An SMT-LIB Theory of Finite Fields

Thomas Hader[1], Alex Ozdemir[2]

[1]TU Wien, Favoritenstraße 9-11, 1040 Wien, Austria
[2]Stanford University; 353 Jane Stanford Way; Stanford, CA, 94305 USA

Abstract

In the last few years there have been rapid developments in SMT solving for finite fields. These include new decision procedures, new implementations of SMT theory solvers, and new software verifiers that rely on SMT solving for finite fields. To support interoperability in this emerging ecosystem, we propose the SMT-LIB theory of finite field arithmetic (FFA). The theory defines a canonical representation of finite field elements as well as definitions of operations and predicates on finite field elements.

1. Introduction

Finite fields are the basis for a large body of security-critical code. They are used in public-key cryptography: elliptic curves over finite fields are used in nearly all web browser connections for key exchange or digital signatures [1, 2, 3]. They are used in private-key cryptography: in both the Poly1305 message authentication code [4] and Galois counter mode (GCM) [5]. They are also the basis of most protocols for secure computation. For instance, many zero-knowledge proof systems prove and verify predicates expressed as finite field equations [6, 7, 8, 9]. Also, many secure multi-party computation protocols evaluate circuits over finite fields [10, 11]. Finally, some homomorphic encryption schemes apply to data in a finite field [12, 13].

The importance and prevalence of (finite-)field-based programs creates a need for tools that can formally verify them. Ideally, such tools would be partially or fully automated. The natural approach is *SMT-based verification*, as taken by prior tools like Dafny [14] and Boogie [15]. In this approach, a software *verifier* reduces the correctness of the program to logical formulas which it dispatches to a *satisfiability modulo theories* (SMT) solver. Applying this approach to field-based software generally requires an SMT solver that can solve finite field equations.

One way to solve field equations is by encoding them as integer equations, which many SMT solvers already comprehend. Consider (for the moment) a finite field of prime order p. Such a field is isomorphic to the integers $\{0, \ldots, p-1\}$ with addition and multiplication modulo p [16]. Thus, (non-linear) equations mod p can be encoded as (non-linear) integer equations. In this encoding, an equation $xy = z$ over field variables x, y, z would be encoded as $(x'y' - z') \bmod p = 0$, where x', y', z' are the integer representatives of the field variables. These equations can now be solved using an integer solver. Or, since all terms can be bounded, they can be solved as bit-vector (bounded integer) equations. However, prior experiments have shown that existing integer and bit-vector solvers perform poorly when given inputs that encode finite field arithmetic [17, 18].

To overcome the limitations of encoding, two direct SMT theory solvers for finite fields have recently emerged. The first is an MCSat [19] solver that is implemented in Yices [20, 21, 22, 23, 24]. The second is a CDCL(T) solver that is implemented in cvc5 [17, 25, 26]. Currently, these solvers accept field terms and equations expressed using a bespoke extension to SMT-LIB. This extension has not been standardized.

These SMT solvers have already enabled a variety of research projects and tools for automatically verifying systems that use zero-knowledge proofs (ZKPs). One project builds an automatically verifiable compiler pass for CirC: a compiler used with ZKPs [27]. Another builds a tool QED^2 that automatically verifies ZKP code in the Circom language [28, 29]. Another builds a tool for automatically verifying

SMT 2024: Satisfiability Modulo Theories, July 22–23, 2024, Montreal, Canada
thomas.hader@tuwien.ac.at (T. Hader); aozdemir@cs.stanford.edu (A. Ozdemir)

ZKP code written using the Halo2 library [30]. All of these projects use an SMT solver with finite field support.

Given the long-term importance of finite fields to security-critical software, the emergence of multiple SMT solvers with finite field support, and the emergence of multiple automatic verification tools expecting finite field support, we think the time is ripe to specify finite fields as an SMT-LIB theory. In this short paper, we do exactly that. In our specification, we consider all finite fields: those of prime order and their extensions. We consider fields of arbitrary size. Many cryptosystems require large fields (such as a prime order field with $p \approx 2^{256}$ or the binary extension field of order 2^{128}), but some can also operate over smaller fields (such as 32-bit or 64-bit fields) [31, 32].

Related Work There is already much work on verifying cryptographic implementations through *interactive* theorem proving and verification languages. Examples of secret-key and public-key cryptography include Fiat cryptography [33], Easycrypt [34], HACL* [35], and Jasmin [36]. There is also some work on interactive verification for ZKPs in the context of the Leo compiler [37], and by using refinement proofs [38, 39]. With better SMT-level support for finite fields, ITP proof automation for finite field lemmas could be improved through ITP-SMT bridges, like SMTCoq [40].

Further afield, some cryptographic implementations have been modeled and analyzed using *automated* symbolic analysis tools like Tamarin [41] and ProVerif [42]. The benefit of these tools is their high level of interpretability and automation, which allows them to be applied to protocols of realistic complexity, such as Signal [43]. However, they struggle to accurately model algebraic structures [44]. SMT-level algebraic reasoning would complement this research.

Another line of research develops SMT solvers for non-linear integer and real arithmetic using CDCL(T) [45, 46, 47, 48, 49, 50, 51, 52] and MCSat [53, 54, 55]. Some works specifically consider local search [56, 57, 58] and quantifier elimination [59, 60]. This research serves as good inspiration into techniques for finite field solving.

2. Background

We provide a brief summary of the relevant concepts of finite fields. A comprehensive overview can be found in [61, 62, 63] as well as in many other algebra textbooks.

Fields. A field $\mathbb{F}$ consists of a set of elements S on which the two binary operators *addition* "$+$" and *multiplication* "$\cdot$" are defined. S is closed under both operators, i.e. when applied on two elements of S, the result is in S. Both operators are commutative, associative, and have distinct neutral elements (denoted as *zero* (0) and *one* (1), respectively). Each element in S has an additive inverse element and all elements in $S \setminus \{0\}$ have a multiplicative inverse element. Further, distributivity of multiplication over addition holds. Informally speaking, a field is a set with well-defined operations addition, subtraction, multiplication, and division (with the exception of division by 0). Well known examples of fields are the rational number $\mathbb{Q}$ and the real numbers $\mathbb{R}$.

Finite Fields. A field $\mathbb{F}$ where S is finite is called a *finite field*.[1] The size of S is the *order* of $\mathbb{F}$. It has been shown that every finite field has order q that is a prime power $q = p^n$. We distinguish between *prime fields* with $n = 1$ and *extension fields* with $n > 1$. All fields of equal order are isomorphic (i.e. equivalent up to relabelling of elements), thus the field of order q is unique (up to isomorphism).

Prime Fields. The prime field of order p can be represented as $S = \{-\left\lfloor \frac{p-1}{2} \right\rfloor, \ldots, 0, 1, \ldots, \left\lfloor \frac{p}{2} \right\rfloor\}$[2] and is denoted $\mathbb{F}_p$. Let the function $\mathsf{smod}_p : \mathbb{Z} \to S$ be defined to map $z \in \mathbb{Z}$ to the unique element

[1] In honor of French mathematician Évariste Galois, finite fields are also called *Galois fields*.

[2] In the (isomorphic) representation $S = \{0, \ldots, p-1\}$, (with addition and multiplication modulo p), small "negative" values (such as -1) are instead represented as large positive values (such as $p-1$), which can be unintuitive to read. We choose our representation because small negative values are common in many applications.

of S that is equivalent to z, modulo p. The function is called "smod_p" because it outputs a *signed* representation of its input.

Addition and multiplication on S are defined by the usual integer operations followed by an application of smod_p. Due to the construction of S, finding the additive inverse is as simple as flipping the sign (assuming odd p).

Example 1. *The finite field $\mathbb{F}_5$ can be represented by the integers $\{-2, -1, 0, 1, 2\}$. In this representation of $\mathbb{F}_5$, $2 + 1 = -2$, $2 \cdot (-1) = -2$, and $(2 + 1) \cdot 2 = 1$ hold.*

Extension Fields. Let $\mathbb{F}_p[\alpha]$ be the set of univariate polynomials in variable α with coefficients from $\mathbb{F}_p$, and let $f \in \mathbb{F}_p[\alpha]$ have degree n and be irreducible (i.e. it cannot be represented as the product of two non-constant polynomials). The extension field of order p^n is denoted $\mathbb{F}_{p^n}$ and can be represented as polynomials in $\mathbb{F}_p[\alpha]$ of degree less than n, with (polynomial) addition and multiplication modulo f. Note that, in this representation, $\{0, 1\} \subseteq \mathbb{F}_p \subseteq \mathbb{F}_{p^n}$.

Example 2. *The finite field $\mathbb{F}_{3^2}$ is represented by the following polynomials of $\mathbb{F}_3[\alpha]$ modulo $\alpha^2 - \alpha - 1$:*

$$\{0,\ \alpha,\ \alpha + 1,\ -\alpha + 1,\ -1,\ -\alpha,\ -\alpha - 1,\ \alpha - 1,\ 1\}$$

Over $\mathbb{F}_{3^2}$ it holds that $(\alpha + 1) \cdot \alpha = (-\alpha + 1)$.

As the choice of f is not unique in general, different (isomorphic) representations of $\mathbb{F}_{p^n}$ exist, even if the representation of $\mathbb{F}_p$ is fixed. Note that no finite field is an *ordered field*. That is, there is no total ordering on S that is compatible with the field operations.

Conway Polynomials Algebraic tools have many choices for how to represent fields internally. But, to facilitate interoperability, the computer algebra community has agreed upon a canonical family of irreducible polynomials that should be used to represent elements of an extension field $\mathbb{F}_{p^n}$ in tool interfaces. These are called the *Conway polynomials* $C_{p,n} \in \mathbb{F}_p[\alpha]$, where p is a prime and $n > 1$. The precise definition of the Conway polynomials is not important for our purposes.[3] There is an algorithm for finding them [64] and they have been pre-computed for many n and p [65]. The Conway polynomials are used by all prominent computer algebra libraries: Sage, Magma, GAP, Singular, etc. We will use the Conway polynomials to define an SMT-LIB syntax for extension field element literals (Sec. 3.3).

Example 3. *The Conway polynomial $C_{3,2}$ is $\alpha^2 - \alpha - 1$, which is the irreducible used to represent $\mathbb{F}_9$ in Example 2.*

3. A Theory of Finite Fields

This section presents the SMT-LIB (version 2.6) *theory of finite field arithmetic* (FFA). Based on the theory of finite field arithmetic are the logics of quantifier-free finite field arithmetic `QF_FFA` as well as its quantified version `FFA`.

[3] The Conway polynomial $C_{p,n}$ is the lexicographically minimal monic primitive polynomial of degree n over $\mathbb{F}_p$ that is compatible with $C_{p,m}$ for all m dividing n. Let $r = (p^n - 1)/(p^m - 1)$ (which is an integer). Then, $C_{p,n} \in \mathbb{F}[\alpha]$ is compatible with $C_{p,m} \in \mathbb{F}[\alpha]$ if for every root $\alpha_0 \in \mathbb{F}_{p^n}$ of the former, α_0^r is a root of the latter. The lexicographic ordering used is also slightly non-standard. Define the alternating-sign coefficient representation of polynomial $f \in \mathbb{F}[\alpha]$ to be $f = \sum_{i=0}^{d} (-1)^i c_{d-i} \alpha^{d-i} = c_d \alpha^d - c_{d-1} \alpha^{d-1} + \cdots + (-1)^d a_0$, with $c_i \in \{0, \ldots, p-1\}$. Then, the order is lexicographic over the tuples $(c_d, \ldots, c_0)$.

3.1. The Finite Field Sorts

The theory of finite fields defines two kinds of finite field sorts, *prime field sorts* and *extension field sorts* for prime and extension fields, respectively (Sec. 2). They are represented by an indexed sort identifier of the form `(_ FiniteField` p`)` and `(_ FiniteField` p n`)` for prime and extension field sorts, respectively. The indexes p and n are numerals specifying the finite field order $q = p^n$. The index p must be a prime number in both cases. For extension field sorts, $n > 1$ must hold, as otherwise the resulting sort would be a prime field. Providing a non-prime number as p may result in unspecified solver behavior, although solvers are encouraged to report an error.[4] As is usual for an indexed sort, two finite field sorts with a different order are different sorts. Solvers implementing this theory are not required to support extension field sorts and may report an error in case an extension field sort is specified.[5]

For the rest of this chapter, a *finite field sort* is a prime or extension field sort with an arbitrary fixed order.

Example 4. *Set the logic to non-linear finite field arithmetic and define finite field sorts of size* 5 *and* 9*:*

```
(set-logic QF_FFA)
(define-sort FF5 () (_ FiniteField 5))
(define-sort FF9 () (_ FiniteField 3 2))
```

3.2. The Domain of Finite Field elements

A finite field of a given order is uniquely defined up to isomorphism. Thus, for the sake of defining an SMT theory for finite fields, a canonical representation for a finite field of a given order needs to be fixed. Otherwise different solvers might present the same model differently.

For a prime field with prime order p, the elements are represented by the integers of the set $\{-\left\lfloor \frac{p-1}{2} \right\rfloor, \ldots, \lfloor \frac{p}{2} \rfloor\}$. Operations are performed with regard to the function smod as defined in Section 2. For an extension field of order p^n the field elements are represented by univariate polynomials over the prime field of order p. The implied field is $\mathbb{F}[\alpha]/C_{p,n}$, where $C_{p,n}$ is the Conway polynomial (Sec. 2). All polynomial operations are performed modulo the Conway polynomial $C_{p,n}$.

3.3. Finite Field Literals

In the theory of finite fields, each element of a finite field sort is represented by a literal. To avoid confusion with the theories of integer and reals, finite field literals are prefixed with the string `ff`. We further say that a literal is *normalized* when it stands for an element from the fixed field representation as defined in Section 3.2. Non-normalized literals are allowed as an input and are mapped to the corresponding normalized literal, however, solvers are required to resort to normalized literals when printing a value.

Prime field literals. As stated in Section 2, elements of prime fields can be represented as integers modulo the field size. This property is used to define literals in the form of `ff`N, where N is an integer value. Given a prime field sort `(_ FiniteField` p`)` with prime order p, elements represented by the literals `ff`N for all values $N \in \{-\left\lfloor \frac{p-1}{2} \right\rfloor, \ldots, \lfloor \frac{p}{2} \rfloor\}$ are normalized. Every literal outside this set is mapped to the corresponding normalized literal by utilizing $\mathsf{smod}(N)$. Using this operation, the input gets mapped to the (unique) normalized representative of the same congruence class modulo p. In the

[4] We recommend that solvers test p's primality probabilistically, for example with a 40-repetition Miller-Rabin test [66]. If p is not prime, the solver can report an error. If p is prime or if the test is inconclusive, the solver may assume that p is prime and continue.

[5] We choose different syntaxes for prime fields and their extensions so that a user who is only interested in prime fields need not understand or even be aware of extension fields.

field type	order	syntax	syntax type
prime field	p	`(_ ff`N p`)`	indexed
		`ff`N`m`p	shorthand
		`(as ff`N `(_ FiniteField` p`))`	annotated
extension field	p^n	`(_ ff`$N.\cdots.N$ p n`)`	indexed
		`ff`$N.\cdots.N$`m`p`p`n	shorthand
		`(as ff`$N.\cdots.N$ `(_ FiniteField` p n`))`	annotated

Table 1
Different ways to define literals

presence of finite field theory, the user may not define their own symbols of form `ff`N (nor may they shadow other theory-defined symbols).

Example 5. *Given the defined sorts of Example 4, then examples of normalized elements of* `FF5` *are* `ff1`*,* `ff0`*, and* `ff-2`*. The (non-normalized) literals* `ff4` *and* `ff10` *describe the same element as the normalized literals* `ff-1` *and* `ff0`*, respectively.*

Extension field literals. Elements of extension field sorts are polynomials. Literals representing elements of extension field sorts are uniquely describing polynomials by specifying their coefficients. As described in Section 2, an element of a finite field of order $q = p^n$ with $n > 1$ is a polynomial $P \in \mathbb{F}_p[\alpha]$ of degree at most $n-1$. Let $P = c_0 + c_1\alpha^1 + \cdots + c_{n-1}\alpha^{n-1}$ where $c_i \in \mathbb{F}_p$, then the element P is represented by the literal `ff`$c_0.c_1.\cdots.c_{n-1}$ where all c_i are integers. Tailing zeros may be omitted, e.g. in `(_ FiniteField 3 6)` the literals `ff1.0.-1.0.0` and `ff1.0.-1` both represent the element $1-\alpha^2$. This further ensures that elements in $\mathbb{F}_p \subseteq \mathbb{F}_{p^n}$ have the same literal representation in both `(_ FiniteField` p`)` and `(_ FiniteField` p n`)`. A (polynomial) literal of `(_ FiniteField` p n`)` is normalized when all integer coefficients are normalized with regard to p and all tailing zeros are omitted. Specifying a literal with more than n coefficients is invalid.

Example 6. *Again, given the defined sorts of Example 4, normalized literals of* `FF9` *are all (normalized) literals of* `(_ FiniteField 3)`*:* `ff0`*,* `ff1`*, and* `ff-1`*, as well as literals representing the further (polynomial) elements of* $\mathbb{F}_{3^2}$*, for example* `ff-1.1`*,* `ff0.1`*, and* `ff-1.-1` *representing the elements* $\alpha - 1$*,* α*, and* $-\alpha - 1$*, respectively. Note that* `ff2.1` *and* `ff1.0` *are both non-normalized versions of* `ff-1.1` *and* `ff1`*, respectively.*

Well-Sortedness of literals. Since the defined literals are overloaded (for example, every finite field has a one element, thus `ff1` is of undetermined sort), the order of the literal's field must be specified in order to satisfy the well-sortedness requirements of SMT-LIB. There are three ways of specifying the sort of a finite field literal. (i) By indexing the literal `(_ ff...` p`)` and `(_ ff...` p n`)` with the finite field order p and p^n, respectively. This allows the literal's sort to be derived, as the order of the finite field specifies the sort uniquely. (ii) Similar to the theory of bit-vectors, there is a shorthand to avoid the _ keyword and specify the sort as part of the literal symbol. This is done by appending the literal with `m`p and `m`p`p`n for prime field sort of order p and extension field sort of order p^n, respectively.[6] (iii) Since p might be a large number, a short-cut is provided by `(as ff...` S`)` where S is a finite field sort. Using the `define-sort` command, one can assign a symbol to a finite field sort to be used for all its literals. Table 1 gives an overview of all three variants. When printing finite field elements, solvers are free to choose between the first two notations.

Example 7. *The expressions* `(_ ff1 5)` *and* `(_ ff1 3 2)` *denote the multiplicative identity (one) of* $\mathbb{F}_5$ *and* $\mathbb{F}_{3^2}$*, respectively. In the shorthand notation, the same literals can be written as* `ff1m5` *and* `ff1m3p2`*. Using the defined sorts from Example 4,* `(as ff1 FF5)` *and* `(as ff1 FF9)` *can be used alternatively.*

[6] Here, "`m`" denotes *modulo* and "p" denotes *power*. But, note that $\mathbb{F}_{p^n}$ and $\mathbb{Z}_{p^n}$ are not isomorphic for $n > 1$.

3.4. Finite Field Operations

All of the following operator definitions represent well known semantics from algebra. Thus, an explicit definition of their semantics is omitted and we refer to Section 2 for further details. Operations always operate on one specific finite field sort S, i.e. all parameters have sort S and an element of S is returned. All functions are defined for all prime field sorts as well as all extension field sorts. For the sake of brevity, the extension field sort variants of the functions are omitted. Table 2 gives an overview of all operations.

Binary arithmetic. For each finite field order, we define operations that take two finite field elements of one finite field sort and return an element of the same sort. Given two inputs, the operations represent sum, difference, product, and quotient, respectively.[7]

```
(ff.add (_ FiniteField p) (_ FiniteField p) (_ FiniteField p) :left-assoc)
(ff.sub (_ FiniteField p) (_ FiniteField p) (_ FiniteField p))
(ff.mul (_ FiniteField p) (_ FiniteField p) (_ FiniteField p) :left-assoc)
(ff.div (_ FiniteField p) (_ FiniteField p) (_ FiniteField p))
```

As hinted by the `:left-assoc` keyword, occurrences of `ff.add` and `ff.mul` may contain more than two arguments and multiple arguments are grouped left associatively. However, note that both operations are associative anyway.[8]

Unary arithmetic. For each finite field sort, there are the following unary operations:

```
(ff.neg (_ FiniteField p) (_ FiniteField p))
(ff.recip (_ FiniteField p) (_ FiniteField p))
```

Here, `ff.neg` returns the unary negation (usually written as $-x$ for an element x), i.e. the inverse element with regard to addition. The operation `ff.recip` returns the reciprocal value (usually written as x^{-1} for an element x), i.e. the inverse element with regard to multiplication. Note that `ff.recip` has total semantics but is unspecified for the zero element.

Division by zero. Two operators (`ff.div`, `ff.recip`) represent mathematical operations with only partial semantics. Mathematically speaking, division by zero is undefined, and computing the reciprocal of zero is undefined. Yet, SMT-LIB requires functions to have total semantics. We require solvers to interpret the reciprocal of zero as zero. Moreover dividing any value by zero gives zero. This choice is somewhat arbitrary. It is acceptable because it easy for solvers to meet and for verification tools to use.

A solver can meet this requirement using a preprocessing transformation. First, it encodes division as multiplication by the divisor's reciprocal. Second, it encodes the reciprocal relation $z =$ `(ff.recip` x`)` by the following (reciprocal-free) formula:

$$[(x \neq 0) \wedge (xz = 1)] \vee [(x = 0) \wedge (z = 0)]$$

This ensures that 0's reciprocal is 0. The are other encodings of reciprocal that do not explicitly contain a disjunction. For example, as $(zzx = z) \wedge (zxx = x)$.

This requirement is also easy for verification tools that use SMT to work with. In particular, a verification tool can create an SMT query where division or reciprocal have different semantics by wrapping them with an if-then-else term that implements those semantics when the appropriate input is zero.

[7] Since `ff.neg` and `ff.recip` are defined as well, `ff.sub` and `ff.div` are redundant. However, we believe that all common mathematical operations should have operations in the finite field theory. Furthermore, other arithmetic theories in SMT-LIB also define redundant subtraction and division operators.

[8] Note that `ff.div` is not Euclidean division, rather it is multiplication by an inverse in the field. Thus, the remainder of a division, i.e. `ff.rem`, would not be meaningful.

Identifier	Sort	Meaning
`ff.add`	$F \times F \to F$	finite field addition
`ff.sub`	$F \times F \to F$	finite field subtraction
`ff.mul`	$F \times F \to F$	finite field multiplication
`ff.div`	$F \times F \to F$	finite field division
`ff.neg`	$F \to F$	finite field negation
`ff.recip`	$F \to F$	finite field reciprocal

Table 2
A summary of finite field operations for a finite field type F.

3.5. Comparison

Since finite fields are not ordered, the theory of finite fields only supports the equality predicate:

```
(= (_ FiniteField p) (_ FiniteField p))
(= (_ FiniteField p n) (_ FiniteField p n))
```

Example 8. *Continuing with the definition of Example 4. First define some variables:*

```
(declare-fun x0 () FF5)
(declare-fun x1 () FF5)
(declare-fun x2 () FF5)
```

Then add some assertions:

```
(assert (= (ff.mul x1 x2) (ff.add x1 x2)))
(assert (= (ff.recip x1) x0))
(assert (= (ff.sub x2 x0)) (as ff1 FF5))
```

This encodes the constraint system in $\mathbb{F}_5$*:*

$$\begin{aligned} x_1 x_2 &= x_1 + x_2 \\ x_1^{-1} &= x_0 \\ x_2 - x_0 &= 1 \end{aligned}$$

4. Existing Finite Field Solvers

There are two existing SMT solvers that support the theory of finite fields: Yices [23] and cvc5 [26]. Both support the logic of quantifier-free finite field arithmetic QF_FFA for prime fields as defined in this paper.

- **Yices2** implements reasoning over prime fields using its MCSat engine [24]. This implementation is based on the approach by Hader et al. [22]. Processing of polynomials over prime fields is done done using an updated version of the LibPoly library [67].
- **cvc5**'s prime field solver is a CDCL($\mathcal{T}$) theory solver that implements two decision procedures designed by Ozdemir et al. The first procedure is based on Gröbner bases and triangular decomposition [17]. The second is based on the same algebraic ideas, but uses multiple, small Gröbner bases for better scalability in some cases [25]. The implementation uses the CoCoALib computer algebra library [68].

5. Future Directions

In designing our theory, we have intentionally omitted a number of potential features. Some of these features might be good additions in the future. We discuss two such features here, together with why they might be useful.

Conversions In this proposal, we do not give operations for converting between finite field elements and other discrete arithmetic types, such as integers and bit-vectors. This might be useful for verification problems about code that converts between these types. For example, the AES-GCM block cipher alternates between treating its data as bit-vectors and elements of $\mathbb{F}_{2^n}$, to perform different kinds of operations on that data. The bit-vector representation is used for the AES permutation and the field representation is used in the GCM message authentication code. Another example is an implementation of $\mathbb{F}_p$ arithmetic on a b-bit CPU, where $2^b \ll p$. Such an implementation is defined by bit-vector arithmetic, but the specification is an equation in $\mathbb{F}_p$. Thus, giving a natural statement of the implementation's correctness requires operations to convert between $\mathbb{F}_p$ and bit-vectors. Since some SMT solvers already allow conversions between bit-vectors and integers, conversions between integers and finite field elements might suffice.

Another kind of conversion which might be useful is one between a field $\mathbb{F}_{p^n}$ and some extension $\mathbb{F}_{(p^n)^e}$ of it (for $e > 1$).

Variable-sized fields In this proposal, we consider only fields of fixed size. This bars the possibility of queries that verify a property that holds generically for many or all fields. Such properties arise naturally in many verification problems. For instance, one might have a function that implements some finite-field operation in which the size of the field is an input to the function. To verify the function for all fields, one might construct a logical formula in which the field size is a variable.

Acknowledgements. We thank Ahmed Irfan, Alp Bassa, Clark Barrett, Daniela Kaufmann, Gereon Kremer, Shankara Pailoor, Sorawee Porncharoenwase, and the SMT'24 reviewers for valuable discussion and feedback. We further thank Stéphane Graham-Lengrand for hosting the first author for a research stay at SRI during which the idea for this work initiated. We acknowledge funding from the TU Wien SecInt Doctoral College, NSF grant number 2110397, the Stanford Center for Automated Reasoning, and the Simons Foundation.

References

[1] E. Barker, L. Chen, A. Roginsky, A. Vassilev, R. Davis, Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography, 2018. https://doi.org/10.6028/NIST.SP.800-56Ar3.

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

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

[4] D. J. Bernstein, The Poly1305-AES message-authentication code, in: FSE, 2005.

[5] J. Salowey, A. Choudhury, D. McGrew, Rfc 5288: AES galois counter mode (GCM) cipher suites for TLS, 2008.

[6] S. Goldwasser, S. Micali, C. Rackoff, The knowledge complexity of interactive proof-systems (extended abstract), in: STOC, 1985.

[7] B. Parno, J. Howell, C. Gentry, M. Raykova, Pinocchio: Nearly practical verifiable computation, Communications of the ACM (2016).

[8] J. Groth, On the size of pairing-based non-interactive arguments, in: EUROCRYPT, 2016.

[9] S. Chaliasos, J. Ernstberger, D. Theodore, D. Wong, M. Jahanara, B. Livshits, SoK: What don't we know? understanding security vulnerabilities in SNARKs, arXiv preprint arXiv:2402.15293 (2024).

[10] I. Damgård, V. Pastro, N. Smart, S. Zakarias, Multiparty computation from somewhat homomorphic encryption, in: CRYPTO, 2012.

[11] M. Hastings, B. Hemenway, D. Noble, S. Zdancewic, SoK: General purpose compilers for secure multi-party computation, in: IEEE S&P, 2019.

[12] O. Regev, On lattices, learning with errors, random linear codes, and cryptography, J. ACM (2009).

[13] A. Viand, P. Jattke, A. Hithnawi, SoK: Fully homomorphic encryption compilers, in: IEEE S&P, 2021.

[14] K. R. M. Leino, Dafny: An automatic program verifier for functional correctness, in: LPAR, 2010.

[15] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino, Boogie: A modular reusable verifier for object-oriented programs, in: Formal Methods for Components and Objects, 2006.

[16] D. S. Dummit, R. M. Foote, Abstract algebra, volume 3, Wiley Hoboken, 2004.

[17] A. Ozdemir, G. Kremer, C. Tinelli, C. Barrett, Satisfiability modulo finite fields, in: CAV, 2023.

[18] A. Niemetz, M. Preiner, Y. Zohar, Scalable bit-blasting with abstractions, in: CAV, 2024.

[19] D. Jovanovic, C. Barrett, L. De Moura, The design and implementation of the model constructing satisfiability calculus, in: FMCAD, 2013.

[20] T. Hader, Non-linear SMT-reasoning over finite fields, 2022. MSc Thesis (TU Wien).

[21] T. Hader, L. Kovács, Non-linear SMT-reasoning over finite fields, in: SMT, 2022. URL: http://ceur-ws.org/Vol-3185/extended3245.pdf, extended Abstract.

[22] T. Hader, D. Kaufmann, L. Kovács, SMT solving over finite field arithmetic, in: LPAR, 2023.

[23] B. Dutertre, Yices 2.2, in: CAV, 2014.

[24] T. Hader, D. Kaufmann, A. Irfan, S. Graham-Lengrand, L. Kovács, MCSat-based Finite Field Reasoning in the Yices2 SMT Solver, 2024. arXiv:2402.17927.

[25] A. Ozdemir, S. Pailoor, A. Bassa, K. Ferles, C. Barrett, I. Dillig, Split Gröbner bases for satisfiability modulo finite fields, 2024. https://ia.cr/2024/572.

[26] 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, Y. Zohar, cvc5: A versatile and industrial-strength SMT solver, in: TACAS, 2022.

[27] A. Ozdemir, R. S. Wahby, F. Brown, C. Barrett, Bounded verification for finite-field-blasting, in: CAV, 2023.

[28] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez, J. Van Geffen, J. Morton, M. Chu, B. Gu, Y. Feng, I. Dillig, Automated detection of under-constrained circuits in zero-knowledge proofs, in: PLDI, 2023.

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

[30] F. H. Soureshjani, M. Hall-Andersen, M. Jahanara, J. Kam, J. Gorzny, M. Ahmadvand, Automated analysis of Halo2 circuits, 2023. https://ia.cr/2023/1051.

[31] P. Zero, Plonky2: Fast recursive arguments with Plonk and FRI, 2022. https://github.com/0xPolygonZero/plonky2/blob/main/plonky2/plonky2.pdf.

[32] E. Ben-Sasson, I. Bentov, Y. Horesh, M. Riabzev, Fast reed-solomon interactive oracle proofs of proximity, in: ICALP, 2018.

[33] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, A. Chlipala, Simple high-level code for cryptographic arithmetic – with proofs, without compromises (2020).

[34] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, P.-Y. Strub, Easycrypt: A tutorial, International School on Foundations of Security Analysis and Design (2012) 146–166.

[35] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, B. Beurdouche, HACL*: A verified modern cryptographic library, in: CCS, 2017.

[36] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, P.-Y. Strub, Jasmin: High-assurance and high-speed cryptography, in: CCS, 2017.

[37] A. Coglio, E. McCarthy, E. W. Smith, Formal verification of zero-knowledge circuits, 2023. URL: http://dx.doi.org/10.4204/EPTCS.393.9.

[38] J. Liu, I. Kretz, H. Liu, B. Tan, J. Wang, Y. Sun, L. Pearson, A. Miltner, I. Dillig, Y. Feng, Certifying zero-knowledge circuits with refinement types, arXiv preprint arXiv:2304.07648 (2023).

[39] K. Jiang, D. Chait-Roth, Z. DeStefano, M. Walfish, T. Wies, Less is more: refinement proofs for probabilistic proofs, in: IEEE S&P, 2023.

[40] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, C. Barrett, SMTCoq: A plug-in for integrating SMT solvers into Coq, in: CAV, 2017.

[41] S. Meier, B. Schmidt, C. Cremers, D. Basin, The TAMARIN prover for the symbolic analysis of security protocols, in: Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25, Springer, 2013, pp. 696–701.

[42] B. Blanchet, B. Smyth, V. Cheval, M. Sylvestre, ProVerif 2.00: automatic cryptographic protocol verifier, user manual and tutorial (2018).

[43] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, D. Stebila, A formal security analysis of the Signal messaging protocol, Journal of Cryptology 33 (2020) 1914–1983.

[44] C. Cremers, D. Jackson, Prime, order please! revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman, in: CSF, 2019.

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

[46] A. Cimatti, A. Griggio, A. Irfan, M. Roveri, R. Sebastiani, Experimenting on solving nonlinear integer arithmetic with incremental linearization, in: International Conference on Theory and Applications of Satisfiability Testing, Springer, 2018, pp. 383–398.

[47] A. Maréchal, A. Fouilhé, T. King, D. Monniaux, M. Périn, Polyhedral approximation of multivariate polynomials using Handelman's theorem, in: VMCAI, 2016.

[48] M. Fränzle, C. Herde, T. Teige, S. Ratschan, T. Schubert, Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure, Journal on Satisfiability, Boolean Modeling and Computation 1 (2006).

[49] V. X. Tung, T. V. Khanh, M. Ogawa, raSAT: An SMT solver for polynomial constraints, in: IJCAR, 2016.
[50] D. Jovanović, L. d. Moura, Cutting to the chase solving linear integer arithmetic, in: CADE, 2011.
[51] I. Dillig, T. Dillig, A. Aiken, Cuts from proofs: A complete and practical technique for solving linear inequalities over integers, 2009, pp. 233–247.
[52] F. Corzilius, G. Kremer, S. Junges, S. Schupp, E. Ábrahám, SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving, in: SAT, 2015.
[53] D. Jovanović, L. De Moura, Solving non-linear arithmetic, ACM Communications in Computer Algebra 46 (2013).
[54] D. Jovanović, Solving nonlinear integer arithmetic with MCSAT, in: VMCAI, 2017.
[55] L. d. Moura, D. Jovanović, A model-constructing satisfiability calculus, in: VMCAI, 2013.
[56] S. Cai, B. Li, X. Zhang, Local search for satisfiability modulo integer arithmetic theories, ACM Transactions on Computational Logic (2023).
[57] X. Zhang, B. Li, S. Cai, Deep combination of CDCL (T) and local search for satisfiability modulo non-linear integer arithmetic theory, 2024.
[58] Z. Wang, B. Zhan, B. Li, S. Cai, Efficient local search for nonlinear real arithmetic, in: VMCAI, 2023.
[59] B. F. Caviness, J. R. Johnson, Quantifier elimination and cylindrical algebraic decomposition, Springer Science & Business Media, 1998.
[60] V. Weispfenning, Quantifier elimination for real algebra—the quadratic case and beyond, Applicable Algebra in Engineering, Communication and Computing 8 (1997).
[61] R. Lidl, H. Niederreiter, Introduction to finite fields and their applications, Cambridge university press, 1994.
[62] R. J. McEliece, Finite fields for computer scientists and engineers, volume 23, Springer Science & Business Media, 2012.
[63] J. A. Gallian, Contemporary Abstract Algebra, Chapman and Hall/CRC, 2021.
[64] L. S. Heath, N. A. Loehr, New algorithms for generating conway polynomials over finite fields, Journal of Symbolic Computation 38 (2004) 1003–1024.
[65] F. Lübeck, Conway polynomials for finite fields, ???? Pre-computed Conway polynomials. Available at https://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol/index.html or https://github.com/sagemath/conway-polynomials.
[66] M. O. Rabin, Probabilistic algorithm for testing primality, Journal of number theory 12 (1980) 128–138.
[67] D. Jovanovic, B. Dutertre, Libpoly: A library for reasoning about polynomials, in: Intl. Workshop on Satisfiability Modulo Theories (SMT), CEUR Workshop Proceedings, 2017.
[68] J. Abbott, A. M. Bigatti, Gröbner bases for everyone with CoCoA-5 and CoCoALib, in: The 50th Anniversary of Gröbner Bases, volume 77, Mathematical Society of Japan, 2018, pp. 1–25.