. More recently, cryptosystems that *compute* on secret data, such as homomorphic encryption [37], multiparty computation [83], and zero-knowledge proofs (ZKPs) [38], often perform computation modulo a prime.

The relationship between integer arithmetic systems with *different* moduli plays a central role in *implementing* cryptography. Generally, this is because the cryptosystem is defined using a modulus that is different from the one natively supported by the computational model. For example, microprocessors efficiently perform arithmetic modulo powers of two (e.g., $2^{16}, 2^{32}, 2^{64}, \ldots$), but elliptic-curve cryptosystems require arithmetic modulo ≈256-bit primes. To bridge the gap, implementations use techniques such as Montgomery and Barrett reduction [7, 56]. Similar problems (and solutions) arise in other contexts, such as when using ZKPs. ZKPs often only support arithmetic modulo a large prime [78] and must then find ways to express and prove any properties that are not natively defined modulo that prime.

* work done while at Veridise

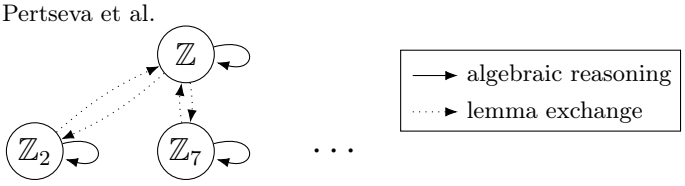

Fig. 1: Overview of our refutation procedure. Integer-reasoning interacts with modules for reasoning about equations modulo constants, e.g. (in this figure, modulo 2 and 7).

Thus, when the correctness of a cryptosystem is expressed as a logical formula, such as a Satisfiability Modulo Theories (SMT) formula, it often contains equations with different moduli. Some of the equations express the cryptographic *specification* (e.g., modulo the prime $2^{255} - 19$), and others express the *operations* performed by the implementation (e.g., modulo 2^{64}). We call such sets of constraints *multimodular systems*.

Unfortunately, such multimodular systems are hard for existing SMT solvers. One approach is to use a theory that explicitly supports the modulus operator, such as integers or bit-vectors. In practice, this leads to poor performance because the solvers for these theories are designed for *general* modular reduction (i.e., where the modulus could be a variable) and are not optimized for the special case of many constraints with constant moduli. Another approach is to combine theories optimized for different constant moduli, such as bit-vectors (modulo a power of two) and finite fields (modulo a prime). This also performs poorly because existing combination mechanisms cannot exchange sufficiently powerful lemmas between the theories.

The primary contribution of this paper is a novel refutation procedure for multimodular systems. The procedure, illustrated in Figure 1, partitions constraints into *subsystems* based on their moduli and then does local reasoning within each subsystem to detect conflicts. *Global reasoning* is done by exchanging lemmas across subsystems. This is facilitated by two key techniques: *lifting*, which promotes modular constraints to the integer domain, and *lowering*, which projects integer constraints into specific moduli. The derivation and exchange of lemmas is further supported by two complementary strategies. First, range analysis is used to verify whether a literal can be soundly lifted or lowered without altering the satisfiability of the overall system. Second, the procedure identifies additional literals *implied* by a given subsystem using algebraic techniques, such as weighted Gröbner bases and integer linear programming, to uncover constraints that enhance cross-subsystem reasoning.

We apply our procedure to unsatisfiable benchmarks based on three kinds of cryptographic implementations. The implementations include Montgomery arithmetic [56], non-native field arithmetic for a ZKP [64, 75], and multi-precision bit-vector arithmetic for a ZKP [49]. We implement our procedure in cvc5 [3], and show that it significantly outperforms prior solvers on these benchmarks. To summarize, our contributions are:

1. a refutation procedure for multimodular systems;

2. two algorithms for finding shareable lemmas, one using a weighted monomial order and the other using integer linear constraints; and
3. multimodular benchmarks from cryptographic verification applications.

The rest of the paper is organized as follows. We give a motivating example in Section 2, provide background in Section 3, introduce a logic for multimodular systems in Section 4, explain our refutation procedure in Section 5 and its implementation in Section 6, present benchmarks and experiments in Section 7, discuss related work in Section 8, and conclude in Section 9.

2 Motivating Example

Consider the following system of constraints Φ, which is based on code from a zero-knowledge proof library written by Succinct Labs [75]:

$$\begin{aligned} \Phi \triangleq \quad & xy \equiv r_1 + c_1 p \mod q \quad \wedge \quad 0 \le r_1, c_1 < p \quad \wedge \\ & r_1 y \equiv r_2 + c_2 p \mod q \quad \wedge \quad 0 \le r_2, c_2 < p \quad \wedge \\ & x + r_2 \equiv r_3 + c_3 p \mod q \quad \wedge \quad 0 \le r_3, c_3 < p \end{aligned}$$

Here, p and q are concrete primes of 32 and 256 bits respectively, but their specific values are not important. Φ is designed to ensure a correctness condition $C \triangleq (x + xy^2 \equiv r_3 \mod p)$, assuming bounds on the inputs x and y: $B \triangleq (0 \le x, y < p)$. Formally, proving Φ is correct is equivalent to proving the validity of $(\Phi \wedge B) \Rightarrow C$, or to proving the unsatisfiability of:

$$\Phi \wedge B \wedge \neg C \tag{1}$$

It is easy enough to refute Formula (1) by hand. First, observe that the range constraints in B and Φ ensure that none of the equivalences in Φ can overflow mod q—so the equivalences hold over the integers. Second, observe that said equivalences must also hold mod p:

$$xy \equiv r_1 \mod p \quad \wedge \quad r_1 y \equiv r_2 \mod p \quad \wedge \quad x + r_2 \equiv r_3 \mod p$$

Third, these equivalences imply $x + xy^2 \equiv r_3 \pmod p$, which contradicts $\neg C$.

However, existing SMT solvers fail to do this refutation. For example, when Formula (1) is encoded in `QF_NIA` using explicit modular reductions, state-of-the-art SMT solvers (including cvc5, z3, MathSAT, and Yices) fail to solve it. And when the formula is encoded in `QF_BV` using 512-bit bit-vectors, none of cvc5, z3, nor bitwuzla can solve it.[1] `QF_FF` solvers (cvc5 and Yices) do not apply because they cannot encode a variable modulo more than one prime.

The key ingredients in the manual refutation were lifting equalities from a modular space into the integers—and then lowering them back to a different modular space. In this paper, we show how to design a procedure that performs this lifting and lowering automatically.

3 Background

In this section, we define notation and provide a brief overview of algebra [54], ideals [22], and SMT [6]. More details can be found in the cited work.

[1] All tests were run with a time limit of 20 minutes and a memory limit of 8GB.

3.1 Algebra

Sets, Intervals, and Functions Let $\mathbb{Z}$ be the integers, $\mathbb{Z}_{\geq 0}$ the non-negative integers, and $\mathbb{Z}^+$ the positive integers. Let $\mathbb{Z}^+_\infty$ be $\mathbb{Z}^+ \cup \{\infty\}$, and let $\mathbb{Z}_n$ be the non-negative integers less than n. X denotes the set of variables $\{x_1, \ldots, x_k\}$. For a set S and some t, the notation S,t abbreviates $S \cup \{t\}$. $[i, j]$ denotes the closed integer interval from i to j. Interval intersection is defined in the usual way: $[a, b] \cap [c, d] = [\max(a, c), \min(b, d)]$. For an interval, pair, or sequence t, we denote by t_i the i^{th} element of t, e.g., $(a, b)_1 = a$. We use two variants of the modulo function. $a \bmod n$, for $a \in \mathbb{Z}$ and $n \in \mathbb{Z}^+$, is the unique $r \in \mathbb{Z}_n$ such that $a = qn + r$ for some $q \in \mathbb{Z}$ (as in SMT-LIB [5]). The signed variant $a \operatorname{smod} n$ is defined as $a \bmod n$ if that value is at most $\frac{n}{2}$ and $(a \bmod n) - n$ otherwise.

Polynomial Rings Let R be a ring [25]. We overload $\mathbb{Z}$ to also denote the integer ring and $\mathbb{Z}_n$ to also denote the ring over $\{0, \ldots, n-1\}$, with addition and multiplication modulo n. Both $\mathbb{Z}$ and $\mathbb{Z}_n$ are principal ideal rings (PIR), and if n is prime, $\mathbb{Z}_n$ is also a *field*. Let $R[X]$ denote the ring of polynomials with variables in X and coefficients in R. In this paper, we focus on the following rings: (1) $\mathbb{Z}[X]$, which is the ring of polynomials with integer coefficients, and (2) $\mathbb{Z}_n[X]$, which is the ring of polynomials with integer coefficients modulo n. In ring $R[X]$, a *monomial* is a polynomial of the form $x_1^{e_1} \cdots x_k^{e_k}$, with $e_i \in \mathbb{Z}_{\geq 0}$. When $e_i = 0$ for every i, we denote the monomial as 1. A *term* is a monomial multiplied by a coefficient. Polynomials are written as a sum of terms with distinct monomials.

Monomial Orders A *monomial order* $\leq$ is a total order on monomials that satisfies the following properties: (i) $1 \leq m$ for every monomial m; and (ii) for all monomials m_1, m_2, m, if $m_1 \leq m_2$, then $m_1 m \leq m_2 m$. Examples of monomial orders for a monomial of the form $x_1^{e_1} \cdots x_k^{e_k}$ include the following: *lexicographic order* compares monomials by (lexicographically) comparing their exponent tuples $(e_1, \ldots, e_k)$, *graded reverse lexicographic order* by comparing $(\sum_{i=1}^{k} e_i, -e_k, \ldots, -e_1)$, and *weighted reverse lexicographic order* by comparing $(\sum_{i=1}^{k} w_i e_i, -e_k, \ldots, -e_1)$ for a fixed tuple of weights $(w_1 \ldots w_k)$. Given a monomial order, the *leading monomial* of a polynomial p, denoted $lm(p)$, is the largest monomial occurring in p with respect to the monomial order. The leading term, denoted $lt(p)$, is that monomial's term.

3.2 Ideals

For a set of polynomials $S = \{f_1, \ldots, f_n\} \subset R[X]$, $I(S) = \{g_1 f_1 + \cdots + g_n f_n \mid g_i \in R[X]\}$ is the *ideal generated by* S. In order to disambiguate which ring R is meant in the definition of an ideal, we use the notation $I_n(S)$, with $n \in \mathbb{Z}^+_\infty$. The meaning of $I_n(S)$ is either the ideal generated by S with $g_i \in \mathbb{Z}_n[X]$, when $n \in \mathbb{Z}^+$, or the ideal generated by S with $g_i \in \mathbb{Z}[X]$, when $n = \infty$.

A *solution* to the polynomial system $S \subset R[X]$, is $\mathbf{a} \in R^k$ such that for all $f \in S$, $f(a_1, \ldots, a_k) = 0$. The set of all solutions is called the *variety* of S, denoted $\mathcal{V}(S)$. As above, we use the subscript n to distinguish among rings. Thus $\mathcal{V}_n(S)$ for $n \in \mathbb{Z}^+$ is a subset of $\mathbb{Z}_n^k$ and $\mathcal{V}_\infty(S)$ is a subset of $\mathbb{Z}^k$. If $1 \in I_n(S)$, then $\mathcal{V}_n(S) = \emptyset$, i.e., S has no solution. However, the converse does not hold.

One incomplete test for ideal membership is *reduction*. For the polynomials p, g, and $r \in R[X]$, where R is a PIR, p *reduces* to r modulo g, written $p \to_g r$, if some term t of p is divisible by $lt(g)$ with $r = p - \frac{t}{lt(g)}g$ [27]. If R is a field, then $p \to_g r$, if some term t of p is divisible by $lm(g)$. Reduction is also defined for a set of polynomials S, written as $p \to_S r$. p reduces to r modulo S if there is a sequence of reductions from p to r, each modulo some polynomial in S, and no further reduction of r modulo S is possible. If p reduces to 0 modulo S then p belongs to the ideal generated by S. However, once again, the converse does not hold.

Gröbner bases A *Gröbner basis* [13] is a set of polynomials with special properties, including that reduction is a complete test for ideal membership: p reduces to 0 modulo a Gröbner basis iff p belongs to the ideal generated by the Gröbner basis. There exist numerous algorithms for computing Gröbner bases, including Buchberger's algorithm [13], F4 [32], and F5 [33]. We use $\texttt{GB}_{n,\leq}$ with $n \in \mathbb{Z}^+_\infty$ to refer to a Gröbner basis computation. The subscript n indicates which ring the Gröbner basis is computed in ($n \in \mathbb{Z}^+$ means $\mathbb{Z}_n[X]$ and $n = \infty$ means $\mathbb{Z}[X]$), while $\leq$ indicates which monomial order to use. In this paper, we assume Gröbner bases are *strong* and *reduced*, meaning that $\texttt{GB}_{n,\leq}$ is always deterministic, producing a single unique basis.

3.3 SMT

In addition to the algebraic domains above, we also work in the logical setting of many-sorted first-order logic with equality [29]. Σ denotes a signature with a set of sort symbols (including Bool), a symbol family $\approx_\sigma$ with sort $\sigma \times \sigma \to \mathsf{Bool}$ for all sorts $\sigma \in \Sigma$,[2] and a set of interpreted function symbols. We assume the usual definitions of well-sorted Σ-terms and literals, and refer to Σ-terms of sort Bool as formulas. To distinguish logical Σ-terms from algebraic terms in polynomials (defined above), we write *Σ-term* to refer to the former, where Σ is the signature. A *theory* is a pair $\mathcal{T} = (\Sigma, \mathbf{I})$, where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations. A *logic* is a theory together with a syntactic restriction on formulas. A formula ϕ is *satisfiable* if it evaluates to `true` in some interpretation in $\mathbf{I}$. Otherwise, ϕ is *unsatisfiable*.

The CDCL($\mathcal{T}$) framework of SMT aims to determine if a formula ϕ is *satisfiable*. At a high level, a *core* module explores the propositional abstraction of ϕ and forwards literals corresponding to the current propositional assignment to the *theory solver*. A *theory solver*, specialized for a particular theory $\mathcal{T}$, checks if there exists an interpretation that satisfies the received set of literals. We focus on three main theories. The theory of finite fields (defined in [65]), which we refer to as $\mathcal{T}_{\texttt{FF}}$, reasons about finite fields, i.e., rings $\mathbb{Z}_n[X]$ where n is prime. We also make use of the standard SMT-LIB [5] theories of bit-vectors, which we denote $\mathcal{T}_{\texttt{BV}}$, and integer arithmetic, which we denote $\mathcal{T}_{\texttt{Int}}$. For two $\mathcal{T}_{\texttt{Int}}$ terms s, t, we abbreviate the literal $\neg(s \approx t)$ as $s \not\approx t$. We use $\bowtie$ to refer to an operator in the set $\{\approx, \not\approx\}$. $\mathsf{QF\_NIA}$ refers to the SMT-LIB logic that uses the theory $\mathcal{T}_{\texttt{Int}}$ and restricts formulas to be quantifier-free.

[2] We drop the σ subscript when it is clear from context.

Symbol	Arity
$n \in \mathbb{Z}$	Int
$-, +, \times$	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$
mod	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$
$\approx$	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$
$\leq, \geq$	$\mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$

(a) Signature used by QF_MIA.

$$
\begin{aligned}
\texttt{Op} &\to \times \mid \texttt{+} \mid \texttt{-} \\
\texttt{Exp} &\to (\texttt{Exp Op Exp}) \mid \texttt{Var} \mid \texttt{Int} \\
\texttt{BExp} &\to \texttt{Var} \leq \texttt{Int} \mid \texttt{Var} \geq \texttt{Int} \\
\texttt{EqExp} &\to \texttt{Exp} \approx 0 \mid \texttt{Exp mod Int} \approx 0 \\
\texttt{Atom} &\to \texttt{BExp} \mid \texttt{EqExp} \\
\texttt{Literal} &\to \texttt{Atom} \mid \neg\texttt{Atom}
\end{aligned}
$$

(b) QF_MIA grammar; $\texttt{Int} \in \mathbb{Z}$ and $\texttt{Var} \in X$.

Fig. 2: The signature and grammar for QF_MIA, a fragment of QF_NIA.

4 A Multimodular Logic

Previous work on verifying arithmetic modulo large primes [65, 66] encodes constraints using $\mathcal{T}_{\texttt{FF}}$. However, the signature of $\mathcal{T}_{\texttt{FF}}$ does not support non-prime moduli or constraints that share variables and use different moduli, limiting the range of problems that can be encoded. Instead, we encode multimodular constraints directly in $\mathcal{T}_{\texttt{Int}}$. However, we restrict the syntax of $\mathcal{T}_{\texttt{Int}}$ by defining a logic called QF_MIA (multimodular integer arithmetic), which is a fragment of QF_NIA. Importantly, QF_MIA is not *semantically* weaker than QF_NIA; it includes integer polynomials and predicates, so all of QF_NIA can be encoded in QF_MIA via standard rewrites. Rather, QF_MIA is a syntactic restriction of QF_NIA that is designed to make the multimodular structure of queries clearer so that our procedure can leverage that structure.

The subset of the signature of $\mathcal{T}_{\texttt{Int}}$ used by QF_MIA is shown in Figure 2a. From now on, we use Σ to denote this signature. A grammar for the syntactic fragment of QF_MIA is shown in Figure 2b. We assume a set $\mathcal{X} = \{\mathsf{x}_1, \ldots, \mathsf{x}_k\}$ of logical variables of sort Int and define corresponding categories of Σ-terms as follows. A QF_MIA *expression* (produced by Exp) is either an integer constant, a variable (in $\mathcal{X}$), or an application of one of the operators $-$, $+$, or $\times$ to two expressions. For simplicity, we often represent multiplication with juxtaposition (e.g., ab instead of $a \times b$) and leave out parentheses when clear from context (i.e., when they can be inferred from standard operator precedence rules). A QF_MIA *atom* (produced by Atom) is an inequality (between a variable and a constant), an equality between an expression and 0, or an equality between an expression modulo some integer constant and zero. A QF_MIA *literal* (produced by Literal) is either an atom or the negation of an atom. From now on, unless otherwise noted, expressions, atoms, and literals are QF_MIA expressions, atoms, and literals. We also assume that interpretations are $\mathcal{T}_{\texttt{Int}}$-interpretations and all notions of satisfiability are modulo $\mathcal{T}_{\texttt{Int}}$. Since we need to work in both the algebraic and the logical domains, we assume a bijection from logical variables in $\mathcal{X}$ to algebraic variables in X and define an operator $[\![\cdot]\!]$ that takes a logical Σ-term and returns the corresponding polynomial in $\mathbb{Z}[X]$ (distributing multiplication and combining like terms as necessary to obtain a sum of terms, each with a unique monomial).

5 Refutation Procedure

In this section, we describe our refutation procedure. First we describe our approach at a high level, and then we discuss its key technical ingredients.

5.1 Key Ideas

A naive approach to solving a multimodular system of constraints is to use a standard encoding in QF_NIA, introducing an auxiliary variable for each modular constraint. For instance, $x \equiv y \pmod{n}$ would be encoded as $x \approx y + n \cdot k$ using an auxiliary integer variable k. While sound, this naive approach scales poorly, as we show experimentally in Section 7. Our key insight is that this limitation can often be overcome by *partitioning* the original system into a *set* of different subsystems, one for each specific modulus. Reasoning in each subsystem can be done efficiently, and if any subsystem is unsatisfiable, then so is the original set of constraints. However, since the converse is not true, our procedure seeks to exchange as much information as possible between the different subsystems, with the goal of improving our ability to detect unsatisfiable constraints. To enable the exchange of information between the different subsystems, we employ the concepts of *lifting* and *lowering*.

Definition 1 (Liftable). *Let C be a set of* QF_MIA *literals. A literal of the form $e \bmod n \bowtie 0$ is liftable (in C) if $C \cup \{e \bmod n \bowtie 0\}$ is equisatisfiable to $C \cup \{e \bmod n \bowtie 0, e \bowtie 0\}$.*

In other words, a constraint is liftable if adding the constraint without the modulus n maintains equisatisfiability.

Definition 2 (Lowerable). *Let C be a set of* QF_MIA *literals. A literal the form $e \bowtie 0$ is* lowerable *(in C) with respect to n if $C \cup \{e \bowtie 0\}$ and $C \cup \{e \bowtie 0, e \bmod n \bowtie 0\}$ are equisatisfiable.*

In other words, a constraint is *lowerable* w.r.t. n if adding it with a modular reduction maintains equisatisfiability.

Remark 1. If the literal $e \bmod n \bowtie 0$ is implied by C, then the lifting definition reduces to: C and $C \cup \{e \bowtie 0\}$ must be equisatisfiable. Similarly, if the literal $e \bowtie 0$ is implied by C, it is lowerable w.r.t. n if C and $C \cup \{e \bmod n \bowtie 0\}$ are equisatisfiable. In the remainder of the paper, we rely on these simpler versions of the definitions.

Lifting provides a way for each subsystem containing modular constraints to share information in a common language without adding new variables. Lowering adds constraints to individual subsystems and can often result in significant simplifications: when we lower an equation with respect to n, all integer constants divisible by n can be replaced by 0. For example, lowering $\mathsf{x}_1 - 6\mathsf{x}_2 \approx 0$ with respect to 6 adds a new equality $\mathsf{x}_1 \approx 0$ to the subsystem with modulus 6. We denote by $simp_n(e)$ the result of replacing every integer constant c in e by $c \operatorname{smod} n$ [3] and simplifying.

[3] We use smod instead of mod to reduce the magnitude of coefficients of $simp_n(e)$ and increase the likelihood that $simp_n(e)$ is liftable according to Lemma 1.

$$\texttt{ModExp}\ \frac{\bowtie \in \{=, \neq\} \qquad (e \bmod n \bowtie 0) \in C}{C := C \setminus \{e \bmod n \bowtie 0\} \quad R_{\mathrm{n}}^{\bowtie} := R_{\mathrm{n}}^{\bowtie}, simp_n(e)}$$

$$\texttt{IntExp}\ \frac{\bowtie \in \{=, \neq\} \qquad (e \bowtie 0) \in C}{C := C \setminus \{e \bowtie 0\} \quad R_{\infty}^{\bowtie} := R_{\infty}^{\bowtie}, e}$$

$$\texttt{GEQ}\ \frac{(\mathsf{x} \geq n) \in C}{C := C \setminus \{\mathsf{x} \geq n\} \quad B(\mathsf{x})_1 := \max(B(\mathsf{x})_1, n)}$$

$$\texttt{LEQ}\ \frac{(\mathsf{x} \leq n) \in C}{C := C \setminus \{\mathsf{x} \leq c\} \quad B(\mathsf{x})_2 := \min(B(\mathsf{x})_2, n)}$$

$$\texttt{nGEQ}\ \frac{\neg(x \geq n) \in L}{C := C \setminus \{\neg(\mathsf{x} \geq n)\} \quad B(\mathsf{x})_2 := \min(B(\mathsf{x})_2, n-1)}$$

$$\texttt{nLEQ}\ \frac{\neg(\mathsf{x} \leq n) \in L}{C := C \setminus \{\neg(\mathsf{x} \leq n) \quad B(\mathsf{x})_1 := \max(B(\mathsf{x})_1, n+1)}$$

Fig. 3: Encoding rules for a multimodular system C, where e is an expression and $n \in \mathbb{Z}_{\infty}^{+}$.

As expected, not all constraints are liftable or lowerable. In order to facilitate the inference of liftable and lowerable constraints, our method splits the constraint system into the following subsystems, for $n \in \mathbb{Z}_{\infty}^{+}$:

- **Modulus-n equality subsystems** are sets of expressions $\{e_1, \ldots, e_m\}$. Each e_i represents the constraint $e_i \bmod n \approx 0$ when $n \neq \infty$ or the constraint $e_i \approx 0$ when $n = \infty$. We write $R_n^{\approx}$ to denote the set of expressions.
- **Modulus-n disequality subsystems** are sets of expressions $\{e_1, \ldots, e_m\}$. Each e_i represents the constraint $e_i \bmod n \not\approx 0$ when $n \neq \infty$ or the constraint $e_i \not\approx 0$ when $n = \infty$. We write $R_n^{\not\approx}$ to denote the set of expressions.
- **Variable bounds:** Our method also maintains a mapping B from each variable in $\mathcal{X}$ to its lower and upper bound (with $-\infty/\infty$ denoting unbounded variables). As we will see shortly, this bound information is crucial for identifying lowerable and liftable equations.

Given a multimodular system C, we assume that it is encoded as a tuple $(B, R_{\infty}^{\approx}, R_{\infty}^{\not\approx}, R_{n_1}^{\approx}, R_{n_1}^{\not\approx}, \ldots, R_{n_k}^{\approx}, R_{n_k}^{\not\approx})$, where $\{n_1, \ldots, n_k\}$ is the set of all integer constants greater than 1 appearing anywhere in a literal in C. For example, for a multimodular system $\{2x \bmod 3 \approx 0, x < 5\}$, the resulting tuple would be $(B, R_{\infty}^{\approx}, R_{\infty}^{\not\approx}, R_2^{\approx}, R_2^{\not\approx}, R_3^{\approx}, R_3^{\not\approx}, R_5^{\approx}, R_5^{\not\approx})$. A set of rules for accomplishing the encoding is included in Figure 3.

Going forward, we use C to refer both to the original set of constraints and to the tuple encoding. We also define $\texttt{CalcBds}(B, e)$ as a function that returns the maximum and minimum possible values that expression e can take when evaluated at the variable assignments permitted by the variable bounds map B, based

on standard interval arithmetic [45]. For example, given an equality $\mathtt{x}_1\mathtt{x}_2 \bmod 6 \approx 0$ and variable bounds $B = \{\mathtt{x}_1 : [0,6], \mathtt{x}_2 : [0,6]\}$, $\mathtt{CalcBds}(B, \mathtt{x}_1\mathtt{x}_2)$ would return the interval $[0, 36]$. Using this machinery, we now state the following lemmas that help identify liftable and lowerable constraints. We include the proofs in the extended version of the paper [69]. Recall that I_n computes the ideal generated by a set of polynomials.

Lemma 1. *Let C be a multimodular system with bounds B and modulus-n equality subsystem $R_n^{\approx}$, with $n \in \mathbb{Z}^+$. Then, an equality $e \bmod n \approx 0$ is liftable in C if (1) $[\![e]\!] \in I_n([\![R_n^{\approx}]\!])$ and (2) $\mathtt{CalcBds}(B, e) \subseteq [1-n, n-1]$.*

This lemma is useful in two ways. First, if e is in $R_n^{\approx}$, then certainly, $[\![e]\!]$ is in $I_n([\![R_n^{\approx}]\!])$; thus, checking whether the constraint represented by an element of $R_n^{\approx}$ is liftable reduces to computing $\mathtt{CalcBds}(B, e)$, which can be done in linear time. Second, this lemma gives a way to find *additional* liftable equalities that are not part of the original constraint system by identifying polynomials that are in the ideal of $[\![R_\infty^{\approx}]\!]$. While listing all the polynomials in an ideal is infeasible, later subsections (Sections 5.4 and 5.5) explore effective methods to identify useful polynomials that are in the ideal. The next lemma states that disequalities are always liftable.

Lemma 2. *Let C be a multimodular system with modulus-n disequality subsystem $R_n^{\not\approx}$, with $n \in \mathbb{Z}^+$. Then a disequality $e \bmod n \not\approx 0$ is liftable in C w.r.t. n if $e \in R_n^{\not\approx}$.*

This lemma states that all of the original disequalities are liftable. However, unlike the equality case, we cannot infer *additional* disequalities using ideals, as disequalities are not preserved under the operations used to construct the elements of an ideal. The next two lemmas are dual to Lemmas 1 and 2, but are for lowerability instead of liftability.

Lemma 3. *Let C be a multimodular system with integer equalities $R_\infty^{\approx}$. If e is an expression and $[\![e]\!] \in I_\infty([\![R_\infty^{\approx}]\!])$, then $e \approx 0$ is lowerable w.r.t. every $n \in \mathbb{Z}^+$.*

Lemma 4. *Let C be a multimodular system with integer disequalities $R_\infty^{\not\approx}$ and bounds B. Then, if $n \in \mathbb{Z}^+$, a disequality $e \not\approx 0$ is lowerable in C w.r.t. to n if (1) $e \in R_\infty^{\not\approx}$ and (2) $\mathtt{CalcBds}(B, e) \subseteq [1-n, n-1]$*

5.2 Refutation Calculus

Next, we leverage the notions of liftability and lowerability defined in the previous subsection to formulate our *refutation calculus* (presented in Fig. 4). The rules in our calculus serve four main roles. First, they establish whether a specific subsystem is unsatisfiable. Second, they attempt to tighten existing bounds and learn new equalities from these bounds. Third, they leverage Lemmas 1–4 to exchange information between different subsystems via lifting and lowering. Finally, they simplify unliftable equalities via branching.

$$\texttt{UnsatOne}\ \frac{1 \in I_n([\![R_n^{\approx}]\!])}{\texttt{unsat}} \qquad \texttt{UnsatDiseq}\ \frac{e \in R_n^{\not\approx} \quad [\![e]\!] \in I_n([\![R_n^{\approx}]\!])}{\texttt{unsat}}$$

$$\texttt{UnsatBds}\ \frac{B(\mathsf{x}_i)_1 > B(\mathsf{x}_i)_2}{\texttt{unsat}}$$

$$\texttt{ConstrBds}\ \frac{[\![e]\!] \in I_\infty([\![R_\infty^{\approx}]\!]) \quad e = a\mathsf{x}_i + e' \quad x_i \notin e' \quad B(\mathsf{x}_i)_1 \leq B(\mathsf{x}_i)_w}{B(\mathsf{x}_i) := \texttt{CalcBds}(B, -\tfrac{1}{a}(e - a\mathsf{x}_i)) \cap B(\mathsf{x}_i)}$$

$$\texttt{InfEq}\ \frac{B(\mathsf{x}_i)_1 = B(\mathsf{x}_i)_2 \qquad [\![\mathsf{x}_i - B(\mathsf{x}_i)_1]\!] \notin I_\infty([\![R_\infty^{\approx}]\!])}{R_\infty^{\approx} := R_\infty^{\approx}, \mathsf{x}_i - B(\mathsf{x}_i)_1}$$

$$\texttt{LiftEq}\ \frac{n \neq \infty \quad [\![e]\!] \in I_n([\![R_n^{\approx}]\!]) \quad \texttt{CalcBds}(B, e) \subseteq [1-n, n-1] \quad [\![e]\!] \notin I_\infty([\![R_\infty^{\approx}]\!])}{R_\infty^{\approx} := R_\infty^{\approx}, e}$$

$$\texttt{LiftDiseq}\ \frac{n \neq \infty \quad e \in R_n^{\not\approx} \quad e \notin R_\infty^{\not\approx}}{R_\infty^{\not\approx} := R_\infty^{\not\approx}, e}$$

$$\texttt{LowerEq}\ \frac{[\![e]\!] \in I_\infty([\![R_\infty^{\approx}]\!]) \quad n \neq \infty \quad [\![simp_n(e)]\!] \notin I_n([\![R_n^{\approx}]\!]) \quad R_n^{\approx} \in C}{R_n^{\approx} := R_n^{\approx}, simp_n(e)}$$

$$\texttt{LowerDiseq}\ \frac{\begin{array}{c} e \in R_\infty^{\not\approx} \\ n \neq \infty \quad \texttt{CalcBds}(B, e) \subseteq [1-n, n-1] \quad simp_n(e) \notin R_n^{\not\approx} \quad R_n^{\not\approx} \in C \end{array}}{R_n^{\not\approx} := R_n^{\not\approx}, simp_n(e)}$$

$$\texttt{RngLift}\ \frac{\begin{array}{c} [\![e]\!] \in I_n([\![R_n^{\approx}]\!]) \quad n \neq \infty \\ \texttt{CalcBds}(B, e) \not\subseteq [1-n, n-1] \quad \texttt{CalcBds}(B, e) \subseteq [1-2n, 2n-1] \\ [\![e-n]\!] \notin I_\infty([\![R_\infty^{\approx}]\!]) \quad [\![e]\!] \notin I_\infty([\![R_\infty^{\approx}]\!]) \quad [\![e+n]\!] \notin I_\infty([\![R_\infty^{\approx}]\!]) \end{array}}{R_\infty^{\approx} := R_\infty^{\approx}, e-n \;||\; R_\infty^{\approx} := R_\infty^{\approx}, e \;||\; R_\infty^{\approx} := R_\infty^{\approx}, e+n}$$

$$\texttt{ZeroOrOne}\ \frac{\begin{array}{c} [\![e]\!] \in I_n([\![R_n^{\approx}]\!]) \quad [\![e]\!] = s^2 - s \\ n \neq \infty \quad \texttt{IsPrime}(n) \quad [\![s]\!] \notin I_n([\![R_n^{\approx}]\!]) \quad [\![s-1]\!] \notin I_n([\![R_n^{\approx}]\!]) \end{array}}{R_n^{\approx} := R_n^{\approx}, s \;||\; R_n^{\approx} := R_n^{\approx}, s-1}$$

Fig. 4: Derivation rules. e, s are expressions, $a \in \mathbb{Z}$, and $n \in \mathbb{Z}_\infty^+$.

We present the calculus as rules that modify *configurations*, as is common in SMT procedures [50, 74]. Here, a configuration is the representation of the system of constraints C as the tuple $(B, R_\infty^{\approx}, R_\infty^{\not\approx}, R_{n_1}^{\approx}, R_{n_1}^{\not\approx}, \ldots, R_{n_k}^{\approx}, R_{n_k}^{\not\approx})$ as described in Section 5.1. The rules are presented in *guarded assignment form*, where the premises describe the conditions on the current configuration under which the rule can be applied, and the conclusion is either unsat or indicates how the configuration is modified. A rule may have multiple alternative conclusions separated by $||$. An application of a rule is *redundant* if it does not change the configuration in any way. A configuration other than unsat is *saturated* if every possible application is redundant. A *derivation tree* is a tree where each

node is a configuration and its children, if any, are obtained by a non-redundant application of a rule of the calculus. A derivation tree is closed if all of its leaves are `unsat`. A *derivation* is a sequence of derivation trees in which each element in the sequence (after the first) is obtained by expanding a single leaf node in the previous tree. We explain each class of rules in the calculus in more detail below.

Checking Unsatisfiability `UnsatOne` is used to conclude unsatisfiability using algebraic techniques: As explained in Section 3, if some ideal $[\![R_n^{\approx}]\!]$ contains the polynomial 1, this indicates that the constraints represented by $R_n^{\approx}$ have no common solution in the modulus-n subsystem. Hence, we can conclude that the whole system is unsatisfiable. `UnsatDiseq` checks if any of the polynomials in the ideal of $[\![R_n^{\approx}]\!]$ match an expression in $R_n^{\not\approx}$. Finally, `UnsatBds` concludes `unsat` if some variable's lower bound exceeds its upper bound.

Tightening Bounds `ConstrBds` tightens bounds based on equations in $R_\infty^{\approx}$. Suppose e is an expression, with $[\![e]\!] \in I_\infty([\![R_\infty^{\approx}]\!])$, consisting of the sum of $a\mathtt{x}_i$ and some other expression e'. `ConstrBds` uses e to compute new bounds for $\mathtt{x}_i$ and then intersects these with the current bounds for $\mathtt{x}_i$. `InfEq` uses the bounds information to infer new equalities: When a variable's upper and lower bounds become equal, we can obtain a new equality.

Lifting/Lowering The lifting and lowering rules are directly based on the lemmas stated in Section 5.1. The rules `LiftEq` and `LiftDiseq` *lift* (dis)equalities from $R_{\mathrm{n}}^{\bowtie}$ to $R_\infty^{\bowtie}$, by relying on Lemmas 1 and 2. We also check entailment (polynomial ideal containment for equalities and set containment for disequalities) to avoid adding redundant information. Dually, the rules `LowerEq` and `LowerDiseq` *lower* (dis)equalities from from $R_\infty^{\bowtie}$ to $R_{\mathrm{n}}^{\bowtie}$ by relying on Lemmas 3 and 4.

Branching Rules When encountering equations that are *almost* liftable, we rely on branching rules to either lift or simplify them. For example, if $\texttt{CalcBds}(B, x) = [0, 7]$, $x \bmod 6 \approx 0$ is *almost* liftable, as its range exceeds the liftable range only slightly. `RngLift` branches on possible values for e that are within $|n|$ of the liftable range. `ZeroOrOne` detects when the value of a variable can only be 0 or 1; it applies only to prime moduli because other moduli have zero divisors.

We state *soundness* and *termination* theorems for our calculus and include the proofs in the extended version of the paper [69]. However, the calculus is *not* complete—a saturated leaf in a derivation tree does not necessarily mean that the constraints at the root of the tree are satisfiable.

Theorem 1. *Soundness: If T is a closed derivation tree with root node C, then C is unsatisfiable in $\mathcal{T}_{\mathtt{Int}}$.*

Theorem 2. *Termination: Every derivation starting from a finite configuration C where every variable is bounded is finite.*

5.3 Refutation Algorithm

Our implementation of the calculus follows the following strategy. It first performs lifting via the `LiftEq` and `LiftDiseq` rules to exhaustion (i.e., until these

In: A set of constraints C
Out: unsat/unknown
1 **def** REFUTE(C)**:**
2 $(B, R_\infty^{\approx}, R_\infty^{\not\approx} \ldots R_n^{\approx}, R_n^{\not\approx}) \leftarrow$ `Split`(C) `// Figure 3`
3 **while** `Exchange`$((B, R_\infty^{\approx}, R_\infty^{\not\approx} \ldots R_n^{\approx}, R_n^{\not\approx}))$ `// Section 5.2-5.5`
4 **do**
5 **if** `CheckUnsat`$((B, R_\infty^{\approx}, R_\infty^{\not\approx} \ldots R_n^{\approx}, R_n^{\not\approx}))$ `// Section 5.2`
6 **then**
7 **return** unsat
8 **if** $S \leftarrow$ `Branch`$((B, R_\infty^{\approx}, R_\infty^{\not\approx} \ldots R_n^{\approx}, R_n^{\not\approx}))$ **then**
9 **for** $c_i \in S$ **do**
10 **if** REFUTE(c_i) == unknown **then**
11 **return** unknown
12 **return** unsat
13 **return** unknown

Algorithm 1: Overview of the Refutation Procedure

rules no longer apply) and then attempts to refute each subsystem using the `UnsatOne` and `UnsatDiseq` rules for each $R_n^{\approx}$, $n \neq \infty$. If this fails, it applies the bounds-related rules (`ConstrBds`, `InfEq`, `UnsatBds`) and lowering rules (`LowerEq` and `LowerDiseq`), also to exhaustion. It then once again attempts refutation, but with $n = \infty$. If that also fails, it applies a branching rule, if applicable (trying first `ZeroOrOne`, then `RngLift`), and then the entire strategy repeats.

Algorithm 1 presents an algorithmic view of the overall process. Given constraints C, the procedure starts by encoding C into a configuration using the `Split` routine, which creates the tuple encoding described in Section 5.1 and applies the rules from Figure 3. The main refutation procedure consists of the while loop in lines 3–7 and works as follows. First, it exchanges expressions between the different subsystems via lifting and lowering and tightens bounds or learns new equations from bounds (line 3). Since lifting requires finding additional polynomials that are in the ideal, the implementation of the `Exchange` procedure utilizes the methods described in Sections 5.4 and 5.5 to find additional polynomials corresponding to exchangeable expressions. The methods can be used individually or in combination, and we defer the discussion of their ordering to Section 7. Next, at line 5, the algorithm invokes `CheckUnsat` to attempt to refute any of the subsystems. If `CheckUnsat` returns true, the algorithm terminates with unsat. Otherwise, the algorithm attempts to apply a branching rule (lines 8-12). If successful, it recursively calls the REFUTE procedure on a new set of constraints and returns unsat if *all* recursive calls perform successful refutation. Since our algorithm is intended only for refutation rather than model construction, it returns unknown in all other cases.

5.4 Finding Liftable Equalities via Weighted Gröbner bases

Recall that several rules in our refutation calculus require finding liftable or lowerable constraints that correspond to a polynomial in the ideal of some exist-

ing set of polynomials. As explained earlier, finding *additional* liftable/lowerable constraints is useful because they can make it easier to prove unsatisfiability. However, since ideals are infinite, we need some algorithm for selecting which polynomials to target. In this section, we present a method for computing "useful" polynomials—those that are likely to correspond to liftable constraints. We focus on the liftability of equalities only, because all equalities are already lowerable, and disequalities do not form an ideal and thus are easy to enumerate.

Based on Lemma 1, a polynomial's bound endpoints are the most important indicators of liftability. As a result, we target polynomials that we call *near-zero polynomials*, or polynomials with a bound whose endpoints have small absolute value. Based on this definition, a polynomial with a bound $[-5, 5]$ is *closer to zero* than a polynomial with a bound of $[-50, 50]$ or $[49, 50]$.

Since we cannot enumerate the ideal, one possibility is to construct a basis for the ideal that contains polynomials that are more likely to refer to liftable constraints. As discussed in Section 3, one possible basis is a Gröbner basis, which uses a monomial order. A Gröbner basis is computed via an algorithm that often eliminates larger monomials with respect to the order. Our idea is to compute a Gröbner basis using a carefully chosen monomial order in which monomials that are closer to zero are smaller than those farther from zero. The insight is that polynomials with near-zero monomials are generally (though not always) more likely to be liftable. To achieve this, our monomial order must encode information about bounds on the monomials. To this end, we first take the maximum of the absolute values of the lower and upper bounds on each variable. Additionally, since weighted monomial orders are determined by the dot product of variable exponents and assigned weights, we apply logarithmic scaling to ensure that the influence of a bound is appropriately adjusted by each variable's degree. Based on this intuition, we use a *weighted reverse lexicographic* monomial order (Sec. 3), with weights:

$$\texttt{weights} \leftarrow \left[\log\left(\max\left(|B(\mathsf{x}_i)_1|, |B(\mathsf{x}_i)_2|\right) + \epsilon^4\right) \mid \mathsf{x}_i \in \mathcal{X}\right] \tag{2}$$

When applying the `LiftEq` rule in the refutation calculus our algorithm computes a Gröbner basis according to the order defined by (2) and only checks the liftability of the polynomials in this Gröbner basis.

Ideally, when we apply the `LiftEq` rule, we want to find a *complete* set of liftable equations—that is, any other liftable constraint should be implied by the ones inferred by our technique. Our weighted Gröbner basis method does not have this completeness guarantee in general, but the following theorem states a weaker completeness guarantee that it *does* provide:

Theorem 3. *Let C be a constraint system containing a modulus-n equality subsystem $R_n^{\approx}$ and variable bounds B s.t. for all $\mathsf{x}_i \in \mathcal{X}$, $0 \in B(\mathsf{x}_i)$. Let $G = GB_{n,\leq}([\![R_n^{\approx}]\!])$ be a Gröbner basis computed with a weighted reverse lexicographical order with weights from Equation 2. Finding liftable equalities derived from*

[3] The ϵ serves to avoid $\log(0)$ which is undefined.

$I(R_n^{\approx})$ by computing G is complete if every generator $[\![e]\!] \in G$ that has a leading monomial $[\![m]\!]$ s.t. $\texttt{CalcBds}(B, m) \subseteq [1-n, n-1]$ corresponds to a liftable expression e mod n in C.

Completeness implies that the ideal generated by the generators of G, which correspond to liftable expressions, contains all polynomials corresponding to liftable expressions from $I_n([\![R_n^{\approx}]\!])$.

Intuitively, completeness means $\{[\![p]\!] : \texttt{CalcBds}(B, p) \subseteq [1-n, n-1] \wedge [\![p]\!] \in I_n([\![R_n^{\approx}]\!])\} \subseteq I_n(\{[\![g]\!] : \texttt{CalcBds}(B, g) \subseteq [1-n, n-1] \wedge [\![g]\!] \in G\})$. A detailed proof can be found in the extended version of the paper [69]. This theorem is useful because it allows us to identify cases when additional heuristics for identifying liftable polynomials might be useful.

5.5 Finding Liftable Equalities via Integer Linear Constraints

Now we present another method, complementary to the one in Section 5.4, for finding liftable constraints. This new method is motivated by the following observation: In many cases, liftable equalities can be obtained from *linear combinations* of existing expressions—i.e., they are of the form $e' = \sum_{i=1}^{l} a_i e_i$ for existing expressions e_i and constant coefficients a_i. The method proposed here aims to find these unknown coefficients $a_1, \ldots, a_n$ by setting up an auxiliary *integer linear constraint*. If this linear constraint is feasible, then the inferred equality is guaranteed to satisfy the conditions from Lemma 1.

To emphasize that the coefficients a_i are the unknowns being solved for, we depict them using a bold font below. Our goal is to encode the condition $\texttt{CalcBds}(B, e') \subset [1-n, n-1]$ in the generated constraint. Since $\texttt{CalcBds}$ depends on e''s coefficients, we express those coefficients in terms of the $\mathbf{a}_i$. Let $[\![m_1]\!], \ldots, [\![m_t]\!]$ be all the monomials present in $[\![R_n^{\approx}]\!]$, including the constant monomial $1 \in \mathbb{Z}[X]$. Moreover, let $[\![e_i]\!]$ have coefficients $c_{i,j}$, such that $[\![e_i]\!] = \sum_{j=1}^{t} c_{i,j}[\![m_j]\!]$. Then, the coefficient of $[\![m_j]\!]$ in $[\![e']\!]$, denoted $\texttt{coef}_j$, is the linear polynomial, $\texttt{coef}_j \triangleq \sum_{i=1}^{t} \mathbf{a}_i c_{i,j}$.

We can now express the relevant bounds. The lower and upper bounds on each *monomial* m_j can be concretely computed as $u_j \triangleq \texttt{CalcBds}(B, m_j)_1$ and $l_j \triangleq \texttt{CalcBds}(B, m_j)_0$. Then, the lower and upper bounds on e' itself are respectively:

$$\texttt{lb} \triangleq \sum_{j=1}^{t} (l_j \cdot \texttt{ReLU}(\texttt{coef}_j) - u_j \cdot \texttt{ReLU}(-\texttt{coef}_j))$$
$$\texttt{ub} \triangleq \sum_{j=1}^{t} (u_j \cdot \texttt{ReLU}(\texttt{coef}_j) - l_j \cdot \texttt{ReLU}(-\texttt{coef}_j)),$$

where $\texttt{ReLU}(a)$ is $\max(a, 0)$. $\texttt{ReLU}$ constraints capture how negating a monomial affects its upper and lower bounds: if $\texttt{coef}_j > 0$, u_j contributes to the upper bound of e'; otherwise, it contributes to the lower bound. These constraints can be encoded using binary variables when the upper and lower bounds are known [76]. Since the operations are performed modulo n, all coefficients must lie within the interval $[1-n, n-1]$. Our encoding Φ is then:

$$\Phi \triangleq \quad \texttt{lb} > -n \quad \wedge \quad \texttt{ub} < n \quad \wedge \quad \sum_{i=1}^{k} (\texttt{ReLU}(\texttt{coef}_j) + \texttt{ReLU}(-\texttt{coef}_j)) > 0$$

The first two conditions ensure e' is liftable, and the last ensures that $e' \neq 0$.

Theorem 4. *Let C be a constraint system containing modulus-n equality subsystem $R_n^{\approx}$ and variable bounds B. Φ is satisfiable iff there exists a linear combination of the form $e' = \sum_{i=1}^{l} a_i e_i$, $a_i \in \mathbb{Z}$ and $e_i \in R_n^{\approx}$, s.t. e' is liftable in C and $e' \neq 0$.*

In the case that Φ is satisfiable, we seek to find all linearly independent solutions, as different solutions correspond to different liftable equations, all of which may be useful for proving unsatisfiability. Thus, we add constraints ruling out linear combinations of solutions found so far and iterate until ϕ is unsatisfiable.

6 Implementation

We implemented our refutation procedure in the cvc5 SMT solver [3] as an alternative to the existing nonlinear integer solver. We use Singular v4.4.0 [23] for the algebraic components of our procedure and glpk [52] v4.6.0 for solving the integer linear constraints problems from Section 5.5. Below we describe the key details in our implementation.

Rewrites Our implementation disables all rewrites for the mod operator but retains the remaining cvc5 rewrites and pre-processing passes.

Ideal Membership Check To check if a polynomial version of an expression is in $I_n(\llbracket R_n^{\approx} \rrbracket)$ we compute a Gröbner basis (GB) of $\llbracket R_n^{\approx} \rrbracket$ using *graded reverse lexicographic order* unless $\llbracket R_n^{\approx} \rrbracket$ is already a GB. (Recall that reduction by a GB is a complete ideal membership test.) To prevent GB computation from becoming a bottleneck, we impose a 30-second timeout. If a timeout occurs, we only check inclusion in the set instead of the ideal.

Liftable Equalities via Integer Linear Constraints Experimental results showed that integer linear constraint solvers perform poorly on problems with large integer constants. To address this issue, our implementation uses an approximate version where we scale constants using signed log defined as $\mathrm{slog}_2(a) = \frac{|a|}{a} \log_2(a)$ and use a 30-second timeout. Details of our approximation, including empirical results that motivate this approximation can be found in the extended version of the paper [69]. All experimental results reported in the paper use this relaxation of the encoding.

7 Experiments

Our experiments are designed to answer two key empirical questions 1. How does our refutation procedure compare to the state of the art? 2. Do our lifting algorithms (Sec. 5.4 and 5.5) improve performance? All of our expriments are run on a cluster with Intel Xeon E5-2637 v4 CPUs. Each run is limited to one physical core, 8GB memory, and 20 minutes.

7.1 Benchmarks

The benchmarks used in our experimental evaluation consist of logical formulas encoding the correctness of simulating arithmetic operations in a given *specification* domain using arithmetic in a corresponding *base* domain. Each domain is either a finite field ($\mathbb{F}_p$ or $\mathbb{F}_q$) or a bit-vector domain ($\mathbb{Z}_{2^b}$). Implementations

Family	Spec.	Base	Reference
`f/b(s)`	$\mathbb{F}_p$	$\mathbb{Z}_{2^b}$	Montgomery arithmetic [56]
`f/b(m)`	$\mathbb{F}_p$	$\mathbb{Z}_{2^b}$	Montgomery arithmetic [56]
`f/f(s)`	$\mathbb{F}_p$	$\mathbb{F}_q$	Succinct labs [43, 75]
`f/f(m)`	$\mathbb{F}_p$	$\mathbb{F}_q$	o1-labs [64]
`b/f(m)`	$\mathbb{Z}_{2^b}$	$\mathbb{F}_p$	xjSnark [49]

Table 1: Our benchmarks, which verify arithmetic implementations with various specification domains, base domains and numeric precisions.

that represent values in the specification domain as multiple *limbs* in the base domain are referred to as *multiprecision* (`m`), while those using a single representation are classified as *single precision* (`s`). As summarized in Table 1, our benchmarks fall under the following categories:

- `f/b(s)` encode the correctness of single-precision *Montgomery arithmetic* [56], which implements arithmetic modulo p using bit-vector operations, without mod-p reductions, and is common in prime field CPU implementations. The benchmarks verify the correctness of the `REDC` subroutine and the end-to-end correctness of Montgomery's approach for evaluating expressions mod p.
- `f/b(m)` encode correctness of multi-precision Montgomery arithmetic.
- `f/f(s)` encode implementations of arithmetic modulo the 31-bit Goldilocks prime [43] modulo a 255-bit prime (the order of the BLS 12-381 elliptic curve [4, 10]). The benchmarks model a Succinct labs implementation [75] that is used for recursive zero-knowledge proofs (ZKPs).
- `f/f(m)` encode the correctness of implementations of arithmetic modulo one 255-bit prime using arithmetic modulo another 255-bit prime. Multiple limbs are used to avoid unintended overflow in the base domain. The benchmarks model an o1-labs implementation [64] used for recursive ZKPs.
- `b/f(m)` encode correctness of multi-precision implementations of bit-vector arithmetic modulo a 255-bit prime. The benchmarks model techniques from the xjSnark ZKP compiler [49] that are used to check RSA signatures.

Determinism We include *determinism* [66] benchmarks in addition to correctness for families with a finite field base domain. Determinism is a weaker property than correctness but rules out most bugs in practice [15]. We do not include determinism `f/f(m)` benchmarks because they are correct but **non**deterministic. Additional benchmark statistics are included in the extended version of the paper [69].

7.2 State of the Art Comparison

Baselines Our benchmarks are SMT problems in QF_MIA (Sec. 4). Since all variables have finite bounds, our benchmarks can also be encoded in QF_BV. Our determinism benchmarks contain only variable bounds and equations modulo one prime and can thus be encoded in QF_FF using standard $\mathbb{F}$ encodings of range constraints [49]. As a result, the baselines for our work are existing solvers for QF_NIA, QF_BV, and QF_FF. We compare against Yices v2.6.5 [26], cvc5

	Family							
	f/b(s)	f/b(m)	f/f(s)		f/f(m)	b/f(m)		
Solver/Logic	cor	cor	cor	det	cor	cor	det	Total
ours QF_MIA	**53**	14	**10**	**84**	**50**	**24**	20	**255**
z3 QF_NIA	49	21	3	34	40	22	21	190
cvc5 QF_NIA	40	35	0	19	**50**	14	8	166
yices QF_NIA	32	24	0	2	10	13	8	89
bitwuzla(abst.) QF_BV	49	**49**	0	2	25	9	8	142
bitwuzla QF_BV	49	**49**	0	0	0	9	8	115
cvc5 QF_BV	43	48	0	0	0	9	7	107
z3 QF_BV	48	37	0	0	0	9	8	102
cvc5 QF_FF	–	–	–	1	–	–	**23**	24
yices QF_FF	–	–	–	0	–	–	20	20
cvc5 (split GB) QF_FF	–	–	–	2	–	–	9	11
# Benchmarks	72	58	10	98	50	24	24	336

Table 2: Solved benchmarks. Here, cor stands for correctness and det for determinism. – indicates the benchmark family cannot be encoded into the logic.

	Family								
	f/b(s)	f/b(m)	f/f(s)		f/f(m)	b/f(m)			
Ablation	cor	cor	cor	det	cor	cor	det	Total	Uniq.
Weighted GB	53	14	**10**	**84**	**50**	**24**	20	**255**	11
Unweighted GB	41	**25**	**10**	61	**50**	17	20	223	1
Lin. Constraints	37	18	0	7	0	7	3	72	3
Weighted GB & Lin.	**54**	23	**10**	73	**50**	**24**	20	254	**12**
# Benchmarks	72	58	10	98	50	24	24	336	

Table 3: Results with different lifting algorithms

v1.2.2 [3], bitwuzla v0.5.0 [60], and z3 v4.13.1 [57], which are state of the art for these logics. We evaluate bitwuzla with and without abstraction [62] and cvc5's QF_FF solver with and without split Gröbner bases [66].

Comparison Table 2 shows the number of unsat benchmarks solved by family, type, logic, and tool, and Figure 5 shows the performance for the top 5 solvers. Here, 'ours' stands for the best version of our solver: one with a lifting algorithm based on a weighted Gröbner basis (Section 5.4). This configuration outperforms existing tools on 5 out of the 7 categories, as well as in total benchmarks solved. It is also the most efficient and has the most unique solves, with 76 benchmarks not solved by any other solver. Of the 81 benchmarks our tool fails to solve, 27 remain unsolved by any solver. For these 81 benchmarks, 10 run out of memory, 29 time out, and 42 return unknown. Our solver performs the worst on the

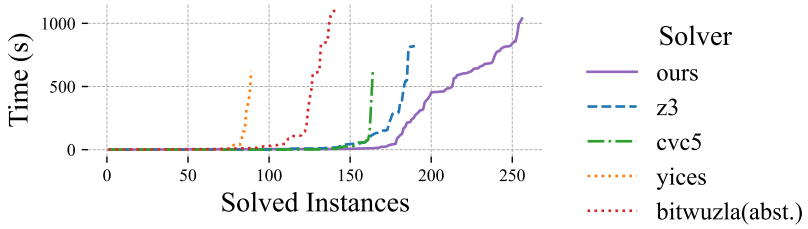

Fig. 5: Benchmarks solved over time for top 5 solvers: ours (QF_MIA), z3 (QF_NIA), cvc5 (QF_NIA), yices (QF_NIA), and bitwuzla w/ abstractions (QF_BV)

`f/b(m)` family; it accounts for 34 of the `unknown` benchmarks. We believe the poor performance in this domain is due to our algorithm's inability to find liftable equalities necessary to detect unsatisfiability, even though such equalities exist in the ideal.

7.3 Evaluation of Lifting Methods

Recall that our method relies on identifying liftable equations, with Sections 5.4 and 5.5 introducing two potentially complementary approaches for this purpose. We now present the results of an evaluation comparing these lifting methods. As summarized in Table 3, the weighted Gröbner basis method achieves the best overall performance. However, on the `f/b(m)` benchmark family, the unweighted Gröbner basis method outperforms the weighted variant, supporting our hypothesis that the weighted Gröbner basis's inability to find certain liftable equalities contributes to its weaker performance in this category. Lifting based on integer linear constraints performs the poorest, identifying almost no liftable equalities required for refutation. Nevertheless, when combined with the weighted Gröbner basis method (in cases where the Gröbner basis method is not guaranteed to be complete per Theorem 3), this hybrid approach has the most unique solves (12).

8 Related work

There are many SMT theory solvers for different kinds of modular integer arithmetic. Some solvers target theories of non-linear integer arithmetic with an explicit modulus operator, such as the theory of integers [2, 8, 14, 17, 19, 35, 46–48, 53, 58, 77, 80], and the theory of bit-vectors [11, 39, 59, 61, 62]. These solvers can be used to solve multimodular systems, but generally perform poorly because they are designed to support *general* modular reduction (i.e., reduction modulo a variable) rather than reduction by constants. There are solvers that efficiently support reduction by a single constant or *class* of constants. For example, finite field solvers [40–42, 65, 66] support reduction modulo primes, while bit-vector solvers support reduction modulo powers of two. But, none of these can reason simultaneously about equations modulo a large prime and powers of two. The Omega test reasons about equations modulo arbitrary constants—but the equations must be linear [71]. Our procedure supports non-linear equations modulo arbitrary constants.

Some of the above solvers use Gröbner bases: an algebraic tool for understanding polynomial systems [12]. For example, most similar to our refutation

procedure, cvc5's finite field solver [65, 66] also relies on a Gröbner basis to detect unsatisfiability of constraint subsystems. However, unlike our approach, it does not separate bounds or apply lifting, lowering, or weighted ordering techniques to share expressions across subsystems.

Many computer algebra systems (CASs) implement algorithms for computing different kinds of Gröbner bases [1, 9, 21, 23, 27, 28, 36, 44, 55, 82, 84]. Following in this vein, our procedure uses strong Gröbner bases [63], and our implementation uses the Singular CAS [23].

Formal methods for modular arithmetic have a long history of applications to cryptography. Interactive theorem provers (ITPs) have been used to verify many cryptographic implementations [31, 70, 73], including the Fiat cryptography library, which is used in all major web-browsers [30]. ITPs have also been used to verify zero-knowledge proofs (ZKPs) [16, 18, 34, 51]. But ITP-based verification requires significant manual effort and expertise. SMT solvers for finite fields [67, 68] and static analyses [20, 79, 81] have been used to verify ZKPs automatically, but these tools are limited by challenging tradeoffs between scalability and generality.

9 Conclusion and Future Work

In this paper we presented a novel refutation procedure for multimodular constraints and two algorithms for sharing lemmas: one based on a weighted Gröbner basis and another utilizing integer linear constraints. Our experiments demonstrate improvement over state-of-the-art solvers on benchmarks that arise from verifying cryptographic implementations. They also show the promise of lifting algorithms, both individually and in combination. Nevertheless, substantial future work remains.

First, as discussed in Sections 5.4 and 5.5, none of our lifting algorithms are complete. Exploring solutions with more completeness guarantees could boost the performance of our method. Second, as shown in the extended version of the paper [69], lifting using linear integer constraints shows poor performance due to the limitations of existing solvers. A custom solver tailored to our encoding could lead to better results. Third, our method focuses on unsatisfiable benchmarks and does not address model construction for satisfiable instances. However, we believe that leveraging the subsystem structure could also be beneficial for restricting the search space of possible assignments for satisfiable multimodular problems. Finally, our method treats Gröbner basis computation as a black box. Exploring more iterative methods based on s-polynomials could also boost performance.

Acknowledgements We thank Ben Sepanski and Kostas Ferles for helpful conversations. We acknowledge funding from NSF grant number 2110397, the Stanford Center for Automated Reasoning, and the Simons foundation.

Disclosure of Interest Shankara Pailoor, Alp Bassa, and Işil Dillig are employed at Veridise. Alex Ozdemir and Sorawee Porncharoenwase previously worked at Veridise. Sorawee Porncharoenwase is currently employed at Amazon Web Services (AWS). Clark Barrett is an Amazon Scholar.

Bibliography

[1] J. Abbott and A. M. Bigatti. CoCoALib: A C++ library for computations in commutative algebra... and beyond. In *International Congress on Mathematical Software*, 2010.

[2] E. Ábrahám, J. H. Davenport, M. England, and G. Kremer. Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. *Journal of Logical and Algebraic Methods in Programming*, 119, 2021.

[3] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS*, 2022.

[4] P. S. Barreto, B. Lynn, and M. Scott. Constructing elliptic curves with prescribed embedding degrees. In *SCN*, 2003.

[5] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[6] C. Barrett and C. Tinelli. Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, 2018.

[7] P. Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *CRYPTO*, 1986.

[8] N. Bjørner and L. Nachmanson. Arithmetic solving in z3. In *CAV*, 2024.

[9] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system I: The user language. *Journal of Symbolic Computation*, 24(3-4):235–265, 1997.

[10] S. Bowe. BLS12-381: New zk-snark elliptic curve construction, Mar. 2017. https://electriccoin.co/blog/new-snark-curve/.

[11] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*, 2009.

[12] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, 1965.

[13] B. Buchberger. A theoretical basis for the reduction of polynomials to canonical forms. *SIGSAM Bulletin*, 1976.

[14] B. F. Caviness and J. R. Johnson. *Quantifier elimination and cylindrical algebraic decomposition*. Springer Science & Business Media, 2012.

[15] S. Chaliasos, J. Ernstberger, D. Theodore, D. Wong, M. Jahanara, and B. Livshits. SoK: What don't we know? understanding security vulnerabilities in SNARKs. In *USENIX Security*, 2024.

[16] C. Chin, H. Wu, R. Chu, A. Coglio, E. McCarthy, and E. Smith. Leo: A programming language for formally verified, zero-knowledge applications, 2021. Preprint at https://ia.cr/2021/651.

[17] A. Cimatti, A. Griggio, A. Irfan, M. Roveri, and R. Sebastiani. Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. *ACM TOCL*, 19(3), 2018.

[18] A. Coglio, E. McCarthy, E. Smith, C. Chin, P. Gaddamadugu, and M. Dellepere. Compositional formal verification of zero-knowledge circuits, 2023. https://ia.cr/2023/1278.

[19] F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Ábrahám. SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In *SAT*, 2015.
[20] F. Dahlgren. It pays to be Circomspect. https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/, 2022. Accessed: 15 October 2023.
[21] J. Davenport. The axiom system, 1992.
[22] C. David. Ideals, Varieties, and Algorithms-An Introduction to Computational Algebraic Geometry and Commutative Algebra. *Undergraduate Texts in Mathematics*, 1991.
[23] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. Singular 4-4-0 — A computer algebra system for polynomial computations. http://www.singular.uni-kl.de, 2024.
[24] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), 1976.
[25] D. S. Dummit and R. M. Foote. *Abstract algebra*, volume 3. Wiley Hoboken, 2004.
[26] B. Dutertre. Yices 2.2. In *CAV*, 2014.
[27] C. Eder and T. Hofmann. Efficient Gröbner bases computation over principal ideal rings. *Journal of Symbolic Computation*, 103:1–13, 2021.
[28] D. Eisenbud, D. R. Grayson, M. Stillman, and B. Sturmfels. *Computations in algebraic geometry with Macaulay 2*, volume 8. Springer Science & Business Media, 2001.
[29] H. B. Enderton. *A mathematical introduction to logic*. Elsevier, 2001.
[30] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Systematic generation of fast elliptic curve cryptography implementations. Technical report, MIT, 2018.
[31] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *ACM SIGOPS Operating Systems Review*, 54(1), 2020.
[32] J. C. Faugére. A new efficient algorithm for computing Gröbner bases (f4). *Journal of Pure and Applied Algebra*, 139(1):61–88, 1999.
[33] J. C. Faugére. A new efficient algorithm for computing Gröbner bases without reduction to zero (f5). In *ISSAC*. ACM, 2002.
[34] C. Fournet, C. Keller, and V. Laporte. A certified compiler for verifiable computing. In *CSF*, 2016.
[35] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1(3-4), 2006.
[36] GAP – Groups, Algorithms, and Programming, Version 4.13dev. https://www.gap-system.org, this year.
[37] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.
[38] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal of Computation*, 18(1):186–208, 1989.
[39] S. Graham-Lengrand, D. Jovanović, and B. Dutertre. Solving bitvectors with MCSAT: explanations from bits and pieces. In *IJCAR*, 2020.
[40] T. Hader. Non-linear SMT-reasoning over finite fields, 2022. MS Thesis (TU Wein).
[41] T. Hader, D. Kaufmann, and L. Kovács. SMT solving over finite field arithmetic. In *LPAR*, 2023.
[42] T. Hader and L. Kovács. Non-linear SMT-reasoning over finite fields. In *SMT*, 2022. Extended Abstract.

[43] M. Hamburg. Ed448-goldilocks, a new elliptic curve, 2015. https://ia.cr/2015/625.
[44] A. Heck and W. Koepf. *Introduction to MAPLE*, volume 1993. 1993.
[45] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *ACM*, 48(5):1038–1068, Sept. 2001.
[46] D. Jovanović. Solving nonlinear integer arithmetic with MCSAT. In *VMCAI*, 2017.
[47] D. Jovanović and L. De Moura. Solving non-linear arithmetic. *ACM Communications in Computer Algebra*, 46(3/4), 2013.
[48] D. Jovanović and L. d. Moura. Cutting to the chase solving linear integer arithmetic. In *CADE*, 2011.
[49] A. Kosba, C. Papamanthou, and E. Shi. xJsnark: A framework for efficient verifiable computation. In *IEEE S&P*, 2018.
[50] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV*. Springer.
[51] J. Liu, I. Kretz, H. Liu, B. Tan, J. Wang, Y. Sun, L. Pearson, A. Miltner, I. Dillig, and Y. Feng. Certifying zero-knowledge circuits with refinement types, 2023. https://ia.cr/2023/547.
[52] A. Makhorin. GNU linear programming kit version 4.6.0. http://www.gnu.org/software/glpk/glpk.html, 2024.
[53] A. Maréchal, A. Fouilhé, T. King, D. Monniaux, and M. Périn. Polyhedral approximation of multivariate polynomials using handelman's theorem. In *VMCAI*, 2016.
[54] N. H. McCoy. *Rings and ideals*, volume 8. American Mathematical Soc., 1948.
[55] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.
[56] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.
[57] L. d. Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
[58] L. d. Moura and D. Jovanović. A model-constructing satisfiability calculus. In *VMCAI*, 2013.
[59] A. Niemetz and M. Preiner. Ternary propagation-based local search for more bit-precise reasoning. In *FMCAD*, 2020.
[60] A. Niemetz and M. Preiner. Bitwuzla. In *CAV*, 2023.
[61] A. Niemetz, M. Preiner, A. Reynolds, Y. Zohar, C. Barrett, and C. Tinelli. Towards bit-width-independent proofs in SMT solvers. In *CADE*, 2019.
[62] A. Niemetz, M. Preiner, and Y. Zohar. Scalable bit-blasting with abstractions. In *CAV*, 2024.
[63] G. H. Norton and A. Sălăgean. Strong Gröbner bases for polynomials over a principal ideal ring. *Bulletin of the Australian Mathematical Society*, 64(3):505–528, 2001.
[64] o1-labs. Foreign field multiplication gate, 2024. https://github.com/o1-labs/rfcs/blob/eeb8070c9901c611c9a557464022bbf9237900b9/0006-ffmul-revised.md.
[65] A. Ozdemir, G. Kremer, C. Tinelli, and C. Barrett. Satisfiability modulo finite fields. In *CAV*, 2023.
[66] A. Ozdemir, S. Pailoor, A. Bassa, K. Ferles, C. Barrett, and I. Dillig. Split Gröbner Bases for satisfiability modulo finite fields. In *CAV*, 2024.
[67] A. Ozdemir, R. S. Wahby, F. Brown, and C. Barrett. Bounded verification for finite-field-blasting. In *CAV*, 2023.

[68] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez, J. Van Geffen, J. Morton, M. Chu, B. Gu, Y. Feng, and I. Dillig. Automated detection of under-constrained circuits in zero-knowledge proofs. In *PLDI*, 2023.
[69] E. Pertseva, A. Ozdemir, S. Pailoor, A. Bassa, S. Porncharoenwase, I. Dillig, and C. Barrett. Integer reasoning modulo different constants in smt, 2025. https://arxiv.org/abs/2505.14998.
[70] J. Philipoom. *Correct-by-construction finite field arithmetic in Coq*. PhD thesis, Massachusetts Institute of Technology, 2018.
[71] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *SC*, 1991.
[72] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
[73] P. Schwabe, B. Viguier, T. Weerwag, and F. Wiedijk. A Coq proof of the correctness of X25519 in TweetNaCl. In *CSF*, 2021.
[74] Y. Sheng, A. Nötzli, A. Reynolds, Y. Zohar, D. Dill, W. Grieskamp, J. Park, S. Qadeer, C. Barrett, and C. Tinelli. Reasoning about vectors using an SMT theory of sequences. In *IJCAR*, 2022.
[75] Succinct Labs. Gnark Plonky2 recursive verifier: The goldilocks field implementation, 2024. https://github.com/succinctlabs/gnark-plonky2-verifier/tree/7025b2efd67b5ed30bd85f93c694774106d21b3d/goldilocks.
[76] C. Tsay, J. Kronqvist, A. Thebelt, and R. Misener. Partition-based formulations for mixed-integer optimization of trained relu neural networks. *NeurIPS*, 2021.
[77] V. X. Tung, T. V. Khanh, and M. Ogawa. raSAT: An SMT solver for polynomial constraints. In *IJCAR*, 2016.
[78] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, 2015.
[79] F. Wang. Ecne: Automated verification of ZK circuits, 2022. https://0xparc.org/blog/ecne.
[80] V. Weispfenning. Quantifier elimination for real algebra—the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8(2), 1997.
[81] H. Wen, J. Stephens, Y. Chen, K. Ferles, S. Pailoor, K. Charbonnet, I. Dillig, and Y. Feng. Practical security analysis of zero-knowledge proof circuits, 2023. https://ia.cr/2023/190.
[82] S. Wolfram. *Mathematica: a system for doing mathematics by computer*. Addison Wesley Longman Publishing Co., Inc., 1991.
[83] A. C. Yao. Protocols for secure computations. In *FOCS*, 1982.
[84] P. Zimmermann, A. Casamayou, N. Cohen, G. Connan, T. Dumont, L. Fousse, F. Maltey, M. Meulien, M. Mezzarobba, C. Pernet, et al. *Computational mathematics with SageMath*. SIAM, 2018.