# Split Gröbner Bases for Satisfiability Modulo Finite Fields

Alex Ozdemir[1,2], Shankara Pailoor[2], Alp Bassa[2], Kostas Ferles[2], Clark Barrett[1], and Işil Dillig[2]

[1] Stanford University (aozdemir@cs.stanford.edu)
[2] Veridise

**Abstract.** Satisfiability modulo finite fields enables automated verification for cryptosystems. Unfortunately, previous solvers scale poorly for even some simple systems of field equations, in part because they build a full Gröbner basis (GB) for the system. We propose a new solver that uses multiple, simpler GBs instead of one full GB. Our solver, implemented within the cvc5 SMT solver, admits specialized propagation algorithms, e.g., for understanding bitsums. Experiments show that it solves important bitsum-heavy determinism benchmarks far faster than prior solvers, without introducing much overhead for other benchmarks.

## 1 Introduction

Finite fields are critical to many cryptosystems. They underlie the AES-GCM cipher and ECDH key-exchange, which are used in over 80% of web requests [2, 43]. They also underlie zero-knowledge proof systems (ZKPs) and multi-party computation protocols that are used in billion-dollar private cryptocurrencies [28, 29, 41, 47], private DNS filters [35], agricultural auctions [8], discrimination studies [5], and US inter-agency data sharing [3].

Since (finite-)field-based cryptosystems are so prevalent, bugs in their implementations can have serious consequences. Furthermore, such bugs are not hypothetical. They routinely cause CVEs in OpenSSL [18, 19, 49] and compromise cryptocurrencies [1, 58, 64].

Motivated by this problem, recent research has explored automated verification for field-based computations [52, 54]. However, these techniques inherit scalability challenges from the field-solving capabilities of current Satisfiability Modulo Theories (SMT) solvers. The best SMT solver [52] for fields of cryptographic size ($\approx 2^{256}$) uses Gröbner bases (GBs) [10]. A GB can answer many questions about a system of equations, but the GB itself must first be computed.

Unfortunately, computing a GB has high theoretical complexity: doubly exponential in the worst case [46]. In practice, computing a GB *can* be feasible for some systems [52], but it is intractable for others, even simple ones. For example, consider a prime field—representable as the integers modulo a prime $p$. Suppose that $p \geq 2^b$ and consider the following system in variables $X_1, \ldots, X_b, Z$:

$$\bigwedge_{i=1}^{b} X_i(1-X_i) = 0 \quad \wedge \quad X_1 + 2X_2 + 4X_3 + \cdots + 2^{b-1}X_b = 0 \quad \wedge \quad X_b Z = 1$$

In some sense, this system is simple: the first equation forces each $X_i$ to be 0 or 1, and the second equation forces every $X_i$ to be 0, which then contradicts the final equation. However, computing a GB for this system using current algorithms takes exponential time. We investigate systems like this in Section 3, but essentially there are two conclusions: first, a GB is hard to compute because of the *combination* of the bitsum $\sum_i 2^{i-1} X_i$ and the bit constraints $X_i(1-X_i) = 0$; second, bitsums and bit constraints are common when verifying systems that use ZKPs. So, the scalability of GB-based reasoning with bitsums is a real problem for ZKP verification.

To overcome this problem, we present a new approach for solving or refuting a system $S$ of finite field equations. The key idea is that of a *split Gröbner basis*. If $S$ is split into (possibly overlapping) subsystems $S_1 \wedge \cdots \wedge S_k = S$, and $B_i$ is a GB for $S_i$, then we call the sequence $B_1, \ldots, B_k$ a split GB for $S$. A split GB approximates a full GB for $S$: it gives detailed information about each subsystem $S_i$, but more limited information about $S$. In exchange for this approximation, if each $S_i$ is "small" or "simple," then the split GB might be easier to compute.

In this paper, we present a decision procedure for finite field arithmetic based on the idea of *iteratively refining* a split GB. It starts with some split of $S$ and then refines it as necessary by sharing equations between the $S_i$'s. We also add an extensible *propagation* algorithm for deducing new equations. Sharing equations increases the cost of computing the split basis but also improves the approximation that it offers. The key advantage is that the procedure can often solve or refute $S$ before any $S_i$ becomes too hard to compute a basis for.

We implement our approach as a solver for prime fields within the cvc5 SMT solver [4]. Our solver (a) splits bitsums and their bit constraints across two subsystems and (b) includes a specialized propagator for bitsum reasoning. This is particularly effective for important, bitsum-heavy verification problems related to ZKPs. For these problems, experiments show that our solver exponentially improves on prior work; for other problems, it has low overhead.

One application we consider is verifying field blaster ($\mathbb{F}$-blaster) rules in a ZKP compiler: these rules encode Boolean and bit-vector operations as (conjunctions of) field equations (see Sec. 2). We give a new SMT encoding for rule correctness, prove our encoding is correct, and show that combining it with our new solver improves the state of the art for $\mathbb{F}$-blaster verification [53]. To summarize, our key contributions are:

1. Split: an abstract decision procedure for field solving using a split Gröbner basis instead of a full Gröbner basis.
2. BitSplit: an instantiation of Split, optimized for bitsums and implemented in cvc5. It is exponentially faster than prior solvers on important benchmarks.
3. An application: a new encoding for $\mathbb{F}$-blaster verification conditions that improves the state of the art for $\mathbb{F}$-blaster verification by leveraging BitSplit.

The rest of the paper is organized as follows. First, we review related work (§1.1), give background (§2), and present a motivating example (§3). Then, we explain our abstract and concrete decision procedures (§4) and present experiments (§5). Last, we apply our solver to the problem of verified $\mathbb{F}$-blasting (§6).

### 1.1 Related Work

There are two prior finite field solvers for SMT: Hader et al. [37, 39, 40] use subresultant regular subchains [59], and Ozdemir et al. [52] use Gröbner bases. As we will see (Sec. 5), only the latter scales to large fields. Our work builds on it.

Other prior works propose verification and linting tools for ZKPs. $\mathsf{QED}^2$ [54] checks whether an output variable $Y$ in some system is uniquely determined by the values of input variables $X_1, \ldots, X_m$. Another project [53] verifies that a ZKP compiler's $\mathbb{F}$-blaster is correct. These both use satisfiability modulo finite fields and could benefit from our work. Other tools are purely syntactic [20, 60, 61].

Further afield, others consider finite fields in *interactive* theorem provers, applied to mathematics [9, 16, 32, 42], to program correctness [25, 26, 55, 56], and even to ZKPs [13, 15, 30, 44]. In contrast, our work is fully automatic.

## 2 Background

Here we summarize necessary definitions and facts about finite fields [21, Part IV], computer algebra [17], satisfiability modulo finite fields (SMFF) [48, 52], and applications of SMFF [53, 54]. See the references for further details.

**Finite Fields and Polynomials.** For naturals $a \geq 1$, $[a]$ denotes $\{1, \ldots, a\}$. In general, $\mathbf{x}$ denotes a list of elements $x_1, \ldots, x_m$. Let $p$ be a prime. $\mathbb{F}_p$ (abbreviated $\mathbb{F}$ when $p$ is clear) denotes the unique finite field of order $p$, represented as $\{0, \ldots, p-1\}$ with addition and multiplication modulo $p$. A field of prime order is also called a *prime field*. Let $\mathbf{X}$ be a list of $n$ variables: $(X_1, \ldots, X_n)$. $\mathbb{F}[\mathbf{X}]$ is the set of polynomials in $\mathbf{X}$ with coefficients from $\mathbb{F}$. For $f \in \mathbb{F}[\mathbf{X}]$, let $\mathsf{deg}(f)$ be its degree and $\mathsf{vars}(f)$ be the set of variables appearing in it.

**Ideals and their Zeros.** Let $S = \{s_1, \ldots, s_m\}$ be a set of polynomials in $\mathbb{F}[\mathbf{X}]$. $\langle S \rangle$ denotes the *ideal* that is *generated* by $S$: the set $\{\sum_i f_i s_i : f_i \in \mathbb{F}[\mathbf{X}]\}$. Let $\mathbf{S} = (S_1, \ldots, S_k)$ be a list of *sets* of polynomials. Then, we define $\langle \mathbf{S} \rangle \triangleq \langle \cup_i S_i \rangle$.

Let $M : \mathbf{X} \to \mathbb{F}$ be a map from variables $\mathbf{X}$ to values in $\mathbb{F}$. For $f \in \mathbb{F}[X]$, denote the evaluation of $f$ on $M$ by $f[M]$; a *zero* of $f$ is an $M$ with $f[M] = 0$. The common zeros of $S$ are denoted $\mathcal{V}_{\mathbb{F}}(S)$ (abbreviated $\mathcal{V}(S)$). Note that $\mathcal{V}(S) = \mathcal{V}(\langle S \rangle)$. When studying polynomial systems, one generally considers the system given by the ideal it generates, as it has more structure and has the same set of zeros. For any $f \in \mathbb{F}[\mathbf{X}]$, if $f \in \langle S \rangle$, then $\mathcal{V}(\{f\}) \supseteq \mathcal{V}(S)$. One implication of this is that $1 \in \langle S \rangle$ implies that $\mathcal{V}_{\mathbb{F}}(S)$ is empty. However, the converse does not hold: for example, the polynomial $X^2 + 1$ has no zero in $\mathbb{F}_3$, but $1 \notin \langle X^2 + 1 \rangle$.

**Gröbner bases.** A Gröbner basis (GB) is a kind of polynomial set that is often used for solving polynomial systems. Two facts about GBs are relevant to this paper. First, there is an algorithm, $\mathsf{GB}$, that for any polynomial set $S$, computes a GB $B$ such that $\langle B \rangle = \langle S \rangle$. In this case, we say that $B$ is a GB for $S$ or for $\langle S \rangle$. (But: note that in this paper, $B$ does not always refer to a GB!) Second, there is an algorithm $\mathsf{InIdeal}(f, B)$ that determines whether $f \in \langle B \rangle$ for

polynomial $f$ and GB $B$.[3] Thus, if $\mathsf{InIdeal}(1, \mathsf{GB}(S))$ returns true, this shows that $\mathcal{V}(S)$ is empty. Moreover, $\mathsf{InIdeal}(1, B)$ is computable in polytime if $B$ is a GB since 1 reduces by $B$ iff $B$ contains a non-zero constant [17].

**Satisfiability Modulo Finite Fields (SMFF).** Previous work [39, 52] defines the theory of finite fields, which we summarize here using the usual terminology of many-sorted first order logic with equality [24]. For every finite field $\mathbb{F}$, let the signature $\Sigma$ include: sort $\mathsf{F}_\mathbb{F}$, binary function symbols $+_\mathbb{F}$ and $\times_\mathbb{F}$, constants $n \in \{0, \ldots, |\mathbb{F}| - 1\} \subset \mathbb{N}$, and the inherited equality symbol $\approx_\mathbb{F}$. The theory of finite fields requires that any $\Sigma$-interpretation interprets $\mathsf{F}_\mathbb{F}$ as $\mathbb{F}$, $n$ as the $n^{\text{th}}$ element of $\mathbb{F}$, and $+$, $\times$, and $\approx$ as addition, multiplication, and equality in $\mathbb{F}$. Previous work reduces the satisfiability problem for this theory to the problem of finding an element of $\mathcal{V}(S)$ given $S$ or determining that there is no such element [52]. In this work, we consider the latter problem.

**Applying SMFF to ZKPs.** Prior work applies SMFF to verification for zero-knowledge proof systems (ZKPs) [52–54]. Practical ZKPs [11, 31, 34] allow one to prove knowledge of a solution to a system of *field equations* $\Phi(\mathbf{X}, \mathbf{Y})$, while keeping all or part of the solution secret. Since $\Phi$ is usually meant to encode a function from $\mathbf{X}$ to $\mathbf{Y}$, recent tools attempt to verify *determinism*: that the value of $\mathbf{X}$ uniquely determines the value of $\mathbf{Y}$ [54, 57, 60, 62]. Determinism can be written as a single satisfiability query solved with SMFF:

$$\Phi(\mathbf{X}, \mathbf{Y}) \wedge \Phi(\mathbf{X}', \mathbf{Y}') \wedge \mathbf{X} = \mathbf{X}' \wedge \mathbf{Y} \neq \mathbf{Y}' \tag{1}$$

The formula (1) is satisfiable if and only if $\Phi$ is **non**deterministic. Determinism is important for two reasons. First, constructing (1) only requires identifying the inputs and outputs, making the specification task trivial and automatable. Second, determinism violations are frequent; one caused the Tornado Cash bug [58], and they are part of over half of the bugs in the ZK Bug Tracker [1]. Third, determinism violations cause real vulnerabilities. A recent survey of ZKP vulnerabilities concludes that insufficient constraints (which typically manifest as non-determinism) account for 95% of constraint-system-level vulnerabilities [12]. In Section 6, we give another reason why determinism is important: it can imply stronger properties.

## 3 Motivating Example

In this section, we explore a class of problems that is both important and challenging for existing SMFF solvers. First (§3.1), we explain the source and prevalence of these problems—determinism queries with bit-splitting. Second (§3.2), we explore why they are hard for GB-based reasoning, and we present evidence that the core challenge is the combination of bitsums and bit-constraints. Third (§3.3), we sketch the design of a decision procedure that can meet this challenge.

[3] The definition of GB and these algorithms depends on a *monomial order*. Throughout the paper, we use grevlex order. We discuss monomial orders in Appendix A.

```
1  template Num2Bits(b) { // split 'in' into 'b' bits.
2      signal input in;
3      signal output out[b];
4      var bitSum = 0;
5      for (var i = 1; i <= b; i++) {
6          out[i] * (out[i] -1 ) === 0; // 'out[i]' is 0 or 1
7          bitSum += out[i] * 2 ** (i - 1); // add a term to the accumulating bitsum
8      }
9      bitSum === in; // 'in' is the bitsum of 'out'
10 }
```

Fig. 1: `Num2Bits`: a widely-used circomlib library function. It converts a prime field element into an $b$-bit binary representation (assuming this is possible).

### 3.1 Verifying the determinism of `Num2Bits`

The circom language is used to synthesize field equations for ZKPs. Figure 1 shows a slice of the circom program `Num2Bits`. It relates an input signal $in$ to its binary representation as an array of signals $out$. The code generates a set of field equations that encode this relationship. The `===` operator generates equations. Line 6 generates the equation forcing $out[i]$ to be either 1 or 0, line 7 adds $out[i]$ to the expression that is accumulating terms in the bitsum, and line 9 generates the equation equating the bitsum to $in$. Thus, the equations are:

$$\Phi(in, out) := \left(in = \textstyle\sum_{i=1}^{b} 2^{i-1} out[i]\right) \wedge \bigwedge_{i=1}^{b} out[i](out[i] - 1) = 0 \qquad (2)$$

Here, $b$ is constant. For any $j \in [b]$, the output $out[j]$ is deterministic if the following SMFF query is unsatisfiable:

$$\exists\, in, in', out, out'.\ \ \Phi(in, out) \wedge \Phi(in', out') \wedge in = in' \wedge out[j] \neq out'[j] \qquad (3)$$

*Importance.* Nearly every circom project uses `Num2Bits` or similar templates that bit-split field elements. This is because bit encodings are a natural way to encode common operations like range-checks ($x \in \{l, \ldots, u\}$) and comparisons ($<, >$) as field equations. In fact, in a crawl of all public circom Github projects, we found that 98% of projects use `Num2Bits` or other circuits with bitsums. Furthermore, bitsums are *very* common in many programs; for example, in circomlib's SHA2 implementation, 64% of the variables appear in some bitsum. We describe our methodology for these measurements in Appendix B.

### 3.2 The challenge of bit-splitting

Unfortunately, state-of-the-art SMFF solvers struggle with (3). The solver of Hader et al. [39] scales poorly with field size (Sec. 5), and ZKP security typically requires $|\mathbb{F}| \approx 2^{255}$. It fails for (3), even when $b = 1$. The GB-based solver of Ozdemir et al. [52] scales better with $|\mathbb{F}|$, but poorly with $b$. It can handle many large-field benchmarks, but it cannot solve (3) for $b = 32$, even in a week.

| Ideal Family | Generators |
|---|---|
| $I_{2,\mathrm{det}}(b)$ | $\mathsf{B\Sigma P}(Y,\mathbf{X}) \cup \mathsf{B\Sigma P}(Y',\mathbf{X}') \cup \{Y-Y'\} \cup \{(X_b - X'_b)Z - 1\}$ |
| $I_2(b)$ | $\mathsf{B\Sigma P}(Y,\mathbf{X}) \cup \mathsf{B\Sigma P}(Y',\mathbf{X}') \cup \{Y-Y'\}$ |
| $I_1(b)$ | $\mathsf{B\Sigma P}(Y,\mathbf{X})$ |
| $I_{1,\mathrm{val}}(b)$ | $\mathsf{B\Sigma P}(Y,\mathbf{X}) \cup \{Y\}$ |

Table 1: Different ideal families with bitsums and bit-constraints.

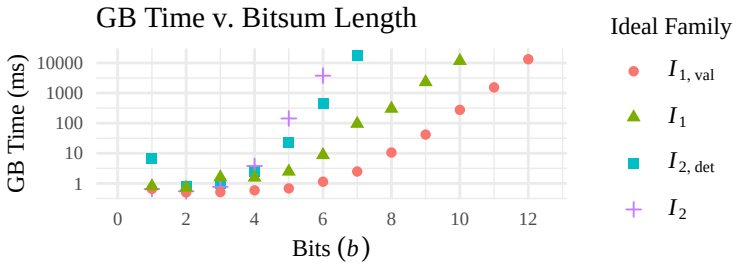


Fig. 2: GB computation time for different systems at different bitsum lengths.

To understand the problem, consider how a GB-based solver handles (3). First, it computes a polynomial set $S$ such that $\mathcal{V}(S)$ encodes solutions to (3):

$$\begin{aligned} S = \{&Y - Y', \quad Y - \textstyle\sum_{i=1}^{b} 2^{i-1} X_i, \quad Y' - \sum_{i=1}^{b} 2^{i-1} X'_i, \\ &X_1^2 - X_1, \ldots, X_b^2 - X_b, \quad {X'}_1^2 - {X'}_1, \ldots, {X'}_b^2 - {X'}_b, \\ &(X'_j - X_j)Z - 1\} \end{aligned} \tag{4}$$

In this system, *in*, *in*′, *out*, and *out*′ are represented by variables $Y$, $Y'$, $\mathbf{X}$, and $\mathbf{X}'$ respectively. The inequality $X_j \neq X'_j$ becomes the polynomial $(X'_j - X_j)Z - 1$ (for fresh $Z$) which can be zero only if $X_j \neq X'_j$. Next, the solver attempts to compute a GB for (4). But this takes time exponential in $b$, as we will see.

To empirically investigate the cause of the slowdown, we consider other families of ideals generated by sets similar to (4). Table 1 shows four ideal families of increasing simplicity that all include bit-splitting. The polynomials are in variables $(X_1, \ldots, X_b, X'_1, \ldots, X'_b, Y, Y', Z)$, and we define the set $\mathsf{B\Sigma P}(Y, \mathbf{X})$ as:

$$\mathsf{B\Sigma P}(Y, (X_1, \ldots, X_b)) \triangleq \{Y - \textstyle\sum_{i=1}^{b} 2^{i-1} X_i, X_1^2 - X_1, \ldots X_b^2 - X_b\}.$$

The first family, $I_{2,\mathsf{det}}(b)$, is exactly (4), for $j = b$. The second, $I_2$, removes the polynomial that enforces disequality. The third, $I_1$, removes one of the bitsum and bit-constraint sets. The fourth, $I_{1,\mathsf{val}}$, fixes the lone bitsum to a specific value $(Y = 0)$. Computing a GB for *any of these families* takes time exponential in $b$.[4]

[4] For Figure 2, we work in $\mathbb{F}_p$, where $p$ is the smallest prime greater than $2^b - 1$. However, the results are similar for other values of $p$ as well.

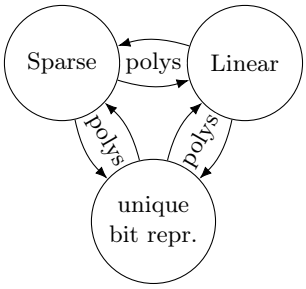


Fig. 3: High-level information flow in BitSplit: our concrete decision procedure.

Figure 2 shows the times (using Singular [33]; others are similar). $I_{1,\mathsf{val}}$ is easiest to compute a GB for, and $I_2$ is the hardest, but all take exponential time.

Interestingly, the singleton set of just the bitsum $\{Y - \sum_{i=1}^{b} 2^{i-1} X_i\}$ and the set of bit-constraints without the bitsum $\{X_1^2 - X_1, \ldots X_b^2 - X_b\}$ are both already GBs. It appears that the **combination** of the bitsum and the bit-constraints is what makes computing a GB hard.

**Translation to bit-vectors: a dead end** Since ZKPs process finite-field equations, the system (2) has coefficients in a *finite field*. Yet, the appearance of the bitsum pattern makes it tempting to attempt some kind of translation into the bit-vector domain. After all, in that domain, bit-decomposition is easy to reason about! However, this intuitive appeal is misleading. In practice, the approach is not trivial, since (in the general case) the system $\Phi$ includes other (non-bitsum) equations too. In fact, previous attempts to solve finite-field equations by translation to bit-vectors have been shown to be very ineffective [51]. Thus, performing some finite-field reasoning seems crucial.

### 3.3 Cooperative reasoning: a path forward

We have seen that verifying `Num2Bits` is hard with only GBs. Yet, `Num2Bits` is easy to verify when we combine GBs with other kinds of reasoning. Consider the following inferences about $\langle S \rangle$ (Eq. 4): Since $\mathbf{X}, \mathbf{X}'$ are bit representations of $Y, Y'$ respectively and $Y - Y'$ is in $\langle S \rangle$, every $X_i' - X_i$ must be too. This is the *congruence* rule for the function from a number to its bit representation. Then, since $f = X_j' - X_j$ and $g = (X_j' - X_j)Z - 1$ are both in $\langle S \rangle$ a GB shows that $1 = fZ - g$ is also in $\langle S \rangle$. But, if $1 \in \langle S \rangle$, then $S$ can have no common zeros. So, (3) is UNSAT, and `Num2Bits` is deterministic. The key here is to use GB-based reasoning *and* non-GB-based reasoning (congruence for bit representations).

Our decision procedure BitSplit mixes GB-based and non-GB-based reasoning to understand the contents of an ideal $\langle S \rangle$. Figure 3 illustrates its architecture. There are three modules: each learns new polynomials in $\langle S \rangle$ and potentially shares them with other modules. The *sparse* module computes a GB for all polynomials except *bitsum polynomials* (or *bitsums*): those of form $Y - \sum_i 2^{i-1} X_i$. Its name refers to the fact that bitsums are dense: they have many terms. The

*linear* module computes a GB for all linear polynomials (including all bitsums). The *unique bit representation* module infers bit equalities using congruence.

This architecture has three key features. First, it includes non-GB-based reasoning. Second, every polynomial is handled by *some* GB-based module (either the sparse or linear module); this will play a role in correctness. Third, by splitting bitsums (which go into the linear module) and bit-constraints (which go into the sparse module), it avoids computing a GB for both simultaneously.

## 4 Approach

In this section, we present our decision procedure. Given a set of polynomials $G$, our procedure either finds a common zero $M \in \mathcal{V}(G)$ or determines that none exists. Recall from Sec. 2 that satisfiability modulo $\mathbb{F}$ reduces to this problem.

To explain our decision procedure, we first introduce a *split Gröbner basis* (§4.1), which can be easier to compute than a full GB, but can also be less useful when deciding satisfiability. Next, we present our abstract decision procedure Split, which manipulates split Gröbner bases (§4.2). Split is parameterized by the number of bases $k$ and also by some subroutines. We show that if the subroutines meet suitable conditions, then Split is sound and terminating (Thm. 3). Finally, we instantiate Split with $k = 2$ by defining the necessary subroutines (§4.3). The result is a concrete decision procedure BitSplit which is optimized for reasoning about bitsums.[5] We evaluate BitSplit experimentally in Section 5.

### 4.1 Split Gröbner bases

**Definition 1 (Split Gröbner basis).** *A **split Gröbner basis** for ideal $I$ is a sequence $(B_1, \ldots, B_k)$ of Gröbner bases such that $I = \langle \mathbf{B} \rangle$.*

We make a few relevant observations about this definition.

1. A split GB generalizes a GB: that is, $(\mathsf{GB}(S))$ is always a split GB for $\langle S \rangle$.
2. Split GBs for an ideal $I$ are not unique.
3. The split GB definition relaxes the GB definition: while GBs can be hard to compute, split GBs need not be. For example, the ideal $\langle f_1, \ldots, f_n \rangle$ has split GB $(\{f_1\}, \ldots, \{f_n\})$.

Informally, a split GB allows one to navigate a trade-off between the computational expense of computing GBs and the power of their ideal membership tests. Generally, a smaller split GB where each individual GB represents more of $I$ makes $\mathsf{InIdeal}(\cdot, B_i)$ more informative. On the other hand, a bigger split GB where each GB represents less of $I$ makes the split basis easier to compute. Section 3 gave an example of this: it is hard to compute a GB for $\langle \sum_{i=1}^{b} 2^{i-1} X_i, X_1^2 - X_1, \ldots, X_b^2 - X_b \rangle$, but $(\{\sum_{i=1}^{b} 2^{i-1} X_i\}, \{X_1^2 - X_1, \ldots, X_b^2 - X_b\})$ is *already* a split GB.

[5] We use the name "BitSplit" because the procedure is optimized for bitsums (used in bit-splitting) and because the name suggests an instantiation of the "Split" procedure.

```
1 Function Monolithic:
      In: G ⊂ 𝔽[X]
      Out: A zero M ∈ 𝒱(G) or ⊥
2
3     B ← GB(G);
4     if 1 ∈ ⟨B⟩ then return ⊥;
5     return FindZero(B)
```

(a) The prior decision procedure [52].

```
1 Function Split:
      In: G ⊂ 𝔽[X]
      Out: A zero M ∈ 𝒱(G) or ⊥
2     G ← ({p ∈ G : init(i, p)})_{i=1}^k;
3     B ← SplitGB(G);
4     if ∃ i. 1 ∈ ⟨B_i⟩ then return ⊥;
5     return SplitFindZero(B)
```

(b) Our abstract procedure Split.

Fig. 4: The prior decision procedure (Monolithic) [52] and our framework (Split).

```
1 Function SplitGB:
      In: G = (G_i)_{i=1}^k: a list of generator sets
      Out: B = (B_i)_{i=1}^k: a split GB; initially each B_i is empty.
2     while ∪_i G_i is not empty do
3         for i ∈ [k] do B_i ← GB(G_i ∪ B_i); G_i ← ∅;
4         for p ∈ (∪_j B_j) ∪ extraProp(B), i ∈ [k] do
5             if admit(i, p) ∧ p ∉ ⟨B_i⟩ then G_i ← G_i ∪ {p};
6     return B
```

**Algorithm 1:** SplitGB computes a split Gröbner basis, with propagation.

### 4.2 Abstract procedure: Split

Our starting point is a prior solver based on Gröbner bases [52]. Figure 4a shows the prior procedure, which we call Monolithic, and Fig. 4b shows our new procedure, which is named Split. Monolithic begins by computing a GB $B$ and returning $\bot$ if $1 \in \langle B \rangle$. Recall that $1 \in \langle B \rangle$ implies $\mathcal{V}(G)$ is empty, but the converse does not hold; thus, this is a sound but incomplete test for unsatisfiability. If the problem remains unsolved, then Monolithic proceeds to FindZero, which is a (complete) backtracking search over elements of $\mathbb{F}$.

The key difference in Split is that it works with a split GB $\mathbf{B}$ for $\langle G \rangle$. First (line 2), we split $G$ into subsets $G_1 \cup \cdots \cup G_k = G$; these may overlap. Second (line 3), we compute a Gröbner basis $B_i$ for each subset $G_i$ (and perform additional propagations, discussed later). If some $\langle B_i \rangle$ contains 1, we return $\bot$. Third (line 5), we fall back to a (complete) backtracking search based on $\mathbf{B}$. We will now discuss each phase in more detail.

**Splitting.** Splitting is done with a function $\mathsf{init}(i, p)$ that decides whether polynomial $p$ should initially be included in basis $i$. The function init is a parameter of Split. The only requirement of init is that no polynomial can be ignored:

**Definition 2 (Covering init).** *The function* init *is* ***covering*** *when for all* $p \in \mathbb{F}[\mathbf{X}]$*, there exists an* $i \in [k]$ *such that* $\mathsf{init}(i, p) = \top$.

**Computing a split GB and propagating.** In the second stage, we compute a split GB $\mathbf{B}$ using SplitGB (Alg. 1). To start, SplitGB sets each $B_i$ to be a

GB for $\langle G_i \rangle$. However, SplitGB also adds to each $B_i$ additional polynomials called *propagations*. Propagations can be *inter-basis* (from a different $B_j$) or *extra* (from a subroutine extraProp). Through extraProp, one can extend SplitGB with specialized reasoning (e.g., for bitsums). Whether a propagation $p$ is admitted into $B_i$ is controlled by a subroutine $\mathsf{admit}(i, p)$. Through admit, a basis can reject a polynomial $p$ that would slow down future GB computations.

Now, we explain SplitGB in detail. In each iteration of the outer loop, $B_i$ is a current basis and $G_i$ is a set of polynomials that will be added in the next round. First, $B_i$ is computed from the previous $G_i$ and $B_i$. Then, polynomials from each $B_j$ are added to each $G_i$ if $\mathsf{admit}(i, \cdot)$ accepts them and $\langle B_i \rangle$ doesn't contain them already. Any propagations from $\mathsf{extraProp}(\mathbf{B})$ are added in the same way. The loop iterates until there are no new additions.

The correctness of SplitGB depends on extraProp, but not admit. As captured by Definition 3, $\mathsf{extraProp}(\mathbf{B})$ must only return polynomials in $\langle \mathbf{B} \rangle$. If extraProp obeys this requirement, then SplitGB terminates and preserves the generated ideal, as stated in Theorem 1. The proof is in Appendix C; correctness is straightforward, and termination follows from the same theory that guarantees termination for Buchberger's algorithm [10]. We discuss efficiency later.

**Definition 3 (Sound extraProp).** *The function* extraProp *is* ***sound*** *when for all* $\mathbf{B} \in (2^{\mathbb{F}[\mathbf{X}]})^k$, $\mathsf{extraProp}(\mathbf{B}) \subseteq \langle \mathbf{B} \rangle$.

**Theorem 1.** *If* extraProp *is sound, then* $\mathsf{SplitGB}(\mathbf{G})$ *terminates and returns a split Gröbner basis* $\mathbf{B}$ *such that* $\langle \mathbf{B} \rangle = \langle \mathbf{G} \rangle$ *and* $\langle B_i \rangle \supseteq \langle G_i \rangle$ *for all* $i$.

**Backtracking search.** SplitFindZero (Alg. 2) is our conflict-driven search. Given a split basis $\mathbf{B}$, it returns $M \in \mathcal{V}(\langle \mathbf{B} \rangle)$ if possible, and $\perp$ if $\mathcal{V}(\langle \mathbf{B} \rangle)$ is empty. It uses a subroutine $\mathsf{SplitZeroExtend}(\mathbf{B})$ which searches for an $M \in \mathcal{V}(\langle \mathbf{B} \rangle)$ by focusing on $B_1$, as we explain below. SplitZeroExtend returns one of three possibilities: an $M \in \mathcal{V}(\langle \mathbf{B} \rangle)$; $\perp$, indicating that $\mathcal{V}(\langle \mathbf{B} \rangle)$ is empty; or a *conflict polynomial* $p \in (\cup_i B_i) \backslash \langle B_1 \rangle$ that it failed to account for in its $B_1$-focused search. In the last case, SplitFindZero adds $p$ to $B_1$ and tries SplitZeroExtend again. Each conflict is new information that is added to $B_1$ from some other $B_i$.

SplitZeroExtend is based on the FindZero algorithm of prior work [52]. FindZero is a backtracking search based on a GB $B$. In each recursive step, it assigns a single variable to a single value. Rather than doing an exhaustive case split for each variable, a subroutine ApplyRule analyzes $B$ and constructs a list (an implicit disjunction) of single-variable assignments $X_{j_1} \mapsto z_1, \ldots, X_{j_\ell} \mapsto z_\ell$ that *cover* $\mathcal{V}(B)$. That is, for each $M \in \mathcal{V}(B)$, there exists $i$ such that $M[X_{j_i}] = z_i$. Thus, we know that if a solution exists, it must agree with at least one of these assignments. For example, with $B = \{X_1^2 - X_2, X_1(X_2 - 1)\}$, every solution must assign $X_1$ to 0 or $X_2$ to 1, so any set of assignments including these would do. ApplyRule might, for instance, return exactly $\{X_1 \to 0, X_2 \to 1\}$. For each $i$, FindZero recurses on $B \leftarrow \mathsf{GB}(B \cup \{X_{j_i} - z_i\})$. It backtracks if $1 \in \langle B \rangle$ and succeeds if every variable has been assigned.

SplitZeroExtend adapts FindZero to a split GB, essentially by running FindZero on $B_1$ and using SplitGB instead of GB. It also uses a limited notion of conflicts

```
1  Function SplitFindZero:
       In: B = (B_i)_{i=1}^k: a split GB
       Out: A zero M ∈ V(⟨B⟩) or ⊥
2      while conflict p ← SplitZeroExtend(B) do
3          B ← SplitGB(B_1 ∪ {p}, B_2, ..., B_k);
4      return SplitZeroExtend(B)
5  Function SplitZeroExtend:
       In: B = (B_i)_{i=1}^k: the current split GB
       In: G ⊂ F[X]: the original generators; if omitted, equal to ∪_i B_i
       In: A partial map M : X → F; if omitted, empty
       Out: A total map M or a conflict polynomial p or ⊥
6      if ∃i. 1 ∈ ⟨B_i⟩ then
7          if ∃p ∈ G \ ⟨B_1⟩, vars(p) ⊆ vars(M) ∧ p[M] ≠ 0 then return p;
8          else return ⊥;
9      if |M| = n then return M;
10     for (X_{j_i} ↦ z_i) ∈ ApplyRule(B_1, M) do
11         r ← SplitZeroExtend(SplitGB((B_j ∪ {X_{j_i} − z_i})_{j=1}^k), G, M ∪ {X_{j_i} ↦ z_i});
12         if r ≠ ⊥ then return r;
13     return ⊥
```

**Algorithm 2:** SplitFindZero finds zeros using split Gröbner bases.

to prune the search space. It is given a split basis $\mathbf{B}$ (that changes in each recursion), a generator set $G$ (that is fixed across recursions and is initially equal to $\cup_i B_i$), and a partial map $M$ from variables to values. First (lines 6–8), it checks whether 1 is in any $\langle B_i \rangle$. There are two cases here. If some polynomial $p \in G \setminus \langle B_1 \rangle$ fully evaluates to a non-zero value, $p$ is returned as a conflict. Otherwise, $\bot$ is returned. Second (line 9), if $M$ is total, then it is returned as a common zero. Third (lines 10–12), SplitZeroExtend uses ApplyRule (from [52]) to obtain a list of single-variable assignments that cover $\mathcal{V}(B_1)$. For each assignment in the list, it attempts to construct a solution by adding that assignment to $M$ and to each $B_i$ and recursing. If no branch succeeds, it returns $\bot$.

For each conflict that SplitZeroExtend returns, SplitFindZero will call it again with a new starting split basis. Theorem 2 states the correctness of SplitFindZero. The correctness of Split (Theorem 3) is a corollary. The proofs are in Appendix D.

**Theorem 2.** *Let* $\mathbf{B}$ *be a split GB. If* extraProp *is sound then* SplitFindZero($\mathbf{B}$) *terminates and returns an element of* $\mathcal{V}_{\mathbb{F}}(\langle \mathbf{B} \rangle)$ *iff one exists.*

**Theorem 3.** *Let* $G$ *be a polynomial set. If* extraProp *is sound and* init *is covering, then* Split($G$) *terminates and returns an element of* $\mathcal{V}_{\mathbb{F}}(G)$ *iff one exists.*

### 4.3 Concrete procedure: BitSplit

**Bases.** To construct BitSplit, we instantiate Split with $k = 2$. We call $B_1$ the *sparse* basis and $B_2$ the *linear* basis, and we define init and admit as shown in Table 3. We explain extraProp later.

| Function signature | Semantics |
|---|---|
| $\mathsf{init}(i \in [k], p \in \mathbb{F}[\mathbf{X}]) \to \{\top, \bot\}$ | whether to initialize basis $B_i$ with $p$ |
| $\mathsf{admit}(i \in [k], p \in \mathbb{F}[\mathbf{X}]) \to \{\top, \bot\}$ | whether to accept $p$ into $B_i$ during propagation |
| $\mathsf{extraProp}(\mathbf{B} \in (2^{\mathbb{F}[\mathbf{X}]})^k) \to 2^{\mathbb{F}[\mathbf{X}]}$ | additional polynomials to propagate |

Table 2: The functions that parameterize Split.

| Basis # ($i$) | Name | $\mathsf{init}(i, p)$ definition | $\mathsf{admit}(i, p)$ definition |
|---|---|---|---|
| 1 | Sparse | $\neg\mathsf{isBitsum}(p)$ | $\mathsf{isEq}(p)$ |
| 2 | Linear | $\mathsf{deg}(p) \leq 1$ | $\mathsf{deg}(p) \leq 1$ |

Table 3: Which polynomials our bases accept. The linear basis accepts linear polynomials. The sparse basis accepts non-bitsums initially, and then equalities.

We carefully avoid allowing a bitsum $X - \sum_{i=0}^{k} 2^i X_i$ and its bit constraints $(X_i^2 - X_i)_{i=1}^k$ in the same basis. Initially, the sparse basis rejects only bitsums ($\mathsf{isBitsum}(p)$ is defined as $\exists\, \ell > 1,\ \exists Y, X_1, \ldots X_\ell \in \mathbf{X},\ p = Y - \sum_{i=0}^{\ell} 2^i X_i$). During propagation, the sparse basis accepts polynomials that encode equalities ($\mathsf{isEq}(p)$ is defined as $\exists\, X, Y \in \mathbf{X}, z \in \mathbb{F}, p = X - Y \vee p = X - z$). The linear basis accepts (in initialization and propagation) any linear polynomial. Our definition of admit is quite narrow (to accelerate calls to GB), but we ensure that both ideals accept equalities, since extraProp generates these. In our experiments, we consider some other definitions of admit, but they do not improve performance.

**Extra Propagation.** Our extraProp subroutine simply implements congruence for bitsums. That is, consider the following polynomials, with $m < \log_2 |\mathbb{F}|$:

$$Y - \textstyle\sum_{i=1}^{m} 2^{i-1} X_i \qquad\qquad Y' - \textstyle\sum_{i=1}^{m} 2^{i-1} X_i'$$

If all $X_i$ and $X_{i'}'$ are known to have value zero or one (because $X_i^2 - X_i$ is in some $\langle B_j \rangle$) and $Y$ and $Y'$ are known to be equal ($Y - Y'$ is in some $\langle B_j \rangle$), then it propagates $X_i - X_i'$ for all $i$. Similarly, if $Y$ is known to be a constant $c$ ($Y - c$ is in some $\langle B_j \rangle$), then each $X_i$ must be equal to the $j^{\text{th}}$ bit of $c$ as an unsigned integer. Soundness for extraProp follows from bit representation uniqueness.

**Inter-Basis interactions.** SplitGB treats each $B_i$ as a source of polynomials that might be added to other $B_j$. It does not use $\langle B_i \rangle$ as the source; this would be sound, but enumerating the infinite set $\langle B_i \rangle$ is impossible. The natural question is whether inter-basis propagation within SplitGB is nevertheless complete, that is, whether all polynomials $p \in \langle B_i \rangle$ that are admissible to $B_j$ are in the ideal generated by the polynomials actually added to $B_j$.

We have both positive and negative results for BitSplit: Lemma 1 shows that propagation from the sparse basis to the linear basis **is** complete. The proof is in Appendix E. Example 1 shows that propagation from the linear basis to the sparse basis **is not** complete. There is a natural way to fix this: enumerate each

variable pair $X, Y$, and propagate $X - Y$ to the sparse basis if $X - Y$ is in the ideal generated by the linear basis. However, our experiments (Sec. 5) show that this doesn't empirically improve solver performance for our benchmarks.

**Lemma 1.** *Let $B$ be a Gröbner basis under a graded order (a degree compatible order, i.e., for all monomials $p$, $q$, $\mathsf{deg}(p) < \mathsf{deg}(q) \implies p < q$); then, every linear $p \in \langle B \rangle$ is in the ideal generated by the linear elements of $B$.*

*Example 1.* Consider $\mathbb{F}_5[W, X, Y, Z]$ in grevlex order. Then $B_1 = \{W - X - Y + Z, Y - Z\}$ is a GB. The only polynomial in $B_1$ that is admissible to the sparse basis is $Y - Z$. Now consider $W - X$. It is in $\langle B_1 \rangle$ (it is the sum of $B_1$'s elements) and it is admissible to the sparse basis. However, it is not in $\langle Y - Z \rangle$; i.e., it **is not** generated by the subset of $B_1$ that is admissible to the sparse basis.

**Connections.** In some respects, our $\mathbb{F}$-solver resembles two prior SMT ideas: theory combination and portfolio solving with clause sharing. As in theory combination [6], we reduce a problem (a system of field equations) to sub-problems (subsets of the original system) that are handled by loosely-coupled sub-solvers (bases and propagators), each using different reasoning. As in portfolio solving with clause sharing [45, 63], each sub-solver derives lemmas in a common language (not clauses, but polynomials) that they share with one another. Our work also resembles a prior combination of algebraic and propositional reasoning for preprocessing Boolean formulas by sharing $\mathbb{F}_2$ equations between algebraic and propositional modules [14]. However, our focus is on solving equations in a very large finite field with constraints of different structure.

**Efficiency.** In the worst case, BitSplit builds a GB for the full system (similar to Monolithic). A GB for degree-$d$ polynomials in $n$ variables can have size $d^{2^n}$ [46], so the worst-case complexity of BitSplit (and Monolithic) is doubly exponential.

However, in the next section we will see that BitSplit is efficient on a number of problems of practical interest. For these problems it improves exponentially on Monolithic. Here, we give intuition for the source of the advantage. Consider a bitsum-heavy determinism problem. As discussed in Section 3, computing a full GB is hard, so Monolithic performs poorly. However, BitSplit can use extraProp to reason about the uniqueness of the bit-splitting and use its split GB to reason about other parts of the system. This might allow it to refute the system of equations without ever directly computing a GB for the full system.

## 5 Experiments

Now we present our experiments, which answer three empirical questions:

1. How does BitSplit perform when solving bitsum-heavy determinism queries? *(Exponentially better than the prior state of the art.)*
2. How does BitSplit perform when solving other queries? *(Similar to the prior state of the art.)*
3. How do BitSplit's components impact its performance? *(Propagation is key.)*

| Family | # | Description |
|---|---|---|
| CirC-D | 640 | Determinism for CirC $\mathbb{F}$-blaster rules of bitwidth $\leq 32$ (Sec. 6) |
| Seq | 100 | Determinism for sequenced bit-splits (App. F) |
| QED$^2$ | 100 | Determinism for circomlib, generated by QED$^2$ [54] |
| CirC-S | 100 | Soundness for CirC $\mathbb{F}$-blaster rules of bitwidth $\leq 4$ [53] |
| TV | 100 | Translation validation for ZKP compilers on boolean programs [52] |
| Small | 100 | Randomly generated with a small field: $\lvert\mathbb{F}\rvert \leq 211$ [39] |

Table 4: Our benchmark families. QED$^2$ [54], Small [39], TV [52], and CirC-S [53] are from prior work. CirC-D is a set of large determinism benchmarks based on prior work [53]; see Section 6. Seq is a set of determinism benchmarks for computations that perform a sequence of bit-splits; see Appendix F.

We implement BitSplit in cvc5 [4] as a solver for the theory of finite fields. This includes preprocessing that identifies bitsums in larger polynomials and isolates them for use in BitSplit. Our test bed is a cluster with Intel Xeon E5-2637 v4 CPUs. Each run gets one CPU, 8GB memory, and a time limit of 300 seconds. After presenting the benchmarks, we compare BitSplit to prior SMT $\mathbb{F}$-solvers ffsat [39][6] and Monolithic [52], and we compare BitSplit to variants of itself.

### 5.1 Benchmarks

Table 4 shows our benchmarks, most of which concern the correctness of ZK libraries (circomlib [7]) and compilers (ZoKrates [23] and CirC [50]). There are six families. The CirC-D benchmarks verify the determinism of operator encoding rules in CirC, at bitwidths up to 32. As we discuss in the next section (Sec. 6), these benchmarks are important to CirC's correctness, but are hard to solve. The Seq benchmarks verify the determinism of constraint systems with sequences of bit-splits. We discuss them further in Appendix F. The QED$^2$ benchmarks are determinism queries for circomlib generated by QED$^2$ [54]. The CirC-S benchmarks are soundness tests for CirC's operator rules, at bitwidths up to 4 [53]. The TV benchmarks are translation validation queries for ZoKrates and CirC, as applied to boolean functions [52]. Finally, the Small benchmarks are random, small-field (i.e., $\lvert\mathbb{F}\rvert < 2^8$) benchmarks from the evaluation of ffsat [39]. To keep the benchmark set from being too big, all families from prior work are sampled at random from that work's benchmarks.

### 5.2 Comparison to prior solvers

First, we compare BitSplit against prior solvers Monolithic [52] and ffsat [39]. Table 5 shows the number of solved benchmarks by family and result. ffsat

[6] A time of our experiments, ffsat was a Sage-based Python tool for solving conjunctions of equations [36]. We wrapped it with a simple SMT-LIB parser that invokes ffsat if the query is sufficiently simple. Since then, ffsat has been re-implemented in Yices [22, 38]; future work should compare against that implementation.

| Solver | Solved | By Family | | | | | | By Result | |
|---|---|---|---|---|---|---|---|---|---|
| | | CirC-D | Seq | QED$^2$ | CirC-S | TV | Small | SAT | UNSAT |
| BitSplit | **969** | **582** | **100** | **59** | 92 | 70 | 66 | 88 | **881** |
| Monolithic | 475 | 191 | 13 | 38 | **94** | **72** | **67** | **90** | 385 |
| ffsat | 67 | 0 | 0 | 0 | 0 | 0 | **67** | 54 | 13 |

Table 5: Solved benchmarks, by family and result. BitSplit's gains are on determinism queries (the QED$^2$ and CirC-D families) and unsatisfiable benchmarks.

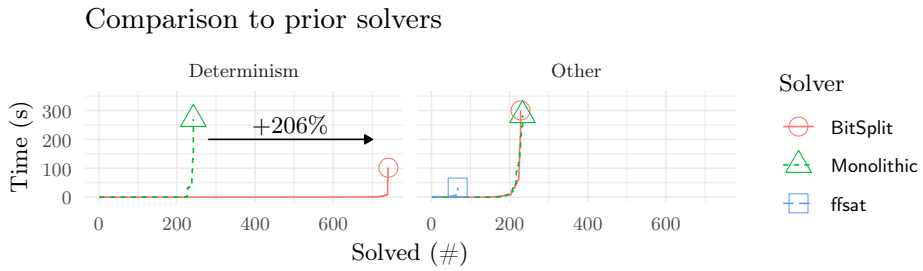


Fig. 5: On determinism benchmarks, BitSplit dominates Monolithic; on other benchmarks, they perform similarly.

is successful only when the field is small. BitSplit improves on Monolithic on families that test determinism (QED$^2$ and CirC-D) but suffers slightly on other benchmarks. BitSplit is slightly worse on SAT instances but better at UNSAT ones. Figure 5 presents the same results as cactus plots for the determinism families and the other families.

To better understand BitSplit's advantage, we focus on the CirC-D family. Each CirC-D benchmark tests the determinism of an operator rule at a specific bitwidth. We consider how the solve time scales with bitwidth. Figure 6 shows the results for arithmetic, shift, and comparison operators. Monolithic's solve time grows exponentially for all of these, while BitSplit's time is generally insignificant. BitSplit struggles only with division and remainder; verifying their determinism would require understanding that integer division is deterministic, as encoded in field constraints. We omit bitwise operators (e.g., `bvor`) from this experiment. Their operator rules assume that the input bit-vectors are *already represented as bits*, so their benchmarks do not include any bitsums. To summarize, BitSplit can verify many operators exponentially faster than Monolithic.

### 5.3 Comparison to variants

To better understand BitSplit, we compare it against six variants of itself:

- BS-LinFirst: make the linear basis (not the sparse basis) $B_1$
- BS-NoIntProp disable inter-basis propagation

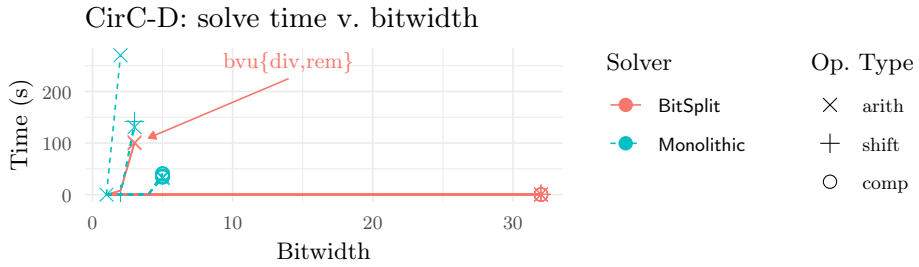


Fig. 6: Solve time for CirC-D benchmarks for different operators. Monolithic's solve time grows exponentially, while BitSplit's solve time usually does not.

| Solver | Solved | By Family | | | | | | By Result | |
|---|---|---|---|---|---|---|---|---|---|
| | | CirC-D | Seq | QED$^2$ | CirC-S | TV | Small | SAT | UNSAT |
| BitSplit | **969** | **582** | **100** | **59** | 92 | 70 | 66 | 88 | **881** |
| BS-LinFirst | 959 | 576 | **100** | 58 | 92 | 69 | 64 | 84 | 875 |
| BS-NoIntProp | 877 | 576 | 24 | 58 | 86 | 70 | 63 | 84 | 793 |
| BS-NoExtProp | 344 | 131 | 0 | 34 | 45 | 69 | 65 | 85 | 259 |
| BS-FullIntProp | 953 | 576 | 97 | 56 | 92 | 69 | 63 | 83 | 870 |
| BS-DenseProp | 898 | 580 | 33 | 58 | 92 | 70 | 65 | 85 | 813 |
| BS-QuadProp | 898 | 580 | 32 | **59** | 92 | 71 | 64 | 86 | 812 |
| Monolithic | 475 | 191 | 13 | 38 | **94** | **72** | **67** | **90** | 385 |

Table 6: BitSplit v. variants of itself. Weaker propagation (BS-NoExtProp, BS-NoIntProp) gives worse results, but other changes have less impact.

- BS-NoExtProp disable extraProp
- BS-FullIntProp: complete linear-to-sparse propagation (Sec. 4.3, fixes Ex. 1)
- BS-DenseProp for the sparse basis, use $\mathsf{admit}(p) = \mathsf{deg}(p) \leq 1 \wedge |\mathsf{vars}(p)| \leq 16$.
- BS-QuadProp for the linear basis, use $\mathsf{admit}(p) = \mathsf{deg}(p) \leq 2$.

Table 6 shows how many benchmarks each variant solves, with both BitSplit and Monolithic for comparison. First, changing the basis order (BS-LinFirst) has little effect. Second, disabling propagation (BS-NoIntProp and BS-NoExtProp) significantly hurts performance. Third, making inter-basis propagation complete (BS-FullIntProp) actually hurts performance slightly, perhaps because it takes quadratic time. Finally, defining admit more admissibly (BS-DenseProp and BS-QuadProp) makes little difference for many families, but significantly hurts performance on sequential bit-splits.

These results justify the key role that propagation plays in BitSplit. They also suggest that BitSplit would be a good choice for cvc5's default field solver.

# 6 Application

Prior work uses Monolithic to do bounded verification for a zero-knowledge proof (ZKP) compiler pass [53]. In this section, we improve their results using BitSplit. Thus, this section is a case study that shows the utility of BitSplit for a downstream verification task. Our improvement relies not just on a new solver (BitSplit), but also on a new verification strategy. First (§6.1), we give background on the verification task. Second (§6.2), we state our new strategy, prove it is correct, and show that it is more efficient—when using BitSplit.

## 6.1 Background on verifiable field-blasting

We consider the *finite field blaster* in a ZKP compiler: its responsibilities include encoding bit-vector operations as field equations [53]. At a high level, the field blaster is a collection of *encoding rules*. Each rule is a small algorithm that is specific to some operator (e.g., `bvadd`). It is given field variables that encode the operator's inputs according to some *encoding scheme*. A rule defines new variables, creates equations, and ultimately returns a field variable that encodes the output of the rule's operator.

As an example, we describe an encoding scheme for bit-vectors and a rule for bit-vector addition. The scheme encodes a length-$b$ bit-vector $x$ as a field variable $x'$ with value in $\{0, \ldots, 2^b - 1\} \subseteq \mathbb{F}$ (assuming $|\mathbb{F}| \gg 2^b$). If $x'$ and $x$ have the same (unsigned) integer value, we say that $\mathit{valid}(x', x)$ holds. Suppose our rule applies to the addition of $x$ and $y$, encoded as $x'$ and $y'$. Our rule defines the following field variables. First, for each $i \in \{1, \ldots, b+1\}$, it defines $z'_i$ to 1 if the $i^{\text{th}}$ bit of the integer sum of the unsigned values of $x'$ and $y'$ is one, and zero otherwise. Second, it defines $z' = \sum_{i=1}^{b} 2^{i-1} z'_i$. Then, it enforces these equations:

$$x' + y' = \textstyle\sum_{i=1}^{b+1} 2^{i-1} z'_i \quad \wedge \quad z' = \sum_{i=1}^{b} 2^{i-1} z'_i \quad \wedge \quad \bigwedge_{i=1}^{b+1} z'_b(z'_b - 1) = 0$$

Finally, it returns $z'$. Informally, the idea of this rule is to bit-decompose the sum $x' + y'$ and then use the bit-decomposition to reduce that sum modulo $2^b$. For example, if $b = 2$, $x' = 3$, and $y' = 1$, then the unique solution for the $z'_i$ is $z'_1 = 0, z'_2 = 0, z'_3 = 1$, and then $z'$ must be 0.

In general, an encoding rule for operator $o$ maps a sequence of input encodings (field variables) $\mathbf{e}$ to three outputs: $F$, $A$, and $e$.[7] Each field variable $e_i$ encodes some bit-vector variable $t_i$. The first output, $F = \{z_1 \mapsto s_1, \ldots z_\ell \mapsto s_\ell\}$, is a mapping that defines $\ell$ fresh field variables: $z_1, \ldots, z_\ell$. Variable $z_i$ is mapped to a term $s_i$ (in variables $\mathbf{e}$) that defines what value $z_i$ is intended to take. The second output, $A$, is conjunction of field equations in variables $\mathbf{e}$ and $\mathbf{z}$. The final output is $e$: a distinguished variable that encodes the rule's output $o(\mathbf{t})$.

[7] Actually, in prior work [53] and in our implementation, encodings are *type-tagged sequences* of field *terms*. In this paper we treat them as single variables to simplify the exposition. Generalization is straightforward, but notationally tedious.

Prior work defines *correctness* for encoding rules as the conjunction of two properties: *completeness* and *soundness*. If all rules are correct, then they constitute a correct $\mathbb{F}$-blaster [53]. Completeness says that if each $e_i$ validly encodes $t_i$ and the $z_i$ take the values prescribed by $F$, then $e$ validly encodes $o(\mathbf{t})$ and $A$ holds. That is, completeness requires the following formula to be valid:

$$((\bigwedge_i \mathit{valid}(e_i, t_i)) \implies (A \wedge \mathit{valid}(e, o(\mathbf{t})))\,[F]$$

Soundness says that if each $e_i$ validly encodes $t_i$ and $A$ holds, then $e$ validly encodes $o(\mathbf{t})$. That is, the following must be valid:

$$(A \wedge \bigwedge_i \mathit{valid}(e_i, t_i)) \implies \mathit{valid}(e, o(\mathbf{t}))$$

*Verifier performance* After fixing the sorts of the $t_i$ (e.g., to bit-vectors of size 4), one can encode soundness and completeness as SMT queries. This enables automatic, bounded verification: one checks these properties up to some input bitwidth bound $b$ using an SMT solver. However, the soundness query is especially challenging for the SMT solver. In prior work, some soundness queries for $b = 4$ could not be solved in 5 minutes with Monolithic. More generally, solving time grew exponentially with bit-width for most operators [53].

### 6.2 A new strategy for verifying operator rules

We propose a different strategy for automatically verifying operator rules. We define *determinism* for operator rules. It says that an operator rule applied to equal inputs should yield equal outputs. That is, if $(A, e)$ and $(A', e')$ are rule outputs for inputs $\mathbf{e}$ and $\mathbf{e}'$ respectively, then the following must be valid:

$$(A \wedge A' \wedge \mathbf{e} = \mathbf{e}') \implies e = e'$$

We prove the following theorem in Appendix G:

**Theorem 4.** *An operator rule that is deterministic and complete is also sound.*

Thus, to verify rule correctness, it suffices to verify completeness and **determinism**. This approach is promising because BitSplit is very effective on determinism queries (they were the CirC-D benchmarks in Section 5). So, a verification strategy comprises two choices: whether to prove soundness (S) or determinism (D) and whether to use BitSplit or Monolithic. In all cases, we prove completeness using exhaust (a specialized approach from prior work) [53]. For each strategy, we try to verify every bit-vector rule up to width 32. We limit SMT queries to 5 minutes each, using the same test bench as before.

Figure 7 shows verification time using different strategies. The best strategy is our new one. This approach verifies 66% more rule-bitwidth pairs than the next best strategy: proving soundness with Monolithic. More importantly, in our new strategy, verifying determinism (using BitSplit) is not the bottleneck: the bottleneck is proving completeness (using exhaust). Whereas, when proving soundness with Monolithic, Monolithic is the bottleneck. Further improvements will require new ideas for proving completeness.

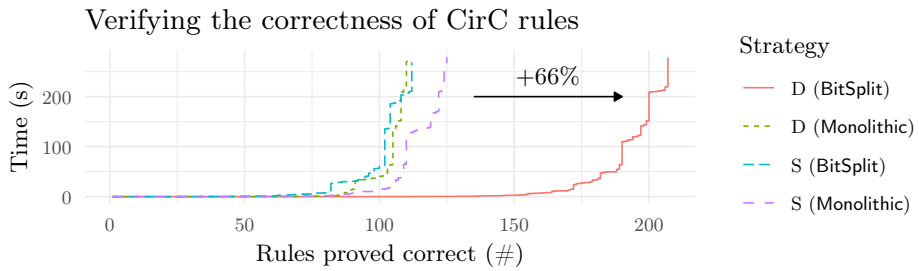


Fig. 7: The best way to verify that CirC rules are fully correct is to prove completeness using exhaust and prove determinism (D) using BitSplit.

## 7 Conclusion

We have presented a new approach for $\mathbb{F}$-solving in SMT. Our contributions are three-fold. First, we proposed an abstract decision procedure Split that avoids computing a full Gröbner basis. Second, we described an instantiation of it (BitSplit) that is highly effective for bitsum-heavy determinism queries. Third, we applied BitSplit to a problem in ZKP compiler verification.

There are many directions for future work. First, we believe other instantiations of Split (beyond BitSplit) might be useful, for example, by considering other kinds of propagations (extraProp) and other conditions under which propagation is allowed (admit). Second, Split makes very limited use of CDCL(T) features that are known to improve performance: it acts only once a full propositional assignment is available; it constructs no theory lemmas; and it propagates no literals. Third, in this paper, we focus on applications of the theory of finite fields to ZKPs. Finite fields should also be relevant to many other kinds of cryptosystems, including algebraic multi-party computation and those based on elliptic curves. We leave these opportunities to future work.

*Acknowledgements* We appreciate the help, support, and advice of Cesare Tinelli, Daniela Kaufmann, Haniel Barbosa, Mathias Preiner, Matthew Sotoudeh, Thomas Hader, the CAV reviewers, and all of the cvc5 developers.

This work was funded in part by NSF grant number 2110397 and the Stanford Center for Automated Reasoning.

# Bibliography


[1] 0xPARC. ZK bug tracker. https://github.com/0xPARC/zk-bug-tracker. Accessed 5 Sept 2023, via archive.org.

[2] B. Anderson and D. McGrew. TLS beyond the browser: Combining end host and network data to understand application behavior. In *IMC*, 2019.

[3] D. Archer, A. O'Hara, R. Issa, and S. Strauss. Sharing sensitive department of education data across organizational boundaries using secure multiparty computation, 2021.

[4] 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.

[5] R. Barlow. Computational thinking breaks a logjam. https://www.bu.edu/cise/computational-thinking-breaks-a-logjam/, 2015.

[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] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[8] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, et al. Secure multiparty computation goes live. In *FC*, 2009.

[9] D. Braun, N. Magaud, and P. Schreck. Formalizing some "small" finite models of projective geometry in coq. In *International Conference on Artificial Intelligence and Symbolic Computation*, 2018.

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

[11] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE S&P*, 2018.

[12] S. Chaliasos, J. Ernstberger, D. Theodore, D. Wong, M. Jahanara, and B. Livshits. Sok: What don't we know? understanding security vulnerabilities in snarks, 2024. https://arxiv.org/abs/2402.15293.

[13] 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.

[14] D. Choo, M. Soos, K. M. A. Chai, and K. S. Meel. Bosphorus: Bridging anf and cnf solvers. In *DATE*. IEEE, 2019.

[15] 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.

[16] C. Cohen. Pragmatic quotient types in coq. In *ITP*, 2013.

[17] D. Cox, J. Little, and D. OShea. *Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra*. Springer Science & Business Media, 2013.

[18] CVE-2014-3570. https://nvd.nist.gov/vuln/detail/CVE-2014-3570.

[19] CVE-2017-3732. https://nvd.nist.gov/vuln/detail/CVE-2017-3732.
[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] D. S. Dummit and R. M. Foote. *Abstract algebra*, volume 3. Wiley Hoboken, 2004.
[22] B. Dutertre. Yices 2.2. In *CAV*, 2014.
[23] J. Eberhardt and S. Tai. ZoKrates—scalable privacy-preserving off-chain computations. In *IEEE Blockchain*, 2018.
[24] H. B. Enderton. *A mathematical introduction to logic*. Elsevier, 2001.
[25] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Systematic generation of fast elliptic curve cryptography implementations. Technical report, MIT, 2018.
[26] 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.
[27] J.-C. Faugere, P. Gianni, D. Lazard, and T. Mora. Efficient computation of zero-dimensional gröbner bases by change of ordering. *Journal of Symbolic Computation*, 16(4), 1993.
[28] Y. Finance. Monero quote. https://finance.yahoo.com/quote/XMR-USD/, 2023. Accessed: 13 October 2023.
[29] Y. Finance. Zcash quote. https://finance.yahoo.com/quote/ZEC-USD/, 2023. Accessed: 13 October 2023.
[30] C. Fournet, C. Keller, and V. Laporte. A certified compiler for verifiable computing. In *CSF*, 2016.
[31] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge, 2019. https://ia.cr/2019/953.
[32] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, et al. A machine-checked proof of the odd order theorem. In *ITP*, pages 163–179, 2013.
[33] G.-M. Greuel, G. Pfister, and H. Schönemann. Singular—a computer algebra system for polynomial computations. In *Symbolic computation and automated reasoning*, pages 227–233. AK Peters/CRC Press, 2001.
[34] J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, 2016.
[35] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish. Zero-knowledge middleboxes. In *USENIX Security*, 2022.
[36] T. Hader. Ffsat. https://github.com/Ovascos/ffsat, commit 67fecde.
[37] T. Hader. Non-linear SMT-reasoning over finite fields, 2022. MS Thesis (TU Wein).
[38] T. Hader, D. Kaufmann, A. Irfan, S. Graham-Lengrand, and L. Kovács. Mcsat-based finite field reasoning in the yices2 smt solver, 2024.
[39] T. Hader, D. Kaufmann, and L. Kovács. SMT solving over finite field arithmetic. In *LPAR*, 2023.
[40] T. Hader and L. Kovács. Non-linear SMT-reasoning over finite fields. In *SMT*, 2022. Extended Abstract.
[41] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. Zcash protocol specification. https://raw.githubusercontent.com/zcash/zips/master/protocol/protocol.pdf, 2013.
[42] V. Komendantsky, A. Konovalov, and S. Linton. View of computer algebra data from coq. In *International Conference on Intelligent Computer Mathematics*, 2011.

[43] P. Kotzias, A. Razaghpanah, J. Amann, K. G. Paterson, N. Vallina-Rodriguez, and J. Caballero. Coming of age: A longitudinal study of TLS deployment. In *IMC*, 2018.

[44] 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.

[45] M. Marescotti, A. E. J. Hyvärinen, and N. Sharygina. Clause sharing and partitioning for cloud-based SMT solving. In C. Artho, A. Legay, and D. Peled, editors, *Automated Technology for Verification and Analysis*, pages 428–443, Cham, 2016. Springer International Publishing.

[46] E. W. Mayr and A. R. Meyer. The complexity of the word problems for commutative semigroups and polynomial ideals. *Advances in mathematics*, 46(3):305–329, 1982.

[47] Monero technical specs. https://monerodocs.org/technical-specs/, 2022.

[48] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(T). *J. ACM*, 2006.

[49] OpenSSL bug 1953. https://www.mail-archive.com/openssl-dev@openssl.org/msg23869.html.

[50] A. Ozdemir, F. Brown, and R. S. Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In *IEEE S&P*, 2022.

[51] A. Ozdemir, G. Kremer, C. Tinelli, and C. Barrett. Satisfiability modulo finite fields, 2022. In submission, Preprint at https://ia.cr/2023/091.

[52] A. Ozdemir, G. Kremer, C. Tinelli, and C. Barrett. Satisfiability modulo finite fields. In *CAV*, 2023.

[53] A. Ozdemir, R. S. Wahby, F. Brown, and C. Barrett. Bounded verification for finite-field-blasting. In *CAV*, 2023.

[54] 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.

[55] J. Philipoom. *Correct-by-construction finite field arithmetic in Coq*. PhD thesis, Massachusetts Institute of Technology, 2018.

[56] P. Schwabe, B. Viguier, T. Weerwag, and F. Wiedijk. A coq proof of the correctness of x25519 in tweetnacl. In *CSF*, 2021.

[57] F. H. Soureshjani, M. Hall-Andersen, M. Jahanara, J. Kam, J. Gorzny, and M. Ahmadvand. Automated analysis of halo2 circuits, 2023. https://ia.cr/2023/1051.

[58] Tornado.cash got hacked. by us. https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8, 2019. Accessed: 13 October 2023.

[59] D. Wang. *Elimination methods*. Springer Science & Business Media, 2001.

[60] F. Wang. Ecne: Automated verification of zk circuits, 2022. https://0xparc.org/blog/ecne.

[61] 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.

[62] 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.

[63] C. M. Wintersteiger, Y. Hamadi, and L. de Moura. A concurrent portfolio approach to SMT solving. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, pages 715–720, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[64] Zcash counterfeiting vulnerability successfully remediated. https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/, 2019. Accessed: 13 October 2023.

# A Additional Background

In this appendix, we provide additional algebraic background that our proofs rely on.

*Monomial Orderings* A *monomial* is a product of variable powers: $\prod_{i=1}^{n} X_i^{e_i}$. A polynomial is a sum of monomials, scaled by *coefficients* in $\mathbb{F}$.

A monomial ordering is a total ordering on monomials respecting the following: for all monomials $f, g, h$, $f < g$ implies $fh < gh$. The lexicographic order monomials $\prod_{i=1}^{n} X_i^{e_i}$ lexicographically by tuple $(e_1, \ldots, e_n)$. The graded lexicographic order is lexicographic by tuple $(\sum_i e_i, e_1, \ldots, e_n)$. The graded reverse lexicographic (*grevlex*) order is lexicographic by tuple $(\sum_i e_i, -e_n, \ldots, -e_1)$. The efficiency of GB construction can depend on the monomial order; empirically, grevlex is usually the most efficient [27]. Any order in which $\mathsf{deg}(f) < \mathsf{deg}(g)$ implies $f < g$ is called a *graded* order.

In a polynomial, the largest monomial (w.r.t. a fixed monomial order) is called the *leading monomial* and denoted $\mathsf{lm}(f)$. Its term is $\mathsf{lt}(f)$.

*Reduction and Gröbner bases* Let $f, g$ be polynomials. We define the reduction relation $\to$ as follows: if some term $t$ of $f$ is equal to $h \cdot \mathsf{lm}(g)$ for monomial $h$, then $f \to_g (f - hg)$. Note that in this process the term $t$ is replaced by terms strictly smaller than $t$. For set $G$, the relation $\to_G$ is the union of $\to_g$ for $g \in G$ (recall: a relation is a set of pairs). If there is no $h$ (except $f$) such that $f \to_g h$ (respectively: $f \to_G h$), then we write $f \not\to_g$ (respectively: $f \not\to_G$). The transitive closure of $\to_g$ (respectively $\to_G$) is denoted $\to_g^*$ (respectively $\to_G^*$). If $f \to_G h$ and $h \not\to_G$, we say $f \twoheadrightarrow_G h$. Note that $\twoheadrightarrow_G$ is not necessarily a function. There is a polytime algorithm for computing **an** $h$ such that $f \twoheadrightarrow_G h$, from $f$ and $G$. The algorithm is called $\mathsf{Reduce}$: $h = \mathsf{Reduce}(f, G)$.

A Gröbner basis is a set $G$ such that $f \twoheadrightarrow_G 0$ if an only if $f \in \langle G \rangle$. For a Gröbner basis $G$, $\twoheadrightarrow_G$ is a function, so $\mathsf{Reduce}$ is deterministic.

The *ascending chain condition* (ACC) states that there is no infinite sequence of ideals $I_1, I_2, \ldots$ in a polynomial ring with field coefficients, such that $I_i \subsetneq I_{i+1}$ for all $i$. The ACC is used to prove termination for various algorithms that compute Gröbner bases.

# B Computing Bitsum Usage in Real World Projects

In this section we describe how we derived two key statistics in Section 3 that 1) over 98 % of Circom repositories used bitsum generating circuits like `Num2Bits`, and 2) approximately 64% of the variables in Circomlib's Sha256 circuit appear in bitsums.

## B.1 Percentage of Repositories using `Num2Bits`

We first wrote a crawler to gather GitHub repositories with Circom circuits. Our crawler downloaded public repositories which either had the language tags of Circom or Solidity

(since Circom is most frequently used within Solidity projects) in order of GitHub stars. We then scanned each repository to check if it contained files with the '.circom' extension. This yielded a total of 655 repositories.

To determine if a repository used a bitsum generating circuit, we manually identified a set of primitive bitsum generating circuits which included all the circuits in circomlib/circuits/bitify.circom which contained `Num2Bits`, `Num2Bits_strict`, `Bits2Num`, `Bits2Num_strict`, `Num2BitsNeg` as well as the circuits `BinSum` and `BinSub` in circomlib/circuits/binsum.circom and circomlib/circuits/binsub.circom respectively. Next, we scanned each uncommented line of the circom files in the implementation and checked if it matched the following pattern: `component [a-Z\[\]0-9]+ = P*` where P is the name of one of the primitive bitsum generating circuits. If this pattern was matched, then we marked the repository as using a bitsum generating circuit and marked the circuit containing the line as being bitsum generating.

We also manually examined some of the circuits which did not use `Num2Bits`. Many of these, like `circom-monolith`, implement their own version of bit-splitting. This indicates that the usage of bitsum generating circuits is even higher than the 98% that we estimate.

### B.2 Percentage of Bitsum Variables in Sha256

We first compiled the Sha256 circuit into R1CS, an intermediate representation of the constraints where every equation is of the form $A(x) * B(x) - C(x) = 0$. To do so, we first concretely instantiated the circuit by declaring a main circuit as follows:

```
component main {public [in]} = Sha256(2);
```

We then compiled the circuit into R1CS with optimization level O1 and then statically analyzed each equation in the circuit to see if $A$, $B$, or $C$ matched the pattern $\sum_{i=0}^{n} 2^i * x_i$. If it did, then we marked all the variables in $A$, $B$, and $C$ as appearing in a bitsum constraint. After counting all such variables, we divided that number by the total number of variables in the circuit to obtain the percentage.

## C Proof of Theorem 1

First, we show correctness. The fact that $\langle B_i \rangle \supseteq \langle G_i \rangle$ follows from the fact that each $B_i$ is initially $G_i$ and only changes by being replaced by a basis for an ideal that contains the previous $B_i$. Thus, further, we have that $\langle \mathbf{B} \rangle \supseteq \langle \mathbf{G} \rangle$. To show the other direction, observe that each $p$ added to some $B_i$ is from another $\langle B_j \rangle \subseteq \langle \mathbf{B} \rangle$ or from $\mathsf{extraProp}(\mathbf{B}) \subseteq \langle \mathbf{B} \rangle$. Thus, $\langle \mathbf{B} \rangle$ does not grow either, and thus equals $\langle \mathbf{G} \rangle$ upon function exit.

Second, we show termination. Suppose non-termination, towards contradiction. Let $G_i^{(j)}$ be $G_i$ at the start of loop iteration $j$; let $B_i^{(j)}$ denote $B_i$ before line 3. Non-termination requires that for all $j \in \mathbb{N}$, there is some $p$ such that for some $i \in [k]$, $p \notin B_i^{(j)}$; therefor, $\langle B_i^{(j+1)} \rangle \supsetneq \langle B_i^{(j)} \rangle$. Since $k$ is finite, then there is some $i \in [k]$ such that the above holds for infinitely many $j$. That is, the sequence $(\langle B_i^{(j)} \rangle)_{j \in \mathbb{N}}$ does not stabilize. But, this violates the ascending chain condition for polynomial rings.

# D Proof of Theorems 2 and 3

## D.1 Proof of Theorem 2

The proof of Theorem 2 is somewhat involved. The reason for this is that SplitZeroExtend can return a number of different kinds of values, and our proof relies on lemmas that express when those different returns can happen.

First, we recall the following lemma about ApplyRule: for any GB $B$ and any partial map $M$,

$$\mathcal{V}(\langle B\rangle) = \bigcup_{(X\mapsto r)\in\mathsf{ApplyRule}(B,M)} \mathcal{V}(\langle B\cup\{X-r\}\rangle)$$

This lemma is proved in Appendix B of [52]. We will use it later. First, we prove 4 lemmas about SplitZeroExtend.

First, SplitZeroExtend terminates This follows from observing that if $|M| \geq |\mathbf{X}|$, then SplitZeroExtend does not recurse, and that if SplitZeroExtend does recurse, then $|M|$ increases.

Second, if $\mathsf{SplitZeroExtend}(\mathbf{B}, G, M)$ returns $p$, then $p \in G \setminus \langle B_1\rangle$. We prove this with an induction over the stack depth when SplitZeroExtend first returns with $p$. In the base case, SplitZeroExtend returns on line 7, and the lemma holds immediately because of the if condition. In the recursive case, the outermost SplitZeroExtend call is recursive, but the recursion is with $\mathbf{B}' = \mathsf{SplitGB}((B_i\cup\{X-z\})_{i=1}^k)$. Thus, $\langle B_1'\rangle \supseteq \langle B_1\rangle$ (in part, through Theorem 1). Thus, because of the recursive hypothesis, we have $p \in G$ and $p \notin \langle B_1'\rangle \supseteq \langle B_1\rangle$. Thus, $p \in G \setminus \langle B_1\rangle$, as desired.

Third, if SplitZeroExtend returns $M$, then $M \in \mathcal{V}(\langle\mathbf{B}\rangle)$. First, we state a technical sub-lemma: in every all to SplitZeroExtend, for every $X_i \mapsto z_i$ in $M$, any every $B_i$, $X_i - z_i \in \langle B_i\rangle$. Let $\mathbf{B}'$ be $\mathbf{B}$ in the call to SplitZeroExtend where the return on line 9 occurs, and let $\mathbf{B}$ be the original $\mathbf{B}$. It suffices to show that $M \in \mathcal{V}(\langle\mathbf{B}_j\rangle)$ for all $j$. Suppose (towards contradiction) that $M$ were not a zero of some $p \in \langle B_j\rangle$; then 1 would be contained in $\langle p, X_1 - z_1, \ldots\rangle$, which is itself contained in $\langle B_j\rangle$. But, this contradicts the if condition on line 4. Thus, $M$ must be a zero of every $p \in \langle B_j\rangle$, for all $j$.

Fourth, if SplitZeroExtend returns $\bot$, then $\mathcal{V}(\langle\mathbf{B}\rangle)$ is empty. We prove this by induction on SplitZeroExtend's recursion. In the base case, $1 \in \langle B_i\rangle$ for some $i$, so $1 \in \langle\mathbf{B}\rangle$ too, so $\mathcal{V}(\langle\mathbf{B}\rangle)$ is empty. In the recursive case, SplitZeroExtend returns $\bot$ only if every recursion does. Thus, by the induction hypothesis, $\mathcal{V}(\langle\cup_i(\{X-z\}\cup B_i)\rangle)$ is empty for each $(X\mapsto z) \in \mathsf{ApplyRule}(B_1, M)$. Using this fact and the completeness of ApplyRule (stated earlier), we have

$$\begin{aligned}
\mathcal{V}(\langle\mathbf{B}\rangle) &= \mathcal{V}(\langle B_2,\ldots\rangle) \cap \mathcal{V}(\langle B_1\rangle) \\
&= \mathcal{V}(\langle B_2,\ldots\rangle) \cap \left(\bigcup_{(X\mapsto z)\in\mathsf{ApplyRule}(B_1,M)} \mathcal{V}(\langle\{X-z\}\cup B_1\rangle)\right) \\
&= \bigcup_{(X\mapsto z)\in\mathsf{ApplyRule}(B_1,M)} \mathcal{V}(\langle\{X-z\}\cup B_1\cup B_2\cup\cdots\rangle) \\
&= \bigcup_{(X\mapsto z)\in\mathsf{ApplyRule}(B_1,M)} \mathcal{V}(\langle\cup_i(\{X-z\}\cup B_i)\rangle) \\
&= \emptyset
\end{aligned}$$

This completes the induction.

Now, we prove a lemma about SplitFindZero itself. Let $\mathbf{B}^{(j)}$ be the basis after iteration $j$. Since each $p^{(j)} \notin \langle B_1^{(j)} \rangle$ (our previous lemma about conflicts from SplitZeroExtend), we have

$$\langle B_1^{(1)} \rangle \subsetneq \langle B_1^{(2)} \rangle \subsetneq \cdots$$

Moreover, since each $p \in \cup_i B_i$, every $\langle B_1^{(j)} \rangle$ is contained in $\langle \mathbf{G} \rangle$.

Now, we prove that SplitFindZero is correct and terminating. To prove that SplitFindZero terminates, observe that the ascending chain condition for polynomial rings implies that the above ideal chain is finite. Thus, SplitFindZero's loop is bounded, and the procedure terminates.

Now, we prove correctness. Consider when SplitFindZero returns $\bot$. Then some SplitZeroExtend$(\mathbf{B}^{(j)}, \cup_i B_i, \emptyset)$ returned bottom, so we have that $\mathcal{V}(\langle \mathbf{B}^{(j)} \rangle)$ is empty. Since $\langle \mathbf{B} \rangle$ contains $\langle \mathbf{B}^{(j)} \rangle$, $\mathcal{V}(\langle \mathbf{B} \rangle)$ is empty too. So, SplitFindZero is correct in this case. Now, consider when SplitFindZero returns $M$. Then some SplitZeroExtend$(\mathbf{B}^{(j)}, \cup_i B_i, \emptyset)$ returned $M$, so we have that $M \in \mathcal{V}(\langle \mathbf{B}^{(j)} \rangle)$. Then, $M$ is also in $\mathcal{V}(\langle \mathbf{B} \rangle)$. So, SplitFindZero is also correct in this case.

### D.2 Proof of Theorem 3

Now, we prove Theorem 3: the correctness and termination of Split (Fig. 4b). Termination follows immediately from the termination of SplitGB (Thm. 1) and SplitFindZero (Thm. 2). To see correctness, observe that since init is covering (Def. 2), $\cup_i G_i = G$. Thus, $\langle \mathbf{G} \rangle = \langle G \rangle$. Then, the correctness of SplitGB guarantees that $\langle \mathbf{B} \rangle = \langle \mathbf{G} \rangle$. Then, if Split exits early because for some $i$, $1 \in \langle B_i \rangle$, then since $\mathcal{V}(\langle G \rangle) \subset \mathcal{V}(\langle B_i \rangle) \subseteq \emptyset$, Split is correct. Otherwise, if Split calls SplitFindZero, then it returns $M \in \mathcal{V}(\langle \mathbf{B} \rangle)$ iff the latter is non-empty. And that holds iff $\mathcal{V}(\langle G \rangle)$ is non-empty (Thm. 2). Thus, Split is correct in this case too.

## E Proof of Lemma 1

As $B$ is a Gröbner basis and $p \in \langle B \rangle$, there is a sequence of reductions

$$p \to_{g_{i_1}} p - h_1 g_{i_1} \to_{g_{i_2}} p - h_1 g_{i_1} - h_2 g_{i_2} \to_{g_{i_3}} \cdots \to_{g_{i_k}} p - h_1 g_{i_1} - \ldots - h_k g_{i_k} = 0.$$

Each term of $p$ has degree 1 ($p$ is linear) and at each reduction steps terms are replaced by smaller terms with respected to a graded order, i.e., by terms whose degree is $\leq$ the degree of the term they are replacing. So all intermediate reduced steps are linear as well. Moreover as $lt(g_{i_j})$ divides the replaced term of degree 1, it has degree 1 and hence each $g_{i_j}$ is linear, as we have a graded order. $p - h_1 g_{i_1} - \ldots - h_k g_{i_k} = 0$ implies that $p$ is in the ideal generated by the linear elements in the Gröbner basis $B$.

## F The Seq Benchmark Family

In this appendix, we describe the Seq benchmark family. We created this family to stress-test a solver's ability to reason about determinism in *complex* bitsum-heavy constraints systems. Thus, it provides a counterpoint to the QED$^2$ and CirC-D families, which mostly test library functions and compiler lowering rules that are individually rather simple.

The family is parameterized by a bit count $b$ and a bitsum count $n$. Essentially, the system tests the determinism of $n$ iterations of splitting a value into its low and high bits, and then multiplying those two parts together. Its variables are:

$$X_0, \ldots, X_n, (B_{1,1}, \ldots, B_{1,b}), \ldots (B_{n,1}, \ldots, B_{n,b})$$

and primed copies of all of these. The equations are

$$\begin{aligned}
& X_0 = X_0' \\
\wedge\ & \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{b} (B_{i,j}^2 = B_{i,j} \wedge B_{i,j}'^2 = B_{i,j}') \\
\wedge\ & \bigwedge_{i=1}^{n} X_{i-1} = \sum_{j=1}^{b} 2^{j-1} B_{i,j} \wedge X_{i-1}' = \sum_{j=1}^{b} 2^{j-1} B_{i,j}' \\
\wedge\ & \bigwedge_{i=1}^{n} X_i = \left( \sum_{j=1}^{\lfloor b/2 \rfloor} 2^{j-1} B_{i,j} \right) \left( \sum_{j=\lfloor b/2 \rfloor + 1}^{n} 2^{j-1} B_{i,j} \right) \\
\wedge\ & \bigwedge_{i=1}^{n} X_i' = \left( \sum_{j=1}^{\lfloor b/2 \rfloor} 2^{j-1} B_{i,j}' \right) \left( \sum_{j=\lfloor b/2 \rfloor + 1}^{n} 2^{j-1} B_{i,j}' \right) \\
\wedge\ & X_n \neq X_n'
\end{aligned}$$

The first line equates the inputs. The second line bit-constraints all $B_{i,j}$ and $B_{i,j}'$. The third line bit-decomposes each $X_{i-1}$ into the $B_{i,j}$ (and likewise for the primed variables). The fourth line sets $X_i$ to the product of $X_{i-1}$ low and high parts (expressed in terms of the bit decomposition). The fifth line does the same for the primed variables. Finally, the last line asserts the disequality of the final $X_n$ and $X_n'$.

Our benchmark family instantiates the above system for $b \in \{2, \ldots, 21\}$ and $n \in \{1, \ldots, 5\}$, giving one hundred benchmarks in total.

## G Proof of Theorem 4

In this appendix, we prove Theorem 4: if an operator rule is deterministic and complete, then it is sound (these properties are defined in Section 6).

Before we begin, we refine the notation for our properties. Let $t$ be a term in variables $x$; we write $t(x)$ to emphasize that $t$ depends on $x$. Let $z_e$ be the variables in $\mathbf{e}$, let $z_t$ be the variables in $\mathbf{t}$, and let $z_F$ be the fresh variables introduced in $F$. With these variables made explicit, our properties are:

- Sound: $\forall z_e, z_t, z_F, (A \wedge \bigwedge_i \mathit{valid}(e_i, t_i)) \to \mathit{valid}(e, o(\mathbf{t}))$
- Complete $\forall z_e, z_t, \left((\bigwedge_i \mathit{valid}(e_i, t_i)) \to (A \wedge \mathit{valid}(e, o(\mathbf{t})))\right) [F]$
- Deterministic $\forall z_e, z_F, z_e', z_F', (A \wedge A' \wedge \bigwedge_i e_i = e_i') \implies e = e'$

Suppose a rule is deterministic and complete. We will show it is sound. Fix $\mathbf{t}$ and $\mathbf{e}$, and let $F$, $A$, and $e$ be the output of the operator rule. Fix $z_e, z_t, z_F$ and assume

$$A(z_e, z_F) \tag{5}$$

and

$$\bigwedge_i \mathit{valid}(e_i(z_e), t_i(z_t)) \tag{6}$$

hold. Our goal is to show

$$valid(e(z_e, z_F), o(\mathbf{t}(z_t))) \tag{7}$$

After instantiating the completeness property with $z_e$ and $z_t$, the property's conditions are satisfied by (6), so we have that:

$$A[F](z_e, z_t) \tag{8}$$

and

$$valid(e[F](z_e, z_t), o(\mathbf{t})) \tag{9}$$

hold. Now, we instantiate determinism with $z_e$ and $z_F$ set to themselves, $z'_e$ set to $z_e$, and $z'_F$ set to $F(z_e, z_t)$. The property's conditions are met by (5) and (8), so we have: $e[F](z_e, z_t) = e(z_e, z_F)$. We substitute this equality into (9) to show our goal, (7).