Proof Production for Satisfiability Modulo Finite Fields with Proof Checking in Pacheck and Lean

Pedro Saccomani[1], Abdalrhman Mohamed[2], Elizaveta Pertseva[2], Daniela Kaufmann[3], Cesare Tinelli[4], Clark Barrett[2], Haniel Barbosa[1]

[1]Universidade Federal de Minas Gerais, Belo Horizonte, Brazil, ✉{pedro.saccomani, hbarbosa}@dcc.ufmg.br
[2]Stanford University, Stanford, USA ✉{abdal, pertseva, barrettc}@stanford.edu
[3]TU Wien, Vienna, Austria, ✉daniela.kaufmann@tuwien.ac.at
[4]The University of Iowa, Iowa City, USA, ✉cesare-tinelli@uiowa.edu

***Abstract*—Finite field reasoning techniques in Satisfiability Modulo Theories (SMT) have gained great attention recently due to their application in the verification of cryptosystems. However, while the production of independently checkable proofs to ensure the correctness of solver implementations has become increasingly relevant in modern SMT solving, no state-of-the-art SMT solver that supports reasoning over finite fields currently produces proofs. We address this limitation by proposing a proof calculus that captures the decision procedure for finite field reasoning employed by the SMT solver CVC5 and by instrumenting the solver to generate proofs within this calculus. Furthermore, we introduce an extension of the Practical Algebraic Calculus (PAC) that mirrors our proposed system, along with a corresponding enhancement to its proof checker, PACHECK, enabling it to verify the generated proofs. Finally, we formalize the proposed calculus in the Lean proof assistant and extend the LEAN-SMT tactic to use it. This enables formally verified proof checking and provides proof automation for Lean formalizations involving finite fields via SMT solving. An evaluation over benchmarks from various zero knowledge proof compilers shows that supporting proof production for finite fields has an overhead on par with proof production for other theories supported by CVC5. All proofs can be checked almost instantaneously with PACHECK; LEAN-SMT, while slower, can verify over 60% of them.**

I. Introduction

Finite fields are central to many cryptosystems [6], [14] due to their usage in cryptographic operations. For example, Zero-Knowledge Proof (ZKP) protocols are used for building systems that maintain privacy [11]. ZKP proofs can be expressed by a set of finite field equations. Since this representation is inconvenient for humans, ZKPs are usually written in a higher-level language and translated by a compiler. A bug in the implementation of these compilers could be catastrophic: it could alter the intended statement, causing the compiler to produce a valid proof for a false claim. Motivated by this problem, and the verification of other cryptosystems, formal verification of finite-field based systems has recently gained great research interest, with SMT solvers used as an essential component [22]. As a result, decision procedures for a theory of finite fields have been developed and implemented in the state-of-the-art SMT solvers CVC5 [20], [21] and Yices2 [12]. An important functionality in SMT solvers is to produce certificates that can be externally verified. For unsatisfiability results, generally corresponding to the validity of verification goals, the certificates correspond to refutation proofs. Proof certificates increase the confidence in a solver's output for allowing independent verification by third-party proof checking tools. Furthermore, they can be used to increase automation in proof assistants, allowing them to discharge certain goals via external solvers provided that those solvers can produce justifications that can be internally reconstructed into proofs accepted by the proof assistant's kernel [10], [16], [19].

Unfortunately, neither CVC5 nor Yices2 can produce proofs for their finite fields reasoning. This hinders their trustworthiness in certain domains and also limits their usage for proof automation in systems that could leverage SMT solvers for finite field reasoning.

Contributions: In this work, we instrument CVC5 to extend its already extensive capabilities to produce proof certificates to producing proofs in the theory of finite fields. CVC5's subsolver for the quantifier-free fragment of the theory of finite fields mainly exploits Gröbner Bases to decide ideal membership instances, which in turn provides an incomplete test to determine whether a set of polynomials have no common roots. When this fails, it falls back to root search, either over roots of univariate polynomials or exhaustively over all elements in the field.

Our *first contribution* (Section III) is a proof calculus capable of expressing the two kinds of reasoning above, together with code instrumentation enabling CVC5 to emit proofs in that calculus as certificates in the ALETHE [26] proof format for SMT. The proof rules for ideal membership are inspired by the Nullstellensatz proof calculus [5], where each ideal membership proof step is represented by a linear combination. We extend it with rules that capture the root search procedure and rules that handle preprocessing, both for term simplification and for converting finite field constraints into the polynomial form used for computing Gröbner Bases.

Our *second contribution* (Section IV) is an extension of the Practical Algebraic Calculus with rules for root search, as well as an extension of its checker, PACHECK [15], into FFPACHECK, to be able to verify our proofs in the theory of

https://doi.org/

finite fields. This is done by lifting PAC and PACHECK from its original pseudo-boolean setting to arbitrary prime fields and adding support for the case-split introduced by the root search which could not be captured by the original calculus.

We leverage the CARCARA proof checker [2], an efficient standalone checker for ALETHE, to verify the complete SMT certificate, which covers preprocessing, propositional reasoning, and the other involved theories. To handle finite-field reasoning, we encapsulate it in a single ALETHE rule whose side condition is discharged by invoking FFPACHECK.

Our *third contribution* (Section V) is a mechanization of the proof calculus for finite-field constraints in the Lean 4 proof assistant [9], together with a proof of soundness for all of its rules. This mechanization allows us to extend LEAN-SMT [19], a tactic that can reconstruct CVC5 proofs into Lean, to support also goals over the theory of finite fields. This amounts to providing translations of Lean goals expressed in integer arithmetic modulo a prime into an SMT theory of finite fields, as well as reconstructions of the resulting SMT proofs into Lean. To represent polynomials, we use the abstract algebraic representation of multivariate polynomials in Lean, several of whose properties (notably evaluation) are non-computable. To handle this, we introduce an intermediate representation that lets us compute the relevant properties.

Our *fourth contribution* (Section VI) is an evaluation of our instrumented version of CVC5 with proof production for finite fields, which shows two main results: (i) the runtime performance of solving together with proof production is comparable to that of solving with proof production *and* checking in Carcara; and (ii) LEAN-SMT is able to reconstruct a substantial fraction of the proofs generated by CVC5. Out of 403 unsat `QF_FF` instances, CVC5 with proof production solves 392, all verified by CARCARA, while LEAN-SMT reconstructs 247.

Related Work: Finite field reasoning in SMT was introduced for CVC5 and later extended by Ozdemir et al. [20], [21]. It was added in Yices2 by Hader et al. [12]. In this paper, we focus on the finite fields theory solver presented in [20]. Proof production has been explored for several SMT theories. For the solver of interest in our work, CVC5, Barbosa et al. [4] introduced the general framework for proof production in an SMT solver. Independent proof checking is supported by the CARCARA proof checker [2], which targets the ALETHE proof format. More recently, ETHOS [25], a checker for the EUNOIA logical framework has also been introduced. Works on integrating proofs from SMT solvers with proof assistants have been explored in Lean, with the LEAN-SMT tactic [19], Rocq with SMTCOQ [3], [10] and Isabelle/HOL, with SLEDGEHAMMER [7], [16], [27].

To the best of our knowledge, this is the first work on proof certificates for the SMT theory of finite fields. Similar work on SMT certificates beyond linear arithmetic (targeting nonlinear real arithmetic) was done by Mascarenhas et al. [18], which describes the formalization of a proof calculus for incremental linearization as well as their extension to proof reconstruction in LEAN-SMT. Work towards the same goal was explored by Li et al. [17], who extend the Isabelle/HOL proof assistant with support for verifying certificates from univariate Cylindrical Algebraic Decomposition, and by Harrison [13] in sums-of-squares certificates.

Beyond certificates for SMT reasoning, Kaufmann and Biere [15] introduced the Practical Algebraic Calculus and the Nullstellensatz proof calculus for certificates from arithmetic circuit verification, together with the proof checkers PACHECK, and its Isabelle/HOL-formally verified counterpart PASTÈQUE, and NUSS-CHECKER. Our work lifts PACHECK to prime fields and extends it to support proofs beyond ideal membership.

II. TECHNICAL PRELIMINARIES

A. Algebra

We start by introducing relevant algebraic notions and notation as well results that are important to our work.

A *field* F is a set equipped with binary operations $+$ and $\cdot$ (addition and multiplication) that have identities 0 and 1, respectively, inverses (except for 0 in the multiplicative case) and satisfy the associative, distributive and commutative properties. A *finite* field is a field with a finite number of elements, called its *order*. All finite fields have order p^n where p is a prime and n is a positive integer. If t and s are finite field terms, we write $t - s$ as an abbreviation of $t + (-1) \cdot s$.

Finite fields of the same order are isomorphic. We denote a representative of the fields of order p by $\mathbb{F}_p$. Throughout the text, $\mathbf{X} = (X_1, \ldots, X_n)$ will denote an *ordered* list of variables, and $\mathbb{F}_p[\mathbf{X}]$ will denote the set of polynomials with variables in $\mathbf{X}$ and coefficients in $\mathbb{F}_p$.

All reasoning techniques currently implemented in SMT solvers we are aware of are limited to fields of order p^n with $n = 1$. Although the procedures we describe here for checking and producing proofs are not subject to this limitation, we will adopt the same restriction in their presentation. This restriction also simplifies our implementation: with $n = 1$, each field $\mathbb{F}_p$ is the integers modulo p, so that arithmetic reduces to modular integer arithmetic.

We can impose an order for the monomials of a given polynomial f based on $\mathbf{X}$. A standard example is lexicographic order, defined as follows. For all monomials $m_1 = X_1^{a_1} \cdots X_n^{a_n}$ and $m_2 = X_1^{b_1} \cdots X_n^{b_n}$, we have have that $m_1 > m_2$ if the leftmost non-zero difference $a_i - b_i$ is positive. For a given variable order, $\mathtt{lm}(f)$ and $\mathtt{lt}(f)$ denote the greatest monomial and term of f, respectively, where a *term* is the product of a monomial and a constant in $\mathbb{F}_p$, its *coefficient*.

Let $B = \{f_1, \ldots, f_k\}$ be a set of polynomials over $\mathbb{F}_p[\mathbf{X}]$. The set $I = \{g_1 \cdot f_1 + \cdots + g_k \cdot f_k \mid g_i \in \mathbb{F}_p[\mathbf{X}]\}$ is the *ideal generated by* B, also denoted by $\langle B \rangle$. Set B is the *basis* of I. Note that, if I is an ideal, then $hf + kg \in I$ for every $f, g \in I$ and $h, k \in \mathbb{F}_p[\mathbf{X}]$.

The set $\mathcal{V}(I) := \{\mathbf{x} \in \mathbb{F}_p^n \mid \forall f \in I, f(\mathbf{x}) = 0\}$ is the *variety* of I, where $f(\mathbf{x})$ denotes the result of evaluating polynomial f at point $\mathbf{x}$ and $\mathbb{F}_p^n$ the set of n-tuples of elements in $\mathbb{F}_p$. Note that, by definition, if $I = \langle B \rangle$, then $\mathcal{V}(I)$ is exactly the set of common roots of polynomials in B. This suggests a basic test for the **existence of common roots**: If we can prove that

$1 \in \langle B \rangle$, then the polynomials in B have no common roots. The converse of this result holds for algebraically closed fields. However, since finite fields are not algebraically closed, it fails in general when roots are restricted to $\mathbb{F}_p$.

A sound test for **ideal membership** is given by the *reduction* operation. Given $f, g \in \mathbb{F}_p[\mathbf{X}]$, if $\mathtt{lm}(g)$ divides the monomial of some term t of f, we say that f *reduces to* $r := f - \frac{t}{\mathtt{lm}(g)} g$ *under* g. This is extended to a set of polynomials P in the expected way: we say that f *reduces to* r *under* P if r can be derived by a sequence of single-step reductions of f, each step using a polynomial from P whose leading monomial divides some term of the current intermediate result. In such a case, we write $f \Rightarrow_P r$. Note that by this definition, if f reduces to 0 under P, then $f \in \langle P \rangle$. Unfortunately, for finite fields, this test is incomplete. However, completeness can be recovered with Gröbner bases. A *Gröbner basis* G for an ideal I is a generating set of I with the additional property that reduction under G is a complete test for ideal membership.

For our presentation, Gröbner basis computation is understood through Buchberger's algorithm [8], which is used in SMT solvers that rely on Gröbner bases [20], [21]. The algorithm relies on two polynomial operations: reduction by a set of polynomials (as defined above) and computation of S-polynomials. Considering polynomials f, g, their *S-polynomial* $\mathtt{spoly}(f, g)$ is:

$$\mathtt{spoly}(f,g) = \frac{\mathtt{lcm}(\mathtt{lm}(f), \mathtt{lm}(g))}{\mathtt{lt}(g)} f - \frac{\mathtt{lcm}(\mathtt{lm}(f), \mathtt{lm}(g))}{\mathtt{lt}(f)} g$$

Here, $\mathtt{lcm}(a, b)$ denotes the *least common multiple* of a and b, i.e., a polynomial q such that any common multiple of a and b is also a multiple of q. Notice that if f, g are in an ideal I then $\mathtt{spoly}(f, g)$ is also in I.

On input G the algorithm computes a sequence of bases $G_0 = G, \ldots, G_m$, where for some i, G_i is a Gröbner basis. Each G_i is built from G_{i-1} by non-deterministically applying one of the following three rules:

1) If for some $f, g \in G_{i-1}$, $\mathtt{spoly}(f, g)$ reduces to $r \neq 0$ under G_{i-1}, then $G_i = G_{i-1} \cup \{r\}$.
2) If for some $f \in G_{i-1}$, f reduces to $r \neq 0$ under $G_{i-1} \setminus \{f\}$, then $G_i = (G_{i-1} \setminus \{f\}) \cup \{r\}$.
3) If for some $f \in G_{i-1}$, f reduces to zero under $G_{i-1} \setminus \{f\}$, then $G_i = G_{i-1} \setminus \{f\}$.

B. Satisfiability Modulo Finite Fields

We work in the setting of many-sorted classical first-order logic with equality. Consider the many-sorted signature Σ containing the sort Bool, interpreted as the binary set of Boolean values, and a family of function symbols $\approx : \sigma \times \sigma \to \mathsf{Bool}$, for every sort σ in Σ, interpreted as the identity relation over the domain of σ. In this context, a theory is a pair $T = (\Sigma, \mathbf{I})$, where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations that fix the meaning of theory symbols. A formula φ is *satisfiable in* a theory T if there is an interpretation in $\mathbf{I}$ that satisfies φ.

In this work, we are interested in satisfiability in the theory of finite fields, more specifically, in the procedure implemented in CVC5 [20] as a theory solver for the CDCL(T) framework on which CVC5 is based. The main task of a theory solver in CDCL(T) is to determine the satisfiability of a conjunction of literals in a specific theory. In the context of finite fields, the literals are expressions of the form $t \bowtie s$ where $\bowtie \in \{\approx, \not\approx\}$ and t and s are polynomials. That is, CVC5's theory solver for finite fields is tasked to determine whether a set of (dis)equations between finite field polynomials admits a solution. It implements a decision procedure whose high-level description is provided in Figure 1.

Since we will produce proofs in terms of ideal membership and varieties but are only interested in finite field expressions, we consider sets of sort Set whose elements are of finite field sort, a membership predicate symbol $\in$, and the empty set $\emptyset$.

III. Proof Calculus and Proof Production

Our proof calculus is designed to fully capture the decision procedure for the theory of finite fields outlined in Figure 1 (proposed by Ozdemir et al. [20]) while facilitating the instrumentation of a solver implementing this procedure to produce proofs and enabling scalable proof checking.

The decision procedure operates by first simplifying terms via rewriting (line 4 of Figure 1a), then, converting the satisfiability of a set of finite fields literals into the problem of deciding the existence of common roots of polynomials (lines 5 and 6 of Figure 1a). Then, using this new framing, it proceeds by trying to refute the problem via ideal membership testing (lines 7 and 8 of Figure 1a), falling back to a common root search procedure (Figure 1b) if needed.

Our proof rules are written as inference rules of the form:

$$\frac{\varphi_1 \ \cdots \ \varphi_n \quad | \quad t_1, \ldots, t_m}{\psi} \ [\mathtt{if}\ C]$$

where $m, n \geq 0$; $\varphi_1, \ldots, \varphi_n$ are the rule's *premises*; $t_1, \ldots, t_m$ are the rule's parameters, or *arguments*; ψ is the conclusion; and C is an optional *side condition*.

A. Proving Polynomial Equalities

SMT solvers often rewrite expressions in order to simplify formulas. This is represented in the decision procedure outlined in Figure 1. For finite field expressions, there are seven arithmetic rewriting rules applied to completion modulo associativity and commutativity (AC) of $\cdot$ and $+$:

$$\begin{array}{rcl@{\qquad}rcl} t + c_1 + c_2 & \longrightarrow & t + a & t + 0 & \longrightarrow & t \\ c_1 \cdot c_2 \cdot t & \longrightarrow & m \cdot t & 1 \cdot t & \longrightarrow & t \\ c_1 \cdot t + c_2 \cdot t & \longrightarrow & a \cdot t & 0 \cdot t & \longrightarrow & 0 \\ t \cdot (t_1 + t_2) & \longrightarrow & (t \cdot t_1) + (t \cdot t_2) & & & \end{array}$$

where c_1 and c_2 are constants, a is the result of their addition and m the result of their multiplication.

The rewrite system above is equivalence preserving and normalizing modulo AC of $\cdot$ and $+$. In particular, any two irreducible rewrites t_1 and t_2 of the same term t are both equivalent to t, are in polynomial form, and can be made identical under a fixed monomial order. Hence we can talk about *the normal form* $\mathtt{Norm}(t)$ *of* t, the polynomial obtained

In: A set of $\mathbb{F}_p$-literals L in variables X
Out: UNSAT/SAT
1 *Decision Procedure*
2 $G \leftarrow \emptyset; W_i \leftarrow \text{freshVar}(), \forall i \in \{1, \ldots, |L|\}$;
3 **for** $(s_i \bowtie_i t_i \in L)$
4 $s_i \leftarrow \mathit{Rewrite}(s_i)$; $t_i \leftarrow \mathit{Rewrite}(t_i)$;
5 **if** $(\bowtie_i = \approx)$ $G \leftarrow G \cup \{s_i - t_i\}$;
6 **elif** $(\bowtie_i = \not\approx)$ $G \leftarrow G \cup \{W_i(s_i - t_i) - 1\}$;
7 $B \leftarrow \mathbf{GB}(G)$;
8 **if** $(1 \Rightarrow_B 0)$ **return** UNSAT**;**
9 $m \leftarrow \mathit{FindZero}(B)$;
10 **if** $(m = \bot)$ **return** UNSAT;
11 **else return** SAT;

(a) Main procedure. *Rewrite* applies to completion the arithmetic simplification rules described in Section III.

In: A Gröbner basis $B \subset \mathbb{F}_p[\mathbf{X}']$
In: A partial map $M : \mathbf{X}' \to \mathbb{F}_p$ (empty by default)
Out: A total map $M : \mathbf{X}' \to \mathbb{F}_p$ or $\bot$
1 *FindZero*
2 **if** $(1 \in \langle B \rangle)$ **return** $\bot$;
3 **if** $(|M| = |\mathbf{X}'|)$ **return** M;
4 **for** $((X'_i \mapsto z) \in \mathit{ApplyRule}(B, M))$
5 $r \leftarrow \mathit{FindZero}(GB(B \cup \{X'_i - z\}), M \cup \{X'_i \mapsto z\})$;
6 **if** $(r \neq \bot)$ **return** r;
7 **return** $\bot$

(b) Common roots search procedure.

Fig. 1: CVC5's decision procedure for the theory of finite fields.

(in finite time) by applying the rewrite rules above to t to completion and then reordering the result according to the chosen monomial order. An important consequence of normalization is that polynomials with the same normal form are equivalent. We capture this property with the `ff_poly_norm` proof rule, which derives the equality of two polynomials s and r when their normal forms are the same:

$$\frac{- \quad | \quad s, t}{t \approx s} \ \texttt{if Norm}(t) \approx \texttt{Norm}(s)$$

The rule above represents polynomial-level rewriting. However, the literals we deal with are (dis)equalities. To lift normalization to equality literals, we introduce the rule `ff_poly_norm_eq`:

$$\frac{c_1 \cdot (t_1 - s_2) \approx c_2 \cdot (t_2 - s_2) \quad | \quad -}{(t_1 \approx s_1) \approx (t_2 \approx s_2)}$$

where c_1 and c_2 are scaling constants and the equality literals in question are $t_1 \approx s_1$ and $t_2 \approx s_2$. Equalities matching the single premise of this rule can be derived using `ff_poly_norm`.

Proving polynomial equalities and the equivalence of equalities is essential both for justifying rewriting steps and for the polynomial conversion proofs. As discussed later, these rules also enable finer-grained proofs of ideal membership.

B. Polynomial Conversion

The first step of the decision procedure is to translate the satisfiability of a set of equational literals into the existence of common roots of a set of polynomials. We break this conversion into three kinds of step: *equality*, *disequality*, and *problem setting conversions*.

1) Equality Conversions: We normalize equality literals $\ell = (t \approx s)$ to literals of the form $\ell' = (t - s \approx 0)$. This way the roots of the polynomial $t - s$ correspond exactly to the satisfying assignments for ℓ. More precisely, if ℓ has been assumed or derived, we first derive the literal $1 \cdot (t-s) \approx 1 \cdot ((t-s)-0)$ by `ff_poly_norm`, which we can do as the two sides of the equality have the same normal form. Then we can apply `ff_poly_norm_eq` to that literal, with $c_1 = c_2 = 1$, and derive the equivalence $(t \approx s) \approx (t - s \approx 0)$. Finally, we can apply a modus-ponens-like rule (not shown here) to derive ℓ' from ℓ.

2) Disequality Conversions: Disequality literals have the form $\ell = (t \not\approx s)$. Notice that if w is a fresh variable, the literal $\ell' = (w \cdot (t - s) - 1 \approx 0)$ is equisatisfiable with ℓ. In fact, all satisfying assignments of ℓ can be extended to w to satisfy ℓ' by just assigning to w the multiplicative inverse of the (non-zero) value assigned to $t - s$. Moreover, every assignment that satisfies ℓ' must give different values to t and s; hence, it satisfies ℓ.

We introduce the rule `ff_diseq` to derive ℓ' from ℓ, but rather than using a fresh variable w to represent multiplicative inverse of $t - s$, we use the *Hilbert choice* operator ε instead. The term $\varepsilon w.\, \phi(w)$ denotes a value satisfying ϕ when one exists; in this context, the constraint on the value is that it is the multiplicative inverse. This way, the soundness of this rule does not depend on global assumptions about the freshness of introduced variables:

$$\frac{- \quad | \quad s, t}{(t \not\approx s) \ \approx \ ((\varepsilon w.\, w \cdot (t - s) - 1 \approx 0) \cdot (t - s) - 1 \approx 0)}$$

Similarly to the equality conversion, given a disequality ℓ and the equality derived by the rule above, we can derive an equality equivalent with ℓ'.

3) Problem Setting Conversions: Given one or more literals in polynomial form ($f \approx 0$), for instance derived as explained above, the rule `ff_poly_conversion` lifts those literals to the setting of polynomial varieties:

$$\frac{g_1 \approx 0 \ \cdots \ g_n \approx 0 \quad | \quad -}{\mathcal{V}(\langle g_1, \ldots, g_n \rangle) \not\approx \emptyset}$$

C. Ideal Membership Proofs

The decision procedure from Figure1a perfoms two forms of ideal membership reasoning:

1) Gröbner Basis computation (line 7 of Figure 1a), that enables a complete test for ideal membership.

2) Ideal membership testing (line 8 of Figure 1a), used as an incomplete refutation procedure.

Considering the relevant operations outlined in the preliminaries section, both forms of reasoning can be captured by just two proof rules: `ff_ideal_generator`, which states that the generators are in the ideal they generate, and `ff_poly_combination`, which states that a linear combination of elements in the ideal is also in the ideal:

$$\frac{- \quad | \quad f, G}{f \in \langle G \rangle} \ \texttt{if}\ f \in G$$

$$\frac{r_1 \in \langle G \rangle \ \cdots \ r_k \in \langle G \rangle \quad | \quad \texttt{Seq}_m}{\sum_{i=1}^{k} m_i \cdot r_i \in \langle G \rangle}$$

where $\texttt{Seq}_m$ denotes the list $m_1, \ldots, m_k$ of *multipliers*, consisting of arbitrary polynomials.

1) Gröbner Basis Computation: For Gröbner basis computation, we use the rules above to prove preservation of ideal membership, following the sequence-of-bases formulation from the background section.

First, membership of every polynomial in $G_0 = G$ is established via `ff_ideal_generator`. Since we only need to preserve membership, only steps 1 and 2 are relevant. These steps use two operations: computation of S-polynomials and reduction by a set of polynomials.

Recall that $\texttt{spoly}(f, g) = q_1 \cdot f + q_2 \cdot g$, with q_1 and q_2 as defined in the background section. Then, if $f, g \in \langle G \rangle$, we can instantiate `ff_poly_combination` with membership facts of f and g as premises and (q_1, q_2) as the list of multipliers, in order to prove that $\texttt{spoly}(f, g) \in \langle G \rangle$.

Reduction can be written as $\texttt{reduce}(f, g) = f + q \cdot g$. For each reduction step with $g \in G_{i-1}$ and $f \in \langle G \rangle$, we instantiate `ff_poly_combination` using the membership facts for f and g as premises and $(1, q)$ as the list of multipliers.

2) Ideal Membership Testing: Our goal is to prove that if an arbitrary polynomial f reduces to zero under $Q \subseteq \langle G \rangle$, then $f \in \langle G \rangle$. By the definition of reduction, $f - q_1 \cdot g_1 - \cdots - q_k \cdot g_k = 0$ for some $g_i \in Q$ and polynomials q_i. Hence $f = q_1 \cdot g_1 + \cdots + q_k \cdot g_k$, and we instantiate `ff_poly_combination` with the membership facts for $g_1, \ldots, g_k$ as premises and $(q_1, \ldots, q_k)$ as the list of multipliers, concluding $f \in \langle G \rangle$.

D. Branching Rules

To achieve completeness, since ideal membership is an incomplete refutation procedure, we must capture the root search procedure `FindZero`. The `ApplyRule` procedure (line 4 of Figure 1b) produces a list of possible assignments according to three criteria:

1) If there is a *univariate* polynomial $f \in B$ in some variable x, we can restrict the search for common roots to the roots of f.
2) When the ideal has additional structure, specifically when $\langle B \rangle$ is zero-dimensional, we can compute a *univariate* minimal polynomial $f \in \langle B \rangle$ in some unassigned variable, and then proceed as above by restricting the search to its roots.
3) As a last resort, we search exhaustively over all possible assignments in the field.

Rules 1 and 2 share a common structure: there is a univariate polynomial in the ideal (whose membership we can prove), and the search can be restricted to its roots. This is captured by a single proof rule, `ff_root_branch`:

$$\frac{\mathcal{V}(\langle G \rangle) \not\approx \emptyset \quad f \in \langle G \rangle \quad | \quad x, \texttt{Roots}(f)}{\bigvee_{v \in \texttt{Roots}(f)} \mathcal{V}(\langle G \cup \{x - v\} \rangle) \not\approx \emptyset}$$

Here, f is a univariate polynomial in x. Including $x - v$ restricts the variety to points with $x = v$. If f has no roots, the conclusion is an empty disjunction, i.e., false, as desired.

For exhaustive branching, it suffices to enumerate all possible values of a variable. This is captured by rule `ff_exhaust_branch`:

$$\frac{\mathcal{V}(\langle G \rangle) \not\approx \emptyset \quad | \quad x}{\bigvee_{v \in F_p} \mathcal{V}(\langle G \cup \{x - v\} \rangle) \not\approx \emptyset}$$

E. Unsatisfiability Conclusion

Finally, we need a rule to conclude that a problem or branch is unsatisfiable. The rule rests on the fact that an ideal containing 1 has an empty variety. Applied to each disjunct of a branching rule conclusion, or to the conclusion of `ff_poly_conversion`, it produces a contradiction.

$$\frac{1 \in \langle G \rangle \quad | \quad -}{\mathcal{V}(\langle G \rangle) \approx \emptyset}$$

F. Implementation

We instrumented CVC5's finite field decision procedure to produce proofs leveraging CVC5's extensive proof infrastructure [4]. We also instrumented its dependency CoCoAlib [1], which performs Gröbner basis computation and ideal membership testing, to emit proofs in our calculus. To deal with the two different representations for polynomials, in CVC5 and CoCoA, we augmented the interface between the tools to handle the translation between them.

Each ideal membership of linear combinations step is initially recorded by an application of a single coarse-grained rule `macro_ff_poly_combination`:

$$\frac{r_1 \in \langle G \rangle \ \cdots \ r_k \in \langle G \rangle \quad | \quad \texttt{Seq}_m, f}{f \in \langle G \rangle} \ \texttt{if}\ f = \sum_{i=1}^{k} m_i r_i$$

In a post-processing step, such applications can be refined using the finer-grained `ff_poly_combination` rule, which has no side condition, as illustrated in Figure 2. The choice to apply this proof elaboration or not is left to the user.

The rule above requires capturing the premise polynomials and the multipliers; both are produced internally by CoCoAlib. The premises can be recorded with high-level hooks in the Gröbner basis algorithm. The multipliers, however, must be intercepted at a lower level, since CoCoAlib uses different polynomial representations depending on the size of the field; capturing them there also lets us reuse CoCoAlib's intermediate results instead of recomputing them, which would incur undue overhead during proof production.

$$\frac{r_1 \in \langle G \rangle \ \cdots \ r_k \in \langle G \rangle}{p \in \langle G \rangle} \quad \Longrightarrow \quad \frac{\dfrac{r_1 \in \langle G \rangle \ \cdots \ r_k \in \langle G \rangle}{\sum_{i=1}^{k} m_i \cdot r_i \in \langle G \rangle} \qquad \dfrac{}{p \approx \sum_{i=1}^{k} m_i \cdot r_i}}{p \in \langle G \rangle}$$

Fig. 2: Elaboration of `macro_ff_poly_combination` (left) into a derivation that combines the finer-grained `ff_poly_combination` with `ff_poly_norm` and a congruence step (right).

We output proofs in two formats: ALETHE [26] and CVC5's internal format, the Cooperating Proof Calculus (CPC) [24].

Figure 3 shows the proof of unsatisfiability in each format for $(a-0)\cdot(a-1)\cdot(a-2) \approx 1$ in $\mathbb{F}_3$, where `@ff.ideal` and `@ff.variety` correspond to new operators used in the proofs to represent the ideal of a set of polynomials and that the variety of an ideal is non-empty, respectively.

In the ALETHE format we package all theory specific reasoning into a single rule, `ff_pac`, of the following form:

$$\frac{\mathcal{V}(\langle G \rangle) \not\approx \emptyset \quad | \quad \texttt{pac\_proof}}{\bot}$$

Here `pac_proof` is a proof in the format detailed in the next section, intended to be checked with our extension of PACHECK. Polynomial conversion and polynomial normalization are modeled as new ALETHE rules. We have extended the ALETHE checker CARCARA with support for them.

The CPC format was extended with rules directly mirroring the abstract calculus above. We have extend LEAN-SMT correspondingly so it can reconstruct all of them into Lean.

IV. PROOF CHECKING

SMT certificates combine proofs for preprocessing and propositional steps with theory specific reasoning. To check these certificates we have a pipeline combining the standalone ALETHE checker CARCARA, an efficient proof checker for the ALETHE format written in Rust, and our extension of PACHECK, called FFPACHECK. All non-finite-field, polynomial conversion and normalization rules are checked by CARCARA, while FFPACHECK is responsible for checking rules justifying that a polynomial variety is empty.

Overall, we have extended both the ALETHE format and CARCARA with support for finite field sorts and expressions, both from the theory and the proof-specific ones, used to represent polynomial ideals and varieties. We added proof checking for the rules `ff_diseq`, `ff_poly_norm`, `ff_poly_norm_eq`, and `ff_poly_conversion`. To check the `ff_pac` rule, which encapsulates polynomial reasoning, we rely on FFPACHECK as an external dependency which takes the argument of the `ff_pac` proof step, collected in a temporary file.

The rules `ff_poly_conversion`, `ff_poly_norm_eq`, and `ff_diseq` are checked directly via pattern matching: we verify that the arguments and premises have the expected format and use them to build an expected conclusion that is matched against the one given in the certificate.

The `ff_poly_norm` rule, which concludes $t_1 \approx t_2$ when t_1 and t_2 have the same normal form, as defined in Section III-A, requires normalizing $t_1 - t_2$ and verifying that the result is the zero polynomial. Polynomials are represented as an IndexMap from monomials to their coefficient. Since monomials serve as keys in this map, they need a canonical representation supporting equality and hashing. We represent each monomial as a list of variable terms sorted by their pointers. Since CARCARA applies *hash consing*, all occurrences of the same variable share the same pointer, and this yields a canonical representation.

FFPACHECK

We use PAC [15] as a baseline of our new proof calculus for finite fields in order to leverage its efficient proof checker, PACHECK. The original version of PAC is defined over pseudo-Boolean polynomials, where variables are Boolean and coefficients are integers. Proofs in PAC consist of a set of polynomial axioms and a series of proof steps admitting linear combinations of axioms and a target polynomial. The goal of a PAC proof is to show that the target polynomial is contained in the ideal generated by the axioms.

The PAC proof format, as defined in Kaufmann et al. [15], can be understood via two proof rules: an *axiom* rule, declaring all the ideal generators and thus analogous to `ff_ideal_generator`, and a rule similar to `ff_poly_combination`, called the *linear combination* rule. The key invariant of a PAC proof is that every derived polynomial from a linear combination rule lies in the ideal generated by the axioms. The original PAC format is specialized towards the pseudo-Boolean setting by handling Boolean arithmetic via the normalization $x \cdot x \longrightarrow x$. We adapted PAC to the finite-field setting and implemented a corresponding proof checker FFPACHECK. We had to adapt the normalization rules to accommodate the finiteness of the coefficients and variables.

Normalization: All polynomials maintained by the checker are kept in a canonical form as follows:

1) *Coefficient reduction.* All coefficients are reduced modulo p after every arithmetic operation.
2) *Functional equivalence over $\mathbb{F}_p$.* When evaluating polynomials on elements of $\mathbb{F}_p$, we use the identity $a^p \approx a$ for all $a \in \mathbb{F}_p$. Equivalently, two polynomials are considered functionally equivalent over $\mathbb{F}_p$ if they differ by a multiple of $x^p - x$.

In contrast to PAC, where the goal is only to prove ideal membership of a target polynomial, in our calculus the main goal is to derive refutations in a more general way, which can include branching on the potential values a variable can take. PAC, however, does not accommodate this type of reasoning, i.e., a PAC proof does not admit branching nor has mechanisms to represent variables being assigned values on a branch.

```
(step t30
  (cl (not (set.is_empty (@ff.variety (@ff.ideal @g)))))
  :rule ff_poly_conversion :premises (...) :args (...))

(step t31 (cl) :rule ff_pac :premises (t30)
  :args (m 3;
         a 1 v1^3 + 2 * v1 + 2;
         r 1 v1;))
```

(a) ALETHE

```
; Below, @g := a^3 + 2*a + 2

(step @p2 (set.member @g (@ff.ideal @g))
  :rule ff_ideal_generator)

(step @p31
  (not (set.is_empty (@ff.variety (@ff.ideal @g))))
  :rule ff_poly_conversion :premises (...))

(step @p32 false
  :rule ff_root_branch :premises (@p31 @p2))
```

(b) CPC

Fig. 3: Theory specific steps of the proof of unsatisfiability for $(a-0)\cdot(a-1)\cdot(a-2)=1$ in $\mathbb{F}_3$ in each output format.

Before describing individual rules here, we introduce the overall shape of an FFPACHECK proof. It extends PAC to support case splitting on the roots, corresponding to the rules `ff_root_branch` and `ff_exhaust_branch` in the abstract calculus of Section III, thus allowing proofs to be not only a linear derivation but also a tree.

A *root* or *exhaustive rule* first declares the complete set of feasible root values of a variable x, enumerating the sub-derivations that must be closed. Opening each of these sub-derivations is then the task of a separate *branching rule*, which extends the axioms currently in scope with $x - v$ for each declared root v, one at a time. Each sub-derivation then proceeds independently as its own linear derivation, which may itself branch further through nested applications of `ff_root_branch` or `ff_exhaust_branch`. Let $x \approx v \vdash \bot$ stand for "$\bot$ is derivable from the assumption $x \approx v$." Every sub-derivation must establish $x \approx v \vdash \bot$, either by deriving the constant 1, validly declaring an empty root set, or by branching further. An FFPACHECK proof closes exactly when every leaf of the tree has been refuted in this way. In the following, we introduce the new proof rules that were added to support this kind of reasoning and also explain how existing PAC rules were adapted.

A. Rule Types

Rules marked as "adapted" were already present in PAC; the remaining rules are new.

Modulus rule (new): `m` ⟨`value`⟩`;`
Sets the prime modulus p for all subsequent polynomial operations. This rule must appear before any rule that depends on field arithmetic. For example, `m 3;` selects $\mathbb{F}_3$.

Axiom rule (adapted): `a` ⟨`id`⟩ ⟨`polynomial`⟩`;`
This rule instantiates `ff_ideal_generator` and declares a new polynomial axiom under the given identifier. The polynomial is immediately normalized as described above. For example, with order $p = 3$ the axiom `a 1 x^5 - 3*x + 2;` introduces the normalized polynomial $x+2$ under identifier 1.

Linear combination rule (adapted):

`l` ⟨`id`⟩ ⟨`id`$_1$⟩`*(`⟨`mult`$_1$⟩`) + `⟨`id`$_2$⟩`*(`⟨`mult`$_2$⟩`) + ⋯,` ⟨`expected`⟩`;`

This rules instantiates `ff_poly_combination` and derives a new polynomial as a linear combination of existing ones and stores it under a fresh identifier. The checker verifies that the computed combination matches the declared expected result.

With these three rules we could simply lift the basic structure of PAC to finite fields, i.e., every result of the linear combination rule is a member of the ideal generated by the axioms in the finite field modulo the value stated by the modulus rule. However, as we now also want to accommodate the setting of case analysis on the roots of a derived univariate polynomial, or exhaustive search, we need to include branch resolution rules.

B. Branch Resolution

The resolution procedure works as follows:

- A root rule `r` (or exhaustive rule `e`) declares the complete set of roots for a variable x, thereby enumerating all branches that must be closed.
- Each branch rule `b` opens one branch for a single declared root v, introducing the polynomial $x - v$ into the local context.
- A branch closes when either (a) a linear combination deriving the constant 1 is recorded in that branch, or (b) a root rule validly declares an empty root set in that branch.
- The proof is globally complete only when every branch opened by every root declaration has been closed.

Root rule (new): `r` ⟨`id`⟩ ⟨`var`⟩ `[`⟨`root`$_1$⟩ ⟨`root`$_2$⟩ `⋯];`
This rule instantiates `ff_root_branch` and validates the roots of the polynomial stored under ⟨`id`⟩, which must be univariate in ⟨`var`⟩. The checker independently computes $\gcd(p, x^p - x)$ using the GCD implementation of FLINT [28], extracts all roots in $\mathbb{F}_p$, and requires the declared root list to match exactly. The verified roots are stored and become the allowed branching domain for that variable. When the declared root list is empty (e.g., `r 2 x;`), the root set is verified to be empty and the current branch closes. This case can happen when the univariate polynomial is either normalized to a constant or all its roots are in the algebraic closure of $\mathbb{F}_p$, but not in $\mathbb{F}_p$.

Exhaustive root rule (new): `e` ⟨`var`⟩`;`
Implementing `ff_exhaust_branch`, this rule declares the entire field $\mathbb{F}_p$ as the branching domain for variable v, without reference to any particular polynomial. This rule subsumes the root rule when all field elements are potential roots.

Branch rule (new): `b ⟨id⟩ ⟨var⟩ ⟨root⟩;`
This rule is not contained as a stand-alone rule in the abstract calculus in Section III. It is rather a fine-grained instantiation of the individual roots in the disjunction in the conclusions of the rules `ff_root_branch` and `ff_exhaust_branch`. It opens a nested branch for one declared root of ⟨var⟩ and stores the polynomial (⟨var⟩ − ⟨root⟩) under the fresh identifier ⟨id⟩. The root must have been declared for that variable by a prior root or exhaustive rule. For example, `b 12 x 1;` opens a branch for the assignment $x = 1$.

The rules above jointly implement *disjunction elimination*, applied to the disjunctive conclusion of `ff_root_branch`. We can state the whole procedure as a single rule, in the same style as the rest of the calculus introduced in Section III. We write $x \approx v$ for the disjunct $\mathcal{V}(\langle G \cup \{x - v\}\rangle) \not\approx \emptyset$ of `ff_root_branch`'s conclusion, matching how the branch rule above is described as opening "a branch for the assignment $x \approx v$" — and reuse $x \approx v \vdash \bot$, as introduced above:

$$\frac{\bigvee_{v \in \texttt{Roots}(f)} x \approx v \quad \{x \approx v \vdash \bot\}_{v \in \texttt{Roots}(f)} \quad | \quad x}{\bot}$$

The left premise is exactly the conclusion of `ff_root_branch`; the right premise requires, for every declared root v, a proof that the corresponding branch is refutable. The same rule works for `ff_exhaust_branch`, ranging both premises over $v \in \mathbb{F}_p$ instead of $\texttt{Roots}(f)$, with no polynomial involved.

Read bottom-up, this rule mirrors the resolution procedure above. The root rule (or exhaustive rule) supplies the left premise by declaring $\texttt{Roots}(f)$ (or $\mathbb{F}_p$). Each entry of the right premise comes from one branch: a branch rule opens it by adding $x - v$ to the local axioms, and the branch closes either by deriving 1 through linear combination steps (which acts as an instance of the *Unsatisfiability Conclusion* rule) or by a nested instance of this very rule.

C. Implementation

FFPACHECK is implemented in C++. Polynomial coefficients and exponents are represented using a custom `BigInt` type backed by the GNU Multiple Precision Arithmetic Library (GMP), allowing the checker to handle arbitrarily large primes. A multivariate polynomial is stored as an `std::map` from monomials (themselves maps from variable names to exponents) to coefficients. All arithmetic operations immediately apply normalization: coefficients are reduced modulo p, and exponents are reduced using the field identity $x^p = x$.

The checker processes proof files line by line. All declared polynomials are stored in a hash map from integer identifiers to polynomials. The branch state is maintained via a substitution stack that records the current variable-to-value assignments; when a branch rule `b` is processed the assignment is pushed onto the stack and all subsequent polynomial operations are performed under that substitution. The set of roots declared for each variable is tracked in a separate hash map. Checking the proof succeeds only when every declared root set has been fully consumed by matching branch rules.

D. Example

The following proof witnesses the unsatisfiability of the system $\{x^2 - x,\ x + y,\ 2x + y + 1,\ x + z + 2,\ 2x + z\}$ over $\mathbb{F}_3$. The proof branches on the two roots of $x^2 - x$ in $\mathbb{F}_3$, namely $x = 0$ and $x = 1$, and derives a contradiction in each branch. Comments in the proof can be added by lines starting with the character `c`. We also present how these rules can be understood by the proof calculus presented on Section III.

```
m 3;
c Axioms ↔ conversion proofs.
a 1 x^2 - x;
a 2 x + y;
a 3 2*x + y + 1;
a 4 x + z + 2;
a 5 2*x + z;
c ff_root_branch on x^2 - x;
r 1 x 0 1;
c Branch x = 0: 2x + 2(x + y) + (2x + y + 1) = 1
b 10 x 0;
c ff_poly_combination for 1 membership
l 11 10 * (2) + 2 * (2) + 3 * (1), 1;
c Branch x = 1: (x − 1) + (x + z + 2) + 2(2x + z) = 1
b 20 x 1;
c ff_poly_combination for 1 membership
l 21 20 * (1) + 4 * (1) + 5 * (2), 1;
```

Lines `a 1`–`a 5` declare the five axiom polynomials. Rule `r 1` validates that 0 and 1 are the only roots of $x^2 - x$ in $\mathbb{F}_3$, opening two branches. In the branch $x = 0$, rule `b 10` introduces $x - 0 = x = 0$, and the linear combination `l 11` derives $2 \cdot x + 2 \cdot (x + y) + (2x + y + 1) = 1$. An analogous derivation closes the branch $x = 1$. With both branches closed, the proof concludes.

V. Proof Reconstruction and Automation

We instrumented LEAN-SMT [19] with the proof calculus from Section III to enable reasoning about finite fields in Lean. LEAN-SMT operates on Lean's `ZMod p` datatype and assumes the typeclass hypothesis `[Fact p.Prime]`, in line with other verification projects that use this representation [23], [29]. For example, the following theorem states that there is no element in `ZMod 5` that squares to 3:

```
example [Fact (Nat.Prime 5)] (x : ZMod 5)
  : ¬(x * x = 3) := by smt
```

To discharge this goal, LEAN-SMT translates it into the following SMT query

```
(set-logic ALL)
(declare-const x (_ FiniteField 5))
(assert (not (not
  (= (ff.mul x x)
     (as ff3 (_ FiniteField 5))))))
(check-sat)
```

dispatches it to CVC5, and reconstructs the resulting proof inside Lean as we describe below.

A Lean proposition holds exactly when its negation is unsatisfiable. Therefore, LEAN-SMT asserts the negated goal as an SMT formula—note the outer `(not ...)` wrapping the body of the assertion above. An `unsat` answer from

```
lemma ideal_generator
  {xs y zs : MvPolynomial Nat (ZMod p)}
  : y ∈ ideal (xs ++ y :: zs) := by
  exact Ideal.subset_span (by simp)
```

Fig. 4: Lemma corresponding to `ff_ideal_generator`.

CVC5 certifies the original theorem. Reconstruction proceeds rule by rule: each step in the proof calculus is discharged by a corresponding lemma in Lean. To reuse results about polynomial varieties and ideal membership from Lean's math library Mathlib, we adopt `MvPolynomial Nat (ZMod p)` as our internal representation of polynomials, where `Nat` indexes variables and `ZMod p` is the coefficient domain. Different proof rules, however, are discharged using different supporting datatypes, as we now illustrate.

Ideal membership rules: The polynomial rules `ff_diseq`, `ff_ideal_generator`, `ff_poly_combination`, and `ff_exhaust_branch` are all justified directly over `MvPolynomial`, by appealing to Mathlib lemmas about ideals and varieties. For instance, the correctness of `ff_ideal_generator` reduces to the lemma in Figure 4, which simply states that any generator of an ideal belongs to the ideal.

Conversion rules: Mathlib's `MvPolynomial` is an algebraic structure that exposes no syntactic representation of its elements, which makes the conversion between Lean's modular arithmetic and our polynomial setting awkward to formalize directly. We therefore introduce an intermediate inductive datatype `ZMod.Expr p` that captures the AST of a `ZMod p` expression. Given a context `ctx` mapping variable indices to `ZMod p` values, we define function `eval : (ctx : Nat → ZMod n) → Expr n → ZMod n`, which interprets an expression as an element of `ZMod p`, and function `toPoly : Expr n → MvPolynomial Nat (ZMod n)`, which embeds the expression into Mathlib's algebraic representation.

A second motivation for `ZMod.Expr p` is that several properties of polynomials are not computable from the algebraic `MvPolynomial` structure but become decidable on the AST. For example, justifying the `ff_root_branch` rule requires checking that a polynomial is univariate and that the roots produced by CVC5 are exhaustive, which can be reduced to the evaluation of the expression on every element of the field via `eval`.[1] Figure 5 shows the corresponding lemma, which relies on the `completeRoots` predicate built on top of `eval`.

Normalization: To support the `ff_poly_norm` rule we implement a verified normalization procedure `ZMod.Expr.toPolynomial` that maps a `ZMod.Expr p` expression to an auxiliary datatype `Polynomial p` representing its canonical form. Correctness, stated in Figure 6, says that two expressions whose canonical forms agree evaluate to the same value in any context.

[1]This could be done more effectively via a GCD computation, similarly to how FFPACHECK does it, but would require significant effort to do an efficient, provably correct, GCD computation in Lean, which we leave as future work.

```
def Expr.completeRoots [NeZero n] (p : Expr n)
  (i : Nat) (rs : List (ZMod n)) : Bool :=
  p.isUnivariateOver i &&
  ∀ r, p.eval (fun _ => r) = 0 ↔ r ∈ rs

lemma root_branch [Fact n.Prime] {p : Expr n}
  : variety (ideal ps) ≠ ∅
  → p.toPoly ∈ ideal ps
  → p.completeRoots i rs
  → orN (rs.map λ r => variety
    (ideal (ps ++ [.X i + .C (-r)]))) ≠ ∅)
```

Fig. 5: Lemma corresponding to `ff_root_branch`.

```
lemma eval_eq_of_beq {e₁ e₂ : Expr o}
  : e₁.toPolynomial == e₂.toPolynomial →
    e₁.eval ctx = e₂.eval ctx
```

Fig. 6: Lemma corresponding to `ff_poly_norm`.

We deliberately do not delegate this normalization to Lean's `ring` tactic, which also performs such a normalization. First, `ring` normalizes its input recursively inside the kernel, which is expensive on large polynomial expressions typical of SMT preprocessing; `toPolynomial` instead computes a single canonical form per relevant top-level expression and decides equality by syntactic comparison. Second, in our experiments `ring` fails to scale on the polynomial sizes that arise in our benchmarks, while `toPolynomial` handles them reliably.

VI. Evaluation

We performed an experimental evaluation aimed at measuring the impact of proof production for the theory of finite fields in CVC5, as well as the effectiveness of the proof checking executed by our extension of CARCARA together with FFPACHECK, and the proof reconstruction via our extension of LEAN-SMT. We compared against the base performance of CVC5 1.3.3 while solving the selected problems.[2] We implemented our extension on top of CARCARA commit f95bbd1 and LEAN-SMT commit 9b31496.

A. Benchmarks

To evaluate our implementations, we used benchmarks from [20], selecting the 403 benchmarks for which CVC5 returns outputs unsat in under 20 minutes with 16GB of RAM. We ran all of our experiments on a cluster with nodes equipped with 48 AMD Ryzen 9 7950X processors running at 4.50GHz with a 20 minutes time out and 16GB of RAM.

B. Results

To establish a baseline, we consider here two configurations of CVC5 where proofs are not proof producing: default CVC5 and CVC5+simp=none, which refers to the option `--simplification=none`, used natively by LEAN-SMT [19]

[2]The experimental data is available in an artifact at https://doi.org/10.5281/zenodo.20133204.

Tool	Solved Instances	TO	MO
CVC5	403	0	0
CVC5+simp=none	392	8	3
CVC5+Proof	392	11	0
CVC5+Proof+CARCARA	392	11	0
LEAN-SMT	348	24	31
Total unsat	403	N/A	N/A

TABLE I: Evaluation of Proof Checker Extensions on unsat QF_FF benchmarks. TO stands for timeout and MO for memout.

Tool	Cumulative Time (s)
CVC5+simp=none	249.48s (1.00×)
CVC5+Proof	599.59s (2.40×)
CVC5+Proof+CARCARA	642.18s (2.57×)
LEAN-SMT	9280.45s (37.20×)

TABLE II: Cumulative solving times on benchmarks solved by all configurations (348 total). Slowdowns are reported relative to CVC5 with `--simplification=none`.

when invoking CVC5. We keep the option for consistency when evaluating all proof-producing versions of CVC5, even though CVC5 can produce proofs for it and CARCARA can check them. The option disables a preprocessing pass that infers a substitution from the equalities present in the input formula and applies it to eliminate symbols, and can have a quite significant impact on performance [4].

The number of solved instances, which may include proof checking or reconstruction, in each pipeline is shown in Table I. The disabled preprocessing pass leads to 11 unsolved problems, but producing proofs (CVC5+Proof), or checking them with CARCARA and FFPACHECK (CVC5+Proof+CARCARA), does not cost other solved instances, i.e., they solve, or solve and verify, the same 392 problems. LEAN-SMT can solve and reconstruct 348 instances. Execution times are heavily affected by the proof reconstruction and type-checking overhead incurred within Lean, most notably for the `ff_poly_norm` rule, whose checking requires normalizing polynomials.[3]

Table II compares cumulative solving times across commonly solved benchmarks. The scatter plot in Figure 7 shows the runtime distribution across all solved benchmarks. As expected, CVC5 without proof production solves the highest number benchmarks and is the fastest. The plot shows that solving with proof production and verification combined has comparable performance to proof production alone. The plot and Table II also show that the impact of proof production is on par with evaluations of the proof production of CVC5 across other theories [4].

[3]In our evaluation each instance is checked in a separate Lean file with a single lemma corresponding to the conversion of the original SMT query, and as a result all dependencies are reloaded for every instance. This includes Mathlib, a massive mathematical library written in Lean. Since Mathlib's loading times have a large variance and loading it repeatedly is arguably an artifact of our evaluation, we do not include this time in the results, but note that the loading time per instance was on average 50s with a median of 10s.

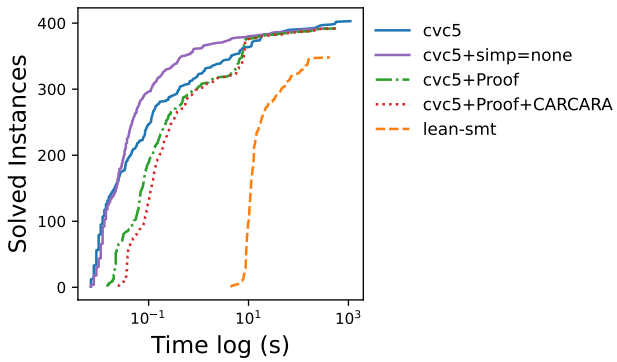

Fig. 7: Benchmarks solved over time

On average, verifying a proof with CARCARA adds only 0.50 seconds over proof production alone. Roughly 77% of the total verification time is spent by FFPACHECK verifying the `ff_pac` rule, i.e., checking all the theory lemmas in the proof.

VII. Conclusion and Future Work

We presented a proof production and checking pipeline for the SMT theory of finite fields that, to our knowledge, is the first of its kind. We designed a proof calculus for CVC5's finite field decision procedure and instrumented the solver to emit certificates in it. We extended PACHECK and integrated it with CARCARA to check these certificates as part of complete proofs in the ALETHE format. We also extended LEAN-SMT with finite field support, providing SMT-based automation for modular-arithmetic goals in Lean 4. Our evaluation shows that proof production and external checking scale to the vast majority of a selection of problems from benchmarks presented in [20], with Lean reconstruction covering a smaller but substantial fraction of the runtime.

Potential future work on proofs for the SMT theory of finite fields could extend our calculus to the *split* Gröbner basis procedure [21]. Other future work could be making the decision procedure implemented in Yices [12] proof-producing. However, using our calculus for the latter would be challenging, since Yices' procedure does not rely on Gröbner Basis, using instead zero decomposition techniques.

Acknowledgements

This work was partially supported by a gift from Amazon Web Services; Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; the Stanford Center for Automated Reasoning; the Austrian Science Fund (FWF) [10.55776/ESP666]; and the Defense Advanced Research Projects Agency (DARPA) under contract FA8750-24-2-1001. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] John Abbott and Anna Maria Bigatt. CoCoALib: a C++ library for doing computations in commutative algebra. Available at https://cocoa.dima.unige.it/cocoa/cocoalib.

[2] Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. Carcara: An efficient proof checker and elaborator for SMT proofs in the Alethe format. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–386, Cham, 2023. Springer Nature Switzerland.

[3] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs (CPP)*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[4] Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, et al. Flexible proof production in an industrial-strength SMT solver. In *International Joint Conference on Automated Reasoning*, pages 15–35. Springer, 2022.

[5] Paul Beame, Russell Impagliazzo, Jan Krajíček, Toniann Pitassi, and Pavel Pudlák. Lower bounds on Hilbert's Nullstellensatz and propositional proofs. *Proceedings of the London Mathematical Society*, 3(1):1–26, 1996.

[6] E Ben-Sasson et al. Decentralized anonymous payments from bitcoin. security and privacy (sp). In *2014 IEEE Symposium*, pages 459–474, 2014.

[7] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C Paulson. Extending Sledgehammer with SMT solvers. *Journal of automated reasoning*, 51(1):109–128, 2013.

[8] Bruno Buchberger. A theoretical basis for the reduction of polynomials to canonical forms. *ACM SIGSAM Bulletin*, 10(3):19–29, 1976.

[9] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag.

[10] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. SMTCoq: A plug-in for integrating smt solvers into Coq. In *International Conference on Computer Aided Verification*, pages 126–133. Springer, 2017.

[11] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4255–4272, 2022.

[12] Thomas Hader, Daniela Kaufmann, Ahmed Irfan, Stéphane Graham-Lengrand, and Laura Kovács. MCSat-based finite field reasoning in the yices2 SMT solver (short paper). In *International Joint Conference on Automated Reasoning*, pages 386–395. Springer, 2024.

[13] John Harrison. Verifying nonlinear real formulas via sums of squares. In *International Conference on Theorem Proving in Higher Order Logics*, pages 102–118. Springer, 2007.

[14] Daira Hopwood, Sean Bowe, Taylor Hornby, Nathan Wilcox, et al. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, 4(220):32, 2016.

[15] Daniela Kaufmann, Mathias Fleury, Armin Biere, and Manuel Kauers. Practical algebraic calculus and Nullstellensatz with the checkers Pacheck and Pastèque and Nuss-Checker. *Formal Methods in System Design*, 64(1):73–107, 2024.

[16] Hanna Lachnitt, Mathias Fleury, Haniel Barbosa, Jibiana Jakpor, Bruno Andreotti, Andrew Reynolds, Hans-Jörg Schurr, Clark W. Barrett, and Cesare Tinelli. Improving the SMT proof reconstruction pipeline in Isabelle/HOL. In Yannick Forster and Chantal Keller, editors, *Interactive Theorem Proving (ITP)*, volume 352 of *LIPIcs*, pages 26:1–26:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[17] Wenda Li, Grant Olney Passer, and Lawrence C Paulson. Deciding univariate polynomial problems using untrusted certificates in Isabelle/HOL. *Journal of Automated Reasoning*, 62(1):69–91, 2019.

[18] Tomaz Mascarenhas, Harun Khan, Abdalrhman Mohamed, Andrew Reynolds, Haniel Barbosa, Clark Barrett, and Cesare Tinelli. Formalization of a proof calculus for incremental linearization for satisfiability modulo nonlinear arithmetic and transcendental functions. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 216–230, 2026.

[19] Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. lean-SMT: An SMT tactic for discharging proof goals in Lean. *arXiv preprint arXiv:2505.15796*, 2025.

[20] Alex Ozdemir, Gereon Kremer, Cesare Tinelli, and Clark Barrett. Satisfiability modulo finite fields. In *International Conference on Computer Aided Verification*, pages 163–186. Springer, 2023.

[21] Alex Ozdemir, Shankara Pailoor, Alp Bassa, Kostas Ferles, Clark Barrett, and Işil Dillig. Split Gröbner bases for satisfiability modulo finite fields. In *International Conference on Computer Aided Verification*, pages 3–25. Springer, 2024.

[22] Alex Ozdemir, Riad S Wahby, Fraser Brown, and Clark Barrett. Bounded verification for finite-field-blasting in a compiler for zero knowledge proofs. *Formal Methods in System Design*, pages 1–28, 2025.

[23] James Parker. zkLean: A DSL for ZK statement verification. https://www.galois.com/articles/zklean-a-dsl-for-zk-statement-verification, 2026. Galois Blog Post, accessed May 7, 2026.

[24] Andrew Reynolds, Hans-Jörg Schurr, Haniel Barbosa, Abdalrhman Mohamed, Aina Niemetz, Hanna Lachnitt, Jibiana Zoe Jakpor, Mathias Preiner, Ofec Israel, Yoni Zohar, Robert Jones, Clark Barrett, and Cesare Tinelli. The Cooperating Proof Calculus: Comprehensive proofs for an SMT solver. In Philipp Rümmer Anthony W. Lin, Eva Darulova, editor, *Computer Aided Verification (CAV)*. Springer, 2026. To appear.

[25] Andrew Reynolds, Hans-Jörg Schurr, Mallku Soldevila, Haniel Barbosa, Cesare Tinelli, and Clark Barrett. Ethos: A fast proof checker for the Eunoia logical framework. In Sara Negri Armin Biere, Carsten Lutz, editor, *Proceedings of the Joint Conference on Automated Reasoning (IJCAR)"*. Springer, 2026. To appear.

[26] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format. *arXiv preprint arXiv:2107.02354*, 2021.

[27] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In André Platzer and Geoff Sutcliffe, editors, *Proc. international Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 450–467. Springer, 2021.

[28] The FLINT team. *FLINT: Fast Library for Number Theory*, 2025. Version 3.4.0, https://flintlib.org.

[29] The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, New Orleans, LA, USA, January 2020. ACM.