Formalizing Soundness Proofs of Linear PCP SNARKs

Bolton Bailey and Andrew Miller, *University of Illinois at Urbana-Champaign*

https://www.usenix.org/conference/usenixsecurity24/presentation/bailey

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

Formalizing Soundness Proofs of Linear PCP SNARKs

Bolton Bailey
University of Illinois at Urbana-Champaign

Andrew Miller
University of Illinois at Urbana-Champaign

Abstract

Succinct Non-interactive Arguments of Knowledge (SNARKs) have seen interest and development from the cryptographic community over recent years, and there are now constructions with very small proof size designed to work well in practice. A SNARK protocol can only be widely accepted as secure, however, if a rigorous proof of its security properties has been vetted by the community. Even then, it is sometimes the case that these security proofs are flawed, and it is then necessary for further research to identify these flaws and correct the record [39, 58].

To increase the rigor of these proofs, we create a formal framework in the Lean theorem prover for representing a widespread subclass of SNARKs based on linear PCPs. We then describe a decision procedure for checking the soundness of SNARKs in this class. We program this procedure and use it to formalize the soundness proof of several different SNARK constructions, including the well-known Groth '16.

1 Introduction

Over the past decade, cryptographic research has produced Succinct Non-interactive Arguments of Knowledge (SNARKs), which allow a prover to demonstrate knowledge of a witness corresponding to a statement in some NP relation. There exist schemes for instantiating SNARKs [42,43,59] that produce proofs consisting of a $O(1)$-sized message. SNARKs promise to have many applications in verifiable computation [16], blockchains [19, 61], and identity management [50].

Since practitioners are applying this new technology to tasks where security is a main concern, it is important for them to have confidence in the cryptographic properties of the protocols they implement. To this end, the academic review process ensures that SNARKs published in the literature come with mathematical proofs of security properties. But in practice, errors arise. Research from Parno [58] and Gabizon [39] has identified flaws in the soundness property of SNARKs in the `libsnark` library originating from [17], which was a modification of the Pinocchio SNARK [59]. Soundness refers to the property of a SNARK protocol that makes it infeasible for a malicious prover to construct proofs of false statements. This type of problem is potentially very serious – Compromising the soundness can lead to a total breakdown of the security assumptions of whatever system uses the SNARK, potentially without an outward sign that anything is wrong.

To prevent errors like this from happening in the future, we look to apply formal methods to guarantee soundness. We focus on the class of "pairing-based" SNARKs which achieve the fully-succinct $O(1)$ proof-message size previously mentioned. In particular, we deal with linear PCP SNARKs that work in the structured reference string (SRS) model: All parties are assumed to have access to a collection of elements of a pairing-friendly cryptographic group, generated by some trusted third party.

1.1 Our Contribution

Our main contribution consists of formalized proofs about the soundness aspect of several SNARK protocols. This formalization effort can be broken down into three parts:

1. We observe that linear PCP SNARK protocols from the literature have a few regularities that make them amenable to formal analysis. We call SNARKs with these properties "Straightforward linear PCP SNARK schemes". Formally, this is a restriction of the class of linear PCP SNARKs, but every linear PCP SNARK scheme that we are aware of belongs to this class.
2. We develop a mathematical structure to represent instantiations of linear PCP SNARKs, and we formalize this structure in the Lean Theorem Prover [31]. This structure is capable of representing any linear PCP SNARK, but is particularly well-suited to instantiations derived from Straightforward schemes. We formalize six SNARK schemes from the literature in Lean with this structure.
3. We automate a procedure to decide the soundness of straightforward linear PCPs SNARK schemes. That is,

our procedure is given a straightforward linear PCP SNARK scheme, and either produces a proof that every instantiation of a SNARK from that scheme will be sound, or fails to do so, in which case it is possible to attack the soundness of some instance. We run this decision procedure on the six SNARK schemes to verify their soundness.

The six protocols we formalize are of some of the most well-cited and widely-deployed in practice. They are:

- **GGPR** [42]: The first SNARK to encode computations using Quadratic Arithmetic Programs (QAPs), which is the main NP-Complete language to which SNARKs are reduced in this line of work.
- **Pinocchio** [59]: A modification of GGPR which improved efficiency.
- **Groth '16** [43]: A widely-cited SNARK which reduced the proof size to only three group elements.
- **Baghery et al.** [6]: A paper which presented a version of Groth '16 for Type-III pairings.
- **Lipmaa** [47]: Another version of Groth '16, redesigned with an eye to a simulation-extractability property.
- **Baby SNARK** [55]: A simplified version of Groth '16 constructed for educational purposes, designed to have an easy-to-follow soundness proof.

Our work puts the soundness of these SNARKs beyond doubt. Additionally, we identified and formalized a variety of techniques for manipulating constructions such as these, in order to make our work more extensible.

1.2 The Lean Theorem Prover

We carry out our formalization in the Lean 4 Theorem Prover [31]. Lean allows a user to write code that encodes mathematical theorems and proofs in the language of dependent type theory. All proofs that the programmer writes in this language are converted into a machine-readable form which is then checked by the Lean kernel.

The soundness proofs we complete involve reasoning about equalities of multivariate polynomials over finite fields. We base our work on `mathlib` [51], the open-source Lean mathematical library that implements structures and lemmas from much of undergraduate-level mathematics, including finite fields and multivariate polynomials.

Lean is built with a metaprogramming facility which allows the user to write "tactic" code which automatically constructs proofs. We use this to implement a recursive tactic to resolve subgoals consisting of systems of equations over a integral domain. Additionally, we make use of the builtin `simp` tactic, which invokes Lean's simplifier – a proof-producing procedure which allows a user to specify lemmas that are then iteratively applied to simplify an expression. We construct a variety of Lean simplification attributes for normalizing statements about polynomials and their coefficients. The end product is a system which is capable of automatically constructing a proof of the knowledge-soundness theorem for a SNARK.

1.3 Related Work

Cryptography in general and proof systems in particular have been of great interest to the formal verification community. In this section, we will go over some of these contributions and their relevance to the problem at hand.

Lean itself is a somewhat rare choice for formalization of cryptographic protocols. We nevertheless considered it appropriate for our work here, for a few reasons: The comprehensive and well-integrated `mathlib` library and its implementation of numerous lemmas about multivariate polynomials makes Lean ideal in the setting of succinct proofs. Avigad et al. [4] also use Lean and `mathlib` and theirs is the only other work of which we are aware that formalizes statements about a general-purpose succinct proof system: Their development provides a full formalization of the Cairo virtual machine in Lean, with a proof of *correctness* for its execution, (rather than soundness). It is encouraging to see another aspect of a proof system formalized, and it suggests that a full proof of completeness and soundness of some system could be within grasp. Extensions to Lean could make it a better basis for cryptography in the future - In the time our project has been underway, `mathlib` has added an implementation of elliptic curves [3], as well as a tactic [18] for solving Gröbner basis problems. This tactic, `polyrith`, works by calling a web interface to a Sage Gröbner-basis solver - while the problems that our code generates are too large to be handled on their own by `polyrith`, it is promising that a more general version of the problem is being worked on, and in the future it could make our system more flexible.

Coq [46] is the theorem proving language perhaps most related to Lean - both languages are based on dependent type theory. The Certicrypt Coq library [11] provides tools for developing formalizations of cryptographic protocols. Highly relevant is Fournet et al.'s work on a certified compiler for the Pinocchio proof system [37]. This work formally verifies a SNARK "front-end" - code that is necessary to compile a high-level language (in this case C) to the QAP. Also relevant are [5], which formalizes the portion of the zero-knowledge stack which compiles relations into the necessary form to be handled by a SNARK, and [12], which uses Certicrypt to formalize Σ-protocols, a class of proof systems involving three rounds of communication. There have also been several applications of Coq to other (non-proof-system) cryptographic protocols, including Proof-of-Stake consensus systems [66],

mix nets [45] and signature schemes [65]. Efforts in Coq to formalize broader classes of cryptographic techniques include SSProve [1], which formalizes a modularization of cryptographic proofs, the formalization [57] of some of the number-theoretic underpinnings of cryptography, and [9], which formalizes the generic model for group-based cryptography and random oracle model. This last is relevant to our own application - the generic group model [63] is related to the algebraic group model that our own work uses, in that both seek to codify an adversary's interaction with a cryptographic group. External to cryptography, Coq also has well developed math libraries, including the [60] tactic for Gröbner bases - the decision to use Lean over Coq for this project comes down to a few matters of convenience, such as Lean's ability to express tactics themselves in Lean.

EasyCrypt is another proof assistant. Like Coq, it is written in OCaml, but EasyCrypt is designed specifically with cryptography in mind. EasyCrypt allows one to reason about probabilistic computation, which is convenient for the formalization of game-based cryptographic proofs. Works in EasyCrypt relevant to proof systems include [2], which formalizes the "MPC-in-the-head" paradigm for zero knowledge and [36], which formalizes a variety of protocols including protocols for proof of knowledge of quadratic residues, discrete logarithms, and Hamiltonian cycles. The first two of the latter are protocols for specific problems not known to be NP-complete, so they cannot be turned into general-purpose proof systems. The Hamiltonian cycle and MPC-in-the-Head approaches are general purpose, but these proof systems are not succinct - they require the verifier to do work linear or more in the problem. EasyCrypt has also been used for many applications beyond proof systems, including verifications of multi-party computation itself [34,44], post-quantum cryptography [8], Pedersen commitments [54], electronic voting [30], and key exchange [10]. The framework has also formalized some more general techniques, including Canetti's Universal Composability framework [24] in [25] and Brzuska et al.'s State separating proofs [20] in [33].

On the other end of the spectrum from strongly-typed languages like Lean and Coq is the Isabelle [56] ecosystem, with its CryptHOL [14] cryptographic framework. Butler et al. [23] have done work to formalize Σ-protocols using this system, and there are also modules for constructive cryptography [13,48], and oblivious transfer [22]. ACL2, a Lisp-based theorem prover, has been used to verify the Ethereum Recursive Length Prefix encoding scheme [29].

It is also worth mentioning that there is some work done in the space of verification for proof systems which does not prove theorems formally, but nevertheless uses automated processes to check the construction of protocols. These often focus on the circuit compilation component of the SNARK toolchain: Ecne [67] is a Julia project which analyzes Rank-1 Constraint Systems to determine if their outputs are uniquely determined. Picus [26] is a symbolic VM for formal verification of Rank-1 Constraint Systems. [28] describes the Leo language, a DSL for writing SNARK programs with a facility for ACL2 verification.

To summarize, formal modelling of cryptography, like cryptography itself, is diverse both in its scope and in its techniques. For more in-depth surveys, the reader can consult the Systematization of Knowledge papers of [7], [53] or [64]

2 Linear PCP SNARKs

2.1 Overview of Elliptic Curve Pairings and the Algebraic Group Model

In the SNARKs we analyze, every message transmitted during the protocol comes in the form of a collection of elliptic curve group elements. These elements come from one of three predefined elliptic curves G_1, G_2, G_T, each of prime order p. The three curves admit a *pairing* operation: That is, there is an efficiently-computable nontrivial operation $e : G_1 \times G_2 \to G_T$ which satisfies the bilinearity property: For any $g \in G_1, h \in G_2, a, b \in \mathbb{F}_p$, we have

$$e(g^a, h^b) = e(g,h)^{a \cdot b}.$$

Fixing some generators g, h of the first two groups G_1, G_2, and fixing $e(g,h)$ as a generator of G_T, any value computed by any algorithm in the protocol can be expressed as a power of one of these generators. In keeping with the literature, we will often express group elements by corresponding power in $\mathbb{F}_p$ used to obtain it from the corresponding generator.

In order to formally check the soundness of a SNARK construction, we must make a cryptographic assumption that limits what the prover is capable of doing in assembling the proof-message, and as is typical with cryptography done using elliptic curves, our assumption is related to the difficulty of evaluating discrete logarithms in the three groups. In particular, we formalize the soundness of our chosen SNARKs with the *Algebraic Group Model* assumption [38], or AGM. The AGM assumes that the prover can only carry out algebraic operations on group elements that come from the above operations. That is, a prover can carry out group operations (multiplication and exponentiation) within any one of the three groups G_1, G_2, G_T and can use the pairing operation to take an element from each of the groups G_1, G_2 and obtain an element of G_T, but can do nothing else in the way of operations on the group elements it holds. A critical consequence of this assumption is that any element of the G_1 or G_2 curves that the prover outputs must necessarily be a linear combination of elements from those curves that the prover has seen before.

Definition 1 (See [6]). *An algorithm $\mathcal{A} : W \times G_b^k \to G_b^n$ is algebraic if there is a polynomial-time $\mathcal{X} : G_i^k \times G_b^n \to \mathbb{F}_p^{k \times n}$ taking the inputs and outputs of $\mathcal{A}$ and returning an $k \times$*

n matrix of field elements such that, except with negligible probability,

$$\mathcal{A}_i(w, g^{n_1}, \cdots, g^{n_k}) = g^{\sum_{j \in n} n_j \mathcal{X}(g^{n_1}, \cdots, g^{n_k}, \mathcal{A}_i(w, g^{n_1}, \cdots, g^{n_k}))_{i,j}}.$$

2.2 SNARKs

As we have mentioned, we focus on soundness proofs for pairing-based SNARKs in the structured reference string model. In particular, we focus on *knowledge soundness*. This refers to a guarantee that a prover who consistently convinces a verifier of a statement can only do so because they possess knowledge of a witness corresponding to that statement.

A typical approach in the field of interactive cryptographic proof systems is to prove this knowledge by defining an *extractor* algorithm. This algorithm has access to the prover algorithm, and can run it (potentially multiple times with different inputs) to extract the witness. This paradigm is challenging for SNARKs, though, because their non-interactive nature limits the information the extractor can access when trying to recover a witness.

To get around this, the SNARKs we formalize construct proofs from a larger piece of data, called a structured reference string (SRS, sometimes also referred to as a common reference string or CRS), the prover's interaction with which is the basis for the extractor. An SRS is produced before the proof phase of the protocol by a trusted third party. This leads us the usual definition for a Non interactive Zero-knowledge argument, (e.g. as found in [43])

Definition 2. *A Non-interactive argument for a relation R is a triple of randomized PPT algorithms* (SRS, P, V) *where:*

- *The SRS algorithm takes no input and generates a structured reference string* $\sigma \leftarrow SRS()$
- *The prover takes a statement x and witness w for it in the relation* $(x, w) \in R$ *as well as the SRS and returns a proof* $\pi \leftarrow P(\sigma, x, w)$
- *The verifier takes an instance of the relation and the SRS and returns a proof* $0/1 \leftarrow V(\sigma, \pi, x)$

Our formal definition is more specialized than this broad one, to better capture the structure of SNARKs we encountered. For example, the values of the field elements corresponding to the SRS elements are represented as polynomial functions of a collection of 1 to 6 field elements that the trusted third party samples uniformly at random. These sample elements are sometimes called *toxic waste* to reflect the fact that if they become known to the prover, the soundness of the proof system can be broken. The trusted third party produces the SRS from the waste samples, which the prover can then use to produce the proof. The verifier will subsequently verify the proof-message by making equality checks on G_T elements produced through the proof elements, the SRS, the statement, and the pairing.

Another way in which our formalization is specialized is that it assumes the witness is a vector of field elements. In order to provide a proof system for any language in NP in the AGM, we must have some way of reducing the problem to a language that involves the field our pairing works over. This is typically done by first encoding the relation into an arithmetic circuit and then reducing the circuit to a language such as the language of Quadratic arithmetic programs (QAPs) or Square Sum Programs (SSPs). The details of these representations are not too important; we simply refer to them collectively as *circuits*. They do work over field elements, though, which informs our choice of witness representation.

Figure 1 shows the Lean data structure which we use to represent the Proof System instantiations. The structure includes:

- A Type of statements
- A variety of Types to index into the set of toxic waste samples, the equality checks made by the verifier, and the pairings the verifier computes, as well as the SRS elements and Proof elements for both the left (G_1) and right (G_2) groups.
- Explicit lists to represent, for both groups, the complete lists of SRS element indices and proof indices. Additionally, there is a family of lists of pairings, one for each equality check.
- Maps from SRS element indices to the multivariable polynomials over the sample elements that the trusted third party uses to compute them.
- Functions that indicate, for a particular pairing input group, statement, pairing, and SRS element (similarly, proof element), the coefficient to which the verifier will include that SRS element (or proof element) in the linear combination which comprises the corresponding input to that pairing.
- A list of identified proof elements, to capture the possibility of non Type III pairings (see Section 2.3.1)

Note that we use the terminology "instantiation" to indicate that a member of this type is a proof system for a specific circuit or problem instance. By contrast, we use the terminology "scheme" or "protocol" to describe systems which, like the six we cover, are general. While these linear PCP SNARK schemes can represent any problem in NP via a circuit crafted, it should be noted that the SRS will depend on the circuit, and that these SNARKs therefore do not have a "universal" trusted setup.

2.3 Formalizing Soundness in the AGM

The only input to an adversarial proof generation procedure, in terms of cryptographic group elements, are the SRS elements that the trusted third party provides. Thus, the AGM

```
structure AGMProofSystemInstantiation
  (F : Type) [Field F] where
 Stmt : Type
 Sample : Type
 SRSElements_G1 : Type
 ListSRSElements_G1 : List SRSElements_G1
 SRSElements_G2 : Type
 ListSRSElements_G2 : List SRSElements_G2
 SRSElementValue_G1 : SRSElements_G1 → MvPolynomial
     Sample F
 SRSElementValue_G2 : SRSElements_G2 → MvPolynomial
     Sample F
 Proof_G1 : Type
 ListProof_G1 : List Proof_G1
 Proof_G2 : Type
 ListProof_G2 : List Proof_G2
 EqualityChecks : Type
 Pairings : EqualityChecks → Type
 ListPairings : (k : EqualityChecks) → List (Pairings k)
 verificationPairingSRS_G1 : Stmt →
  (k : EqualityChecks) → Pairings k → SRSElements_G1 →
     F
 verificationPairingSRS_G2 : Stmt →
  (k : EqualityChecks) → Pairings k → SRSElements_G2 →
     F
 verificationPairingProof_G1 : Stmt →
  (k : EqualityChecks) → Pairings k → Proof_G1 → F
 verificationPairingProof_G2 : Stmt →
  (k : EqualityChecks) → Pairings k → Proof_G2 → F
 Identified_Proof_Elems : List (Proof_G1 × Proof_G2) :=
     []
```

Figure 1: Data structure for SNARK instantiations

guarantees that each curve point in G_1 or G_2 that comprises the proof will be formed as a linear combination of the SRS elements from those respective groups. With this, and the SNARK and AGM definitions in hand, we can define knowledge soundness against algebraic prover adversaries.

Definition 3. *A SNARK (SRS, P, V) is knowledge-sound with respect to a relation R if for any input statement x and adversarial prover consisting of two algebraic algorithms $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$ to output proof components in either group, there is a PPT extractor which takes the output of the corresponding $\mathcal{X}_1, \mathcal{X}_2$ and recovers a value w such that either $(x, w) \in R$ or $V(\sigma, \pi, x) = 0$, except with negligible probability.*

Again, the formalization of this makes a few structural elaborations for simplicity. All of the operations the algebraic prover and verifier execute are either additions and multiplications on the underlying elements of $\mathbb{F}_p$ coming from the SRS elements and linear combination coefficients of the prover. Since the SRS elements are themselves multivariable polynomials over the toxic waste elements, we can therefore obtain the expressions which the verifier ultimately compares in its equality checks as multivariable polynomials over the toxic waste. Since the toxic waste elements are chosen at random, the Schwartz-Zippel lemma guarantees that the checks almost certainly will only pass (and therefore that the proof is only likely to succeed) if these multivariable polynomials are exactly equal.

Lemma 1 ([62, 68]). *Let $p \neq q$ be n-variable polynomials of total degree d over a finite field $\mathbb{F}$. Let $r_1, \cdots, r_n \leftarrow \mathbb{F}$ be sampled uniformly at random. Then with probability at most $\frac{d}{|\mathbb{F}|}$ over the samples,*

$$\Pr[p(r_1, \ldots, r_n) = q(r_1, \ldots, r_n)] \leq \frac{d}{|F|}$$

We use this fact to do away with the "except with negligible probability" clause, and simply directly check in our formalization that the polynomials are equal. Figure 2 shows how this is captured in terms of a Lean Proposition, which asserts that this equality holds for all the checks in the index.

```
def AGMProofSystemInstantiation.Prover
  (F : Type) [Field F]
  (P : AGMProofSystemInstantiation F) : Type :=
 (P.Proof_G1 → P.CrsElements_G1 → F)
 × (P.Proof_G2 → P.CrsElements_G2 → F)

def AGMProofSystemInstantiation.verify
  {F : Type} [Field F]
  (P : AGMProofSystemInstantiation F) (prover : P.Prover)
    (stmt : P.Stmt) : Prop :=
 (∀ check_idx : P.EqualityChecks,
  ((P.ListPairings check_idx).map fun pairing =>
   (((P.ListProof_G1.map fun pf_elem =>
    C (P.verificationPairingProofLeft
       stmt check_idx pairing pf_elem)
    *
    P.proof_element_G1_as_poly prover pf_elem).sum)
    +
    ((P.ListCrsElements_G1.map fun crs_elem =>
      C (P.verificationPairingCRSLeft
         stmt check_idx pairing crs_elem)
     * (P.crsElementValue_G1 crs_elem)).sum))
  *
   (((P.ListProof_G2.map fun pf_elem =>
    C (P.verificationPairingProofRight
       stmt check_idx pairing pf_elem)
     *
    P.proof_element_G2_as_poly prover pf_elem).sum)
    +
    ((P.ListCrsElements_G2.map fun crs_elem =>
      C (P.verificationPairingCRSRight
         stmt check_idx pairing crs_elem)
     * (P.crsElementValue_G2 crs_elem)).sum))).sum = 0)
 ∧
 ∀ pfs ∈ P.Identified_Proof_Elems,
  P.proof_element_G1_as_poly prover pfs.fst =
    P.proof_element_G2_as_poly prover pfs.snd
```

Figure 2: Verification procedure for SNARKs.

This gives us the path to defining soundness (see Figure 3): In order to formally assert the soundness of such a SNARK instantiation, we provide an extractor which gets access to the prover in the form of the coefficients of the linear combinations of SRS elements used to construct the proof, and

a predicate that the witness should satisfy. We say that if the satisfaction of the multivariate polynomial equalities corresponding to the verifier checks implies that the extracted witness satisfies the predicate, then the system is sound with respect to that predicate and extractor.

```
def AGMProofSystemInstantiation.soundness
  (F : Type) [Field F]
  (P : AGMProofSystemInstantiation F)
  (Wit : Type)
  (relation : P.Stmt → Wit → Prop)
  (extractor : P.Prover → Wit) : Prop :=
 ∀ stmt : P.Stmt,
  ∀ prover : P.Prover,
   P.verify prover stmt →
    relation stmt (extractor prover)
```

Figure 3: Formalization of the soundness predicate.

To summarize, formally specifying and proving the soundness of a pairing-based linear PCP SNARK in the AGM model consists of:

- Identifying the toxic waste elements
- Formally modeling the SRS elements as polynomials over the toxic waste elements
- Formally modeling the proof elements as parameterized linear combinations of SRS elements
- Formalizing the verification equations as equations over the proof elements and SRS elements
- Formalizing the satisfaction condition for the NP-Complete relation which the SNARK certifies instances of.
- Formally proving that the verification equations holding implies that the extractor obtains a valid witness under this relation.

2.3.1 Pairing types

As a short remark on our framework, we note that elliptic curve pairings can actually come in one of three types: Type I, Type II, or Type III, per the classification of [41]. A Type III elliptic curve is one in which there is no efficiently computable homomorphism between G_1 and G_2. Practically, this means that for a SNARK defined using a Type III curve, we may assume that the prover cannot include components from G_1 SRS elements in a G_2 proof element and vice versa. Most of the SNARKs we deal with use Type III groups. Type II and Type I pairings assume, respectively, an efficiently computable one-way map or an efficiently computable bijection between G_1 and G_2. Thus, for SNARKs designed to work with these pairings, we must assume that SRS elements of G_1 are available in G_2 as well, or that elements of both groups are available in either. This means that adversaries for such SNARKs have access to many more variables, and the soundness proofs can therefore be more involved. The Groth '16 SNARK is notably compatible with any kind of pairing, and we found that the proof of this SNARK was the most computationally intensive (see Table 2). While our own formalization most closely reflects the semantics of the Type III pairing, we can also formalize Type I and II SNARKs in it by providing these additional SRS elements, specifying that proof elements are proved in *both* groups, and using the final `Identified_Proof_Elems` field to guarantee that these two proof elements are equal.

3 Automation of the Soundness Proof

Our development uses a variety of techniques and tools from the Lean ecosystem to automate the soundness proofs for the SNARKs we study. In this section, we outline the general structure of the formal proofs by way of a toy example of a SNARK.[1] While this proof is very simple, it constitutes an outline which is analogous in its steps to the process we applied to all the SNARKs we formalized. In Section 4, we explain some of the reasons why this procedure works even for the much more complicated SNARKs we see in the literature.

3.1 Specification of the Toy SNARK

We first describe the relation our Toy SNARK certifies. We assume that there are two witness elements $A, B \in \mathbb{F}_p$ that the SNARK verifier has access to and three statement elements $x, y, z \in \mathbb{F}_p$ that the prover has access to. The relation that the verifier wants to check is the disjunction "either $Ax = z$ or $By = z$".

$$R = \{((x,y,z),(A,B)) | A \cdot y = z \vee B \cdot x = z\}$$

The SNARK operations work as follows:

- The trusted third party's produces the SRS production by sampling two toxic waste sample elements which we denote $\alpha, \beta \leftarrow \mathbb{F}_p$, it then outputs the group elements $\alpha_1 = g_1^{\alpha}, \beta_1 = g_1^{\beta}$ in group G_1 and $\alpha_2 = g_2^{\alpha}, \beta_2 = g_2^{\beta}$ in group G_2.
- The prover outputs a single proof element, in the first group, $\pi = \alpha_1^A \beta_1^B = g_1^{A\alpha + B\beta}$.
- The verifier constructs the group 2 element $\alpha_1^x \beta_1^y = g_1^{x\alpha + y\beta}$ and computes the pairing with π to get

$$e(\pi, \alpha_1^x \beta_1^y) = g_T^{(A\alpha + B\beta)(x\alpha + y\beta)}$$

[1] For completeness, we also provide this SNARK formalized in our repository.

It also computes another pairing using z and the SRS elements:

$$e(\alpha_1^z, \beta_2) = g_T^{z\alpha\beta}.$$

As its only check, the verifier determines if these pairings are equal.

- Finally, to prove the soundness, we construct an extractor which obtains values for the witness elements A, B by assuming that they are exactly the coefficients to α and β used to construct π in the AGM, as they are supposed to be.

3.2 The Steps of the Proof

We now describe the proof of soundness step-by-step. Accompanying this explanation is Table 1, which depicts the sequence of operations in tabular form.

3.2.1 Introducing the equality check equations

Taking stock, our formalization of the soundness requires us to prove that the witness elements given by the extractor, if they lead to a satisfactory proof, are themselves satisfactory of the relation.

The first step in the proof is to unfold this definition: A Lean proof environment consists of a number of hypotheses and a goal that the user must prove from those hypotheses. To make sure we have the equations corresponding to the verifier checks among our hypotheses, we can use the Lean `intro` tactic. For our toy SNARK, this gives us the equation corresponding to our single verifier check:

$$(\tilde{A}\alpha + \tilde{B}\beta)(x\alpha + y\beta) = z\alpha\beta$$

Or, in Lean notation,

$$(\texttt{extA} * \texttt{X}\ \alpha + \texttt{extB} * \texttt{X}\ \beta) * (x * \texttt{X}\ \alpha + y * \texttt{X}\ \beta) = z * \texttt{X}\ \alpha * \texttt{X}\ \beta.$$

Note the `X`s applied to α, β. These indicate that, rather than being instances of the sample type of which α, β are members, the multiplicands here are the multivariate polynomials in α, β over F (`MvPolynomial.X` is a `mathlib` function which takes a variable and returns the `MvPolynomial` corresponding to that variable).

3.2.2 Normalizing polynomial equations

The first nontrivial step is to normalize our equation expression by distributing multiplications over additions, to get

$$\begin{aligned}\texttt{extA} * x * \texttt{X}\ \alpha^2 + \texttt{extA} * y * \texttt{X}\ \alpha * \texttt{X}\ \beta + \texttt{extB} * x * \texttt{X}\ \alpha * \texttt{X}\ \beta \\ + \texttt{extB} * y * \texttt{X}\ \beta^2 = z * \texttt{X}\ \alpha * \texttt{X}\ \beta\end{aligned}$$

This can be accomplished using Lean's simplifier: Lean has a built-in tactic called `simp` which takes in a list of lemmas which have the form of a equivalence and recursively applies them to selected hypotheses or goals until it no longer can. So for example, we could call `simp [mul_add]`, which invokes the simplifier with the `mul_add` lemma which asserts that $a(b+c) = ab + ac$ for a, b, c in a ring. Calling `simp` with this and the symmetrical `add_mul` fully distributes multiplication over addition wherever this can be done.

3.2.3 Isolating coefficients

We now take advantage of the form of this equation as a multivariable polynomial over α and β. Two multivariable polynomials are equal only if each corresponding coefficient is equal. If we apply this principle to each of the three nonzero coefficients on the left hand side $\alpha^2, \alpha\beta$, and β^2, we get the three equations:

$$\begin{aligned}&\texttt{coeff}(\alpha \mapsto 2, \beta \mapsto 0)(\texttt{extA} * x * \texttt{X}\ \alpha^2 + \texttt{extA} * y * \texttt{X}\ \alpha * \texttt{X}\ \beta \\ &\qquad + \texttt{extB} * x * \texttt{X}\ \alpha * \texttt{X}\ \beta + \texttt{extB} * y * \texttt{X}\ \beta^2) \\ &= \texttt{coeff}(\alpha \mapsto 2, \beta \mapsto 0)(z * \texttt{X}\ \alpha * \texttt{X}\ \beta)\end{aligned}$$

$$\begin{aligned}&\texttt{coeff}(\alpha \mapsto 1, \beta \mapsto 1)(\texttt{extA} * x * \texttt{X}\ \alpha^2 + \texttt{extA} * y * \texttt{X}\ \alpha * \texttt{X}\ \beta \\ &\qquad + \texttt{extB} * x * \texttt{X}\ \alpha * \texttt{X}\ \beta + \texttt{extB} * y * \texttt{X}\ \beta^2) \\ &= \texttt{coeff}(\alpha \mapsto 1, \beta \mapsto 1)(z * \texttt{X}\ \alpha * \texttt{X}\ \beta)\end{aligned}$$

$$\begin{aligned}&\texttt{coeff}(\alpha \mapsto 0, \beta \mapsto 2)(\texttt{extA} * x * \texttt{X}\ \alpha^2 + \texttt{extA} * y * \texttt{X}\ \alpha * \texttt{X}\ \beta \\ &\qquad + \texttt{extB} * x * \texttt{X}\ \alpha * \texttt{X}\ \beta + \texttt{extB} * y * \texttt{X}\ \beta^2) \\ &= \texttt{coeff}(\alpha \mapsto 0, \beta \mapsto 2)(z * \texttt{X}\ \alpha * \texttt{X}\ \beta)\end{aligned}$$

Here, `coeff` (more specifically `MvPolynomial.coeff` from `mathlib`) is a function which takes a finitely supported natural-valued function on the space of coefficients of a multivariate polynomial and returns the coefficient of the monomial in that polynomial for which the exponents to variables correspond to the values of the function. Thus, for the function which maps α to 2 and β to 0, (denoted above as $(\alpha \mapsto 2, \beta \mapsto 0)$), it returns the α^2 coefficient of the polynomial.

This can be carried out in Lean by use of the `congr_arg` lemma, a generic lemma that states that given a function f, $x_1 = x_1$ implies $fx_1 = fx_2$.

3.2.4 Distribute coefficient-taking

We now can carry out another simplification step, which invokes the `coeff_add` lemma. This lemma states that a coefficient of an addition of polynomials is the addition of the coefficients. By applying this lemma, we can transform the

Stage	Description	Tactics invoked	Hypotheses in Proof State
Stage 0	Initial state		`(Aα+Bβ) (xα+yβ) =zαβ`
Stage 1a	Polynomial put in normal form	`simp [...] at eqn`	`Axα²+(Ay+Bx)αβ+Byβ² =zαβ`
Stage 1b	Coefficients are isolated	`h := congr_arg (coeff (...)) eqn`	`coeff αβ (Axα²+(Ay+Bx)αβ+Byβ²` `= coeff αβ zαβ` `...`
Stage 1c	Distribute coefficient-taking	`simp [...] at h`	`coeff αβ Axα²` `+ coeff αβ Ay+Bxαβ` `+ coeff αβ Byβ²` `= coeff αβ zαβ` `...`
Stage 1d	Expression broken down into term-by-term coefficient comparisons	`simp [...] at h`	`(if αβ = α² then Ax else 0)` `+ (if αβ = αβ` `then Ay+Bx else 0)` `+ (if αβ = β² then By else 0)` `= if αβ = αβ then z else 0` `...`
Stage 1e	Coefficient comparisons decided, leaving proof state of polynomial equations in the prover coefficients	`simp [...] at h`	`Ax = 0` `Ay + Bx = z` `By = 0`
Stage 2a	Polynomials are simplified algebraically	`integral_domain_tactic`	`A = 0` or `x = 0` `Ay + Bx = z` `B = 0` or `y = 0`
Stage 2b	Proof state consists of simple equations of prover coefficients	`integral_domain_tactic` or `polyrith`	`Bx = z` `or` `Ay = z`

Table 1: Describing the stages of a proof. The left row gives an example for a toy (incomplete) SNARK illustrating the type of the hypotheses at each stage, with trapdoor elements in blue and prover coefficients in red.

hypotheses to consist of additions of coefficient evaluations solely of monomial terms.

$$\begin{aligned}&\texttt{coeff}(\alpha\mapsto 2,\beta\mapsto 0)(\texttt{extA}*x*\texttt{X}\ \alpha^2)\\&+\texttt{coeff}(\alpha\mapsto 2,\beta\mapsto 0)(\texttt{extA}*y*\texttt{X}\ \alpha*\texttt{X}\ \beta)\\&+\texttt{coeff}(\alpha\mapsto 2,\beta\mapsto 0)(\texttt{extB}*x*\texttt{X}\ \alpha*\texttt{X}\ \beta)\\&+\texttt{coeff}(\alpha\mapsto 2,\beta\mapsto 0)(\texttt{extB}*y*\texttt{X}\ \beta^2)\\&=\texttt{coeff}(\alpha\mapsto 2,\beta\mapsto 0)(z*\texttt{X}\ \alpha*\texttt{X}\ \beta)\end{aligned}$$

$$\begin{aligned}&\texttt{coeff}(\alpha\mapsto 1,\beta\mapsto 1)(\texttt{extA}*x*\texttt{X}\ \alpha^2)\\&+\texttt{coeff}(\alpha\mapsto 1,\beta\mapsto 1)(\texttt{extA}*y*\texttt{X}\ \alpha*\texttt{X}\ \beta)\\&+\texttt{coeff}(\alpha\mapsto 1,\beta\mapsto 1)(\texttt{extB}*x*\texttt{X}\ \alpha*\texttt{X}\ \beta)\\&+\texttt{coeff}(\alpha\mapsto 1,\beta\mapsto 1)(\texttt{extB}*y*\texttt{X}\ \beta^2)\\&=\texttt{coeff}(\alpha\mapsto 1,\beta\mapsto 1)(z*\texttt{X}\ \alpha*\texttt{X}\ \beta)\end{aligned}$$

$$\begin{aligned}&\texttt{coeff}(\alpha\mapsto 0,\beta\mapsto 2)(\texttt{extA}*x*\texttt{X}\ \alpha^2)\\&+\texttt{coeff}(\alpha\mapsto 0,\beta\mapsto 2)(\texttt{extA}*y*\texttt{X}\ \alpha*\texttt{X}\ \beta)\\&+\texttt{coeff}(\alpha\mapsto 0,\beta\mapsto 2)(\texttt{extB}*x*\texttt{X}\ \alpha*\texttt{X}\ \beta)\\&+\texttt{coeff}(\alpha\mapsto 0,\beta\mapsto 2)(\texttt{extB}*y*\texttt{X}\ \beta^2)\\&=\texttt{coeff}(\alpha\mapsto 0,\beta\mapsto 2)(z*\texttt{X}\ \alpha*\texttt{X}\ \beta)\end{aligned}$$

3.2.5 Break down into monomial equality conditionals

We can now simplify further: The `X` function is defined as an application of `mathlib`'s `MvPolynomial.monomial` function, which constructs a monomial out of coefficient terms and uses a finitely supported function to represent the exponents of the variables. We can use the lemma `monomial_mul` to collapse the multiplications into single applications of `MvPolynomial.monomial` and the `coeff_monomial` lemma, which asserts that `coeff` applied to `monomial` yields either

the coefficient argument of `monomial` when the function arguments match, or 0 when they do not. We can then apply *function extensionality*, which asserts that functions are equal if they are equal on all of their arguments. Because there are only a finite number of possible arguments to the functions we are comparing, we can convert the comparison of the functions into a conjunction over comparisons of their evaluations. We get:

$$\begin{aligned}
&\texttt{if } (2=2\wedge 0=0) \texttt{ then } (\texttt{extA}*x) \texttt{ else } 0\\
&\quad + \texttt{ if } (2=1\wedge 0=1) \texttt{ then } (\texttt{extA}*y) \texttt{ else } 0\\
&\quad + \texttt{ if } (2=1\wedge 0=1) \texttt{ then } (\texttt{extB}*x) \texttt{ else } 0\\
&\quad + \texttt{ if } (2=0\wedge 0=2) \texttt{ then } (\texttt{extB}*y) \texttt{ else } 0\\
&\quad = \texttt{ if } (2=1\wedge 0=1) \texttt{ then } (z) \texttt{ else } 0
\end{aligned}$$

$$\begin{aligned}
&\texttt{if } (1=2\wedge 1=0) \texttt{ then } (\texttt{extA}*x) \texttt{ else } 0\\
&\quad + \texttt{ if } (1=1\wedge 1=1) \texttt{ then } (\texttt{extA}*y) \texttt{ else } 0\\
&\quad + \texttt{ if } (1=1\wedge 1=1) \texttt{ then } (\texttt{extB}*x) \texttt{ else } 0\\
&\quad + \texttt{ if } (1=0\wedge 1=2) \texttt{ then } (\texttt{extB}*y) \texttt{ else } 0\\
&\quad = \texttt{ if } (1=1\wedge 1=1) \texttt{ then } (z) \texttt{ else } 0
\end{aligned}$$

$$\begin{aligned}
&\texttt{if } (0=2\wedge 2=0) \texttt{ then } (\texttt{extA}*x) \texttt{ else } 0\\
&\quad + \texttt{ if } (0=1\wedge 2=1) \texttt{ then } (\texttt{extA}*y) \texttt{ else } 0\\
&\quad + \texttt{ if } (0=1\wedge 2=1) \texttt{ then } (\texttt{extB}*x) \texttt{ else } 0\\
&\quad + \texttt{ if } (0=0\wedge 2=2) \texttt{ then } (\texttt{extB}*y) \texttt{ else } 0\\
&\quad = \texttt{ if } (0=1\wedge 2=1) \texttt{ then } (z) \texttt{ else } 0
\end{aligned}$$

Here `if ... then ... else ...` is Lean syntax for the function that takes a proposition as its first argument and, according to its truth value, returns either the second or third argument.

3.2.6 Decide monomial equality conditionals

Finally, we leverage the fact that these equalities can be decided to reduce these equations.

$$\begin{aligned}
(\texttt{extA}*x) &= 0\\
(\texttt{extA}*y)+(\texttt{extB}*x) &= z\\
(\texttt{extB}*y) &= 0
\end{aligned}$$

We are left with a simple collection of equations over the base field.

3.2.7 Recursively factor and simplify

The convenience of having our hypotheses in the form of equations over `F` is that our goal is a formula over equations of values having type `F`. All that remains, then, is to use equational reasoning prove our hypotheses imply this goal.

Because the SNARK equations arise through pairings, our hypotheses are all quadratic in the atoms. In fact, many of the equations are of the form `A * B = 0` for atoms `A` and `B`. This is by design, as it is necessary for the proofs to leverage the fact that the product of two values equating to zero implies at least one of the multiplicands is zero. This leads us to formulate the following approach to simplifying the goal: We use the fact (inferred by Lean's typeclass system) that a polynomial ring over a field is an integral domain, and we simplify all equations of the form `A * B = 0` to `A = 0 or B = 0`.

$$\begin{aligned}
\texttt{extA} = 0 \vee x = 0\\
(\texttt{extA}*y)+(\texttt{extB}*x) = z\\
\texttt{extB} = 0 \vee y = 0
\end{aligned}$$

We can then split these hypotheses into two cases and prove the goal for each case, simplifying our hypotheses by rewriting `A` or `B` to 0, and carry on this process until we are left with a collection of goals that cannot be simplified through these rules. This finishes the soundness proof for the toy SNARK, since whichever disjunction we case over, we will eliminate a term from the second equation and be left with an equation which satisfies the goal relation.

To facilitate this, we wrote a tactic `integral_domain_tactic`, which carries out the above simplifications and calls itself recursively until it reaches a point where it can make no more progress. We can then either solve these goals by hand, or by dispatching them with the built-in `mathlib` tactic `polyrith`, which solves problems of this type.

The core of this recursive operation is only around 13 lines of Lean code, as seen in Figure 4 :

4 "Straightforwardness" for SNARKs from the literature

The development outlined in the preceding sections covers a broad design space for the construction of linear PCP SNARKs instantiations. In our experience, though, there are deep-cutting regularities in the way that SNARK schemes in the literature construct these instantiations for particular problem instances. We call SNARK schemes that display these regularities *straightforward*: We now discuss the requisite properties be straightforward and how they allow us to automate the soundness proving process.

One High-Degree Variable One property of the SNARKs we consider is that all only have one toxic waste element that appears to a non-bounded degree in the SRS elements.

```
syntax "integral_domain_tactic" : tactic

macro_rules
| `(tactic| integral_domain_tactic) =>
 `(tactic|
   -- Simplify
   simp_all
    (config := {decide := false, failIfUnchanged :=
     false})
    only [false_or, or_false, true_or, or_true, not_true,
     not_false_iff, add_zero, zero_add, mul_zero,
     zero_mul, mul_one, one_mul, neg_zero, neg_eq_zero,
     add_eq_zero_iff_eq_neg, eq_self_iff_true, Ne.def,
     eq_zero_of_zero_eq, one_ne_zero, mul_ne_zero_iff,
     zero_sub_eq_iff, mul_eq_zero];
   first
    -- If we are done, halt
    | done
    -- If possible, split and recurse
    | cases_or _ ∨ _
     all_goals integral_domain_tactic
    | skip
 )
```

Figure 4: Recursive simplification tactic for systems of equations in an integral domain.

For an example to demonstrate this, consider the Pinocchio protocol. We show the sample elements and SRS elements for Pinocchio in Equation 1. Note how the only variable that appears to a degree greater than 1 is s – all other variables appear only to degree 1 or not at all in each of the SRS elements.

This feature informs how we represent the type of sample elements in our formalization. Since there is just a single element which has this special feature of being raised to large powers in the SRS element, we represent the type of elements as an `Option` type over an inductive type `Vars`, where the unbounded element is represented by the `none` value, and `some` gives values for the bound elements.

```
inductive Vars : Type where
 | r_v : Vars
 | r_w : Vars
 | α_v : Vars
 | α_w : Vars
 | α_y : Vars
 | β : Vars
 | γ : Vars
deriving Repr, BEq

local notation "poly_s" => X (none)
local notation "poly_r_v" => X (some Vars.r_v)
...
```

Figure 5: Pinocchio's sample element type, as it appears in our formalization. We use the `local notation` feature to work with these variables as `MvPolynomials`

The convenience of this is that it allows is to invoke `mathlib`'s `optionEquivRight` just before the "Isolating coefficients" step described in Section 3.2.3. The `optionEquivRight` object is an isomorphism between the type of multivariate polynomials in this option type `MvPolynomial (Option Vars) F` and the type of multivariable polynomials *over single-variable polynomials in* F `MvPolynomial Vars (Polynomial F)`. Rewriting using this equivalence allows us to "hide" the unbound variable polynomials within the `Polynomial F` type, so that we only have to deal with `MvPolynomials` with a bounded total degree. This is important for the coefficient-taking step to go smoothly, since our proof can only extract a fixed number of these coefficients.

Extractor is a projection Another key commonality displayed by all the SNARK schemes in the literature is that the extractor used to prove soundness is extremely simple. In fact, for the six SNARKs we consider, it is always a projection function: Each witness component that the extractor outputs corresponds to exactly one coefficient of one of the components of the proof-message.

The convenience of this is that a projection function is always linear, and can therefore be interpreted as a polynomial function in the prover coefficients which make up its inputs. This means that, when we reach the stage of our proof described in Section 3.2.7 where our hypotheses have been reduced to equations in the prover coefficients over a field, the goal will itself be of the same form as these hypotheses - an equation over the field, where the equated expressions consist solely of additions and multiplications on the prover coefficients and statement and witness values.

This is an important observation. If we subtract off the right hand side of each of these equations, we are left to prove that when the polynomials that remain in the hypotheses are zero, the polynomial in the goal is also zero. This is known as the *ideal membership testing problem*, and has been well-studied for decades – it is known to be decidable by a class of algorithms known as Gröbner basis methods [21], which `polyrith` implements. This is a strong hint that we have the right approach - because of this regularity, we are consistently able to reduce our soundness proofs to problems of this decidable form.

SRS Components regular and fixed over circuits One issue is that Gröbner basis algorithms are superexponential the size of the circuit [32,52]. This frustrates the goal of verifying soundness because many circuits verified by SNARKs are themselves cryptographic in nature, and therefore too large to be handled by brute force.

However, a second regularity in the SNARK systems in the literature allows us to get around this. While a particular SNARK instantiation may have arbitrarily many SRS variables, a particular *scheme* like Pinocchio or Groth '16 consistently organizes these variables into a fixed number

of components. Each component always has either a single element, or is a collection of elements for which the prover or verifier computes an inner product with the statement or part of the witness to produce either a proof or paring input. When there is a collection with multiple elements, the expression of the elements is always uniform - all elements of the component are obtained by a multiplication of some polynomial in the degree-unbound variable with a fixed polynomial in the bound-degree variables.

We leverage these components by abstracting over them to get a soundness proof that works for all circuit instances at once. The uniformity of the SRS collections is important - when we factor out constants from these sums, we are left with sums over free variables. Due to the linearity of the adversary in the AGM, it will be the case that whenever one of these elements appears in the proof, it is always as part of a sum indexed over all members of the collection. Because we have no constraints over the instance-specific polynomials, once we factor out constant-degree factors, these formal sums can be treated as atoms by the soundness proof.

Abstracting over SNARK schemes in this way, the size of the ideal membership testing problem instances we encounter will depend only on the SNARK scheme. Gröbner basis methods therefore now have a chance of working, since the instance sizes will be smaller and will not implicitly encode cryptographically-hard problems.

Straightforwardness and General SNARK Decidability The observations made in this section can be crystallized into a *straightforwardness* predicate on linear SNARK schemes which captures all of the properties we need in order for us to be able to carry out our proof process. We find that an informative theorem can be demonstrated about SNARK schemes of this kind:

Theorem 1. *Let $\mathcal{E}$ be the set of all cryptographically secure elliptic curve pairings (G_1, G_2, G_3, e), and let ι be the set of all circuit instances accepted by the scheme. Let $\mathcal{S} : \mathcal{E} \times \iota \to \Pi$ be a straightforward linear PCP SNARK scheme for relation $\mathcal{R}$ which returns an instantiated SNARK protocol Π, given a pairing and instance. Then the proposition*

$$\forall E \in \mathcal{E}, \forall i \in \iota, \forall \pi \in \mathcal{A}_{\mathsf{AGM}}, \forall s$$

$$\mathcal{S}_{E,i}.\mathsf{Verify}(s, \pi) \implies (s, \mathcal{S}_{E,i}.\mathsf{Extractor}(\pi)) \in \mathcal{R}$$

is decidable.

A proof of this fact roughly follows from the procedural outline given in the above parts of this section. On the level of individual SNARK instantiations with fixed fields, Theorem 1 does not tell us much. If we were to fix the field and circuit instance, there is a brute-force soundness deciding procedure for the resulting linear PCP SNARK: Since every algorithm in play takes an input consisting of a finite number of group or field elements, one can explicitly enumerate the possible inputs at each step and determine exactly how often each potential adversarial prover tricks the verifier.

But this naive decision procedure has the downsides of having exponential running time in both the size of the field and the size of the instance, as well as only ultimately deciding soundness for a single instantiation. Our procedure, on the other hand, works over a scheme as a whole, rather than particular instantiations of one. Thus, the running time depends only on the description of the scheme, and while this running time is still technically super-exponential due to the Gröbner basis procedure, our results show that it terminates in a reasonable amount of time in practice.

5 Design of SNARKs and their proofs

In this section, we discuss some of the SNARKs as they appear in the original references on which our formalizations were based. The process of creating these formalizations was not always smooth, in part because there are a few places in which the arguments presented in the papers are misleading. In the interest of a better understanding of these proofs by the community as a whole, we will explain why we found these proofs confusing, and the impact it had on our proof efforts.

5.1 Pinocchio

First, we discuss Pinocchio. We present the protocol itself in Equation 1 – we match the notation and description from Protocol 2 of [59], but with some of the intermediate definitions made explicit, and the fact that all values are powers of the generator g made implicit.

Pinocchio represents its circuit in terms of various polynomials v_k, w_k, y_k, with k ranging over various index sets: The I_{mid} associated with the prover's knowledge of the witness for the circuit being evaluated and $0 \cup [N]$ associated with the statement the verifier is checking relationship membership for. The $V_{mid}, W_{mid}, Y_{mid}$ proof elements are meant to represent sums over these polynomials with the witness elements as coefficients - so that the extractor can eventually extract them. Pinocchio structures itself so that these polynomials are mostly kept separate from each other except when necessary, with three of the verifier checks being dedicated to validating these three proof elements related to a single group of these polynomials. It is ensured that none of the prover coefficients for $V_{mid}, W_{mid}, Y_{mid}$ can be manipulated to influence these checks by introducing the $\alpha_v, \alpha_w, \alpha_y$ sample elements and the $V'_{mid}, W'_{mid}, Y'_{mid}$ proof elements. These three samples and proof elements appear only in the validation checks corresponding to the v, w, and y polynomials respectively, and they effectively isolate these equations from a malicious prover introducing coefficients to $V_{mid}, W_{mid}, Y_{mid}$ proof elements that should not be there. As the paper puts it, they are to "Check that the linear combinations of $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{Y}$ are in their appropriate spans". Something similar happens

with the Z proof element – Z is intended to be computed as a sum of polynomials from all three sets, so γ is introduced in the corresponding check to ensure only these coefficients can be used in that check. The final check then proves that, when the verifier adds their own statement-related terms to the V, W, and Y polynomial sums, the Y polynomials can be subtracted from the product of the V and W polynomials to get a resulting polynomial which has zeros at a number of critical points corresponding to "gates" from the circuit, so that each zero's presence proves that the corresponding gate was computed correctly. The presence of the zeros are proved by equating the subtraction on the right-hand side of the last check to a pairing on the left-hand side which takes the product of a polynomial t which has the same zeros with an additional h polynomial (which the prover can choose to make the equation work out).

$$
\begin{aligned}
&\texttt{Pinocchio.Sample} := r_v, r_w, s, \alpha_v, \alpha_w, \alpha_y, \beta, \gamma \leftarrow \mathbb{F}_p \\
&\texttt{Pinocchio.SRS} := \\
&\{r_v v_k(s)\}_{k\in I_{mid}}, \{r_w w_k(s)\}_{k\in I_{mid}}, \{r_v r_w y_k(s)\}_{k\in I_{mid}} \\
&\{r_v \alpha_v v_k(s)\}_{k\in I_{mid}}, \{r_w \alpha_w w_k(s)\}_{k\in I_{mid}}, \{r_v r_w \alpha_y y_k(s)\}_{k\in I_{mid}} \\
&\{r_v \alpha_v v_k(s)\}_{i\in[d]}, \{r_v \beta v_k(s) + r_w \beta w_k(s) + r_v r_w \beta y_k(s)\}_{k\in I_{mid}} \\
&g, \alpha_v, \alpha_y, \alpha_w, \gamma, \beta\gamma, r_v r_w t(s), \\
&\{r_v v_k(s)\}_{k\in 0\cup[N]}, \{r_w w_k(s)\}_{k\in 0\cup[N]}, \{r_v r_w y_k(s)\}_{k\in 0\cup[N]} \\
&\texttt{Pinocchio.ProofElements} := \\
&V_{mid} := r_v \sum_{k\in I_{mid}} c_k v_k(x), V'_{mid} := r_v \alpha_v \sum_{k\in I_{mid}} c_k v_k(x) \\
&W_{mid} := r_w \sum_{k\in I_{mid}} c_k w_k(x), W'_{mid} := r_w \alpha_w \sum_{k\in I_{mid}} c_k w_k(x) \\
&Y_{mid} := r_v r_w \sum_{k\in I_{mid}} c_k y_k(x), Y'_{mid} := r_v r_w \alpha_y \sum_{k\in I_{mid}} c_k y_k(x) \\
&H = h(x), Z = \beta \sum_{k\in I_{mid}} r_v c_k v_k(x) + r_w c_k w_k(x) + r_v r_w c_k y_k(x) \\
&\texttt{Pinocchio.Checks} := \\
&e(V'_{mid}, 1) = e(V_{mid}, \alpha_v), \\
&e(W'_{mid}, 1) = e(W_{mid}, \alpha_w), \\
&e(Y'_{mid}, 1) = e(Y_{mid}, \alpha_y), \\
&e(Z, \gamma) = e\left(V_{mid} + W_{mid} + Y_{mid}, \beta\gamma\right) \\
&e\left(\sum_{k\in\{0\}\cup[N]} r_v c_k v_k(s) + V_{mid}, \sum_{k\in\{0\}\cup[N]} r_w c_k w_k(s) + W_{mid}\right) \\
&- e\left(\sum_{k\in\{0\}\cup[N]} r_v r_w c_k y_k(s) + Y_{mid}, 1\right) = e(r_v r_w t(s), H),
\end{aligned}
\tag{1}
$$

In our formalization we surprisingly had the most trouble with the first three checks. The "appropriate spans" in the paper are identified as "the $v_k(x)$'s, $w_k(x)$'s, and $y_k(x)$'s, respectively". Somewhat confusing is whether this is meant to be just the k indices in the witness or in the statement too - if the witness and statement polynomials are not linearly independent, then an AGM adversary can add these linear combinations to their proof coefficients, leading to a verifying set of coefficients for which the coefficients of the witness polynomials in $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{Y}$ do not match the witness itself. This does not actually change the proof elements, but it makes them trickier to reason about formally in the model.

A more glaring issue is that these checks *by themselves* do not guarantee that the proof elements will be in the span of the statement and witness polynomials combined: It is technically possible for a prover to construct proof elements with nonzero coefficients for verifier key elements, and for which these checks pass. Specifically, after constructing a key honestly, the prover can multiply $g_v^{V_{mid}}$ and $g_v^{V'_{mid}}$ by g^1 and g_v^{α} respectively, and similarly with W_{mid} and Y_{mid}. We noticed this by way of our attempt to construct a proof that the only nonzero coefficients for $\mathcal{V}$ are in this set, finding that even after simplifying using the equalities generated by this check, there were still terms that we could not eliminate.

The fifth check turns out to be enough to complete the guarantee: It primarily shows "that the same coefficients were used in each of the linear combinations over $\mathcal{V}$, $\mathcal{W}$ and $\mathcal{Y}$", but it also precludes the construction of proof terms which fall outside the span. This caused us to change our original plan for the proof: Rather than prove $\mathcal{V}$ was exactly equal to $g^{V_{mid}}$, we left the g^1 terms (as well as the statement polynomial terms) in our simplifications of $\mathcal{V}$, $\mathcal{W}$ and $\mathcal{Y}$, then proved that when one plugs these into the fifth check, the extra terms can be ignored and the coefficients of the $g_v^{\beta v_k(s)} g_w^{\beta w_k(s)} g_y^{\beta y_k(s)}$ terms in Z are equal to the supposed coefficients of $\mathcal{V}$, $\mathcal{W}$ and $\mathcal{Y}$, so that *these* coefficients can act as the extracted witness.

5.2 Groth '16

Consider Groth '16, described in Equation 2 (following the notation from Section 3 of [43]). Groth '16 came after Pinocchio chronologically and made several refinements – its proof is more compact, and it requires only four pairing computations and a single verifier check, making it much quicker to verify.

The one Groth '16 check has to do the work of all the checks from Pinocchio, so the proof is more complicated, but some features are roughly analogous. The A, B and C proof elements essentially correspond to the roles of V, W and Y elements from Pinocchio (note that v, w, y from Pinocchio have been renamed to u, v, w here). The main conceptual difference is that the Z and H proof elements have been folded into C, and some additional terms have been moved around or added. Another small difference is the presence of r and s terms – r and s are random values meant to keep the prover from leaking information.

$$\texttt{Groth16.Sample} := x, \alpha, \beta, \gamma, \delta \leftarrow \mathbb{F}_p$$

$$\texttt{Groth16.SRS} :=$$

$$\alpha, \beta, \gamma, \delta, (x^i)_{i=0}^{n-1}, \left(\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\right)_{i=0}^{l}$$
$$\left(\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta}\right)_{i=l+1}^{m}, \left(\frac{x^i t(x)}{\delta}\right)_{i=0}^{n-2}$$

$$\texttt{Groth16.ProofElements} :=$$

$$A := \alpha + \sum_{i=0}^{m} a_i u_i(x) + r\delta, B := \beta + \sum_{i=0}^{m} b_i v_i(x) + r\delta, \tag{2}$$
$$C := \frac{\sum_{i=0}^{m} a_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) + h(x)t(x)}{\delta}$$
$$+ As + rB - rs\delta,$$

$$\texttt{Groth16.Checks} :=$$

$$e(A,B) - e(\alpha,\beta)$$
$$= e(C,\delta) + e\left(\frac{\sum_{i=0}^{l} a_i(\beta u_i(x) + \alpha v_i(x) + w_i(x))}{\gamma}, \gamma\right)$$

A first obstacle in formalizing the system as described here is that γ and δ appear in the denominator of some expressions. This means that these are not technically polynomials, but rather Laurent polynomials. This poses a slight problem for the formalization, as `mathlib` does not support multivariable Laurent polynomials.

Luckily, we can sidestep this issue by multiplying through all the SRS elements by $\gamma\delta$. This transformation doesn't impact the functioning or the soundness of the SNARK – we create a formal version of a theorem to this effect in our codebase. After we have done the SRS multiplication transformation, we get the new SRS:

$$\texttt{Groth16.SRS}' :=$$

$$\alpha\gamma\delta, \beta\gamma\delta, \gamma^2\delta, \gamma\delta^2, \{x^i\gamma\delta\}_{i=0}^{n-1}, ((\beta u_i(x) + \alpha v_i(x) + w_i(x))\delta)_{i=0}^{l}$$
$$\{(\beta u_i(x) + \alpha v_i(x) + w_i(x))\gamma\}_{i=l+1}^{m}\}, \{x^i t(x)\gamma\}_{i=0}^{n-2} \tag{3}$$

In the written proof of soundness of Groth16, similar to Pinocchio, it is tricky to parse which equalities of terms in the check polynomial guarantee which relationships between coefficients of the SRS elements. The soundness proof given by Theorem 1 of [43] proceeds by first analyzing the $\alpha^2, \alpha\beta$, and β^2 coefficients of the polynomial, which allows one to simplify this polynomial, zeroing out a few of the terms without loss of generality. We are then shown that the terms involving $1/\delta^2$ give us:[2]

$$\sum_{i=l+1}^{m} A_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) + t(x)A_h(x) = 0$$

[2]Incidentally, there is a typo in the Groth 16 paper here – A_t is written instead of A_h.

We then get the confusing claim that "the terms in" $\alpha\frac{\sum_{i=l+1}^{m} B_i(\beta u_i(x)+\alpha v_i(x)+w_i(x))+B_h(x)t(x)}{\delta} = 0$ show us:

$$\sum_{i=l+1}^{m} B_i(\beta u_i(x) + \alpha v_i(x) + w_i(x)) + B_h(x)t(x) = 0 \tag{4}$$

What is meant by "the terms in" this expression? It seems the paper is saying that the coefficients corresponding to the monomials α^2/δ, $\alpha\beta/\delta$ and α/δ (the only monomials that appear in this expression) imply Equation 4. It is indeed the case that α^2/δ gives us $\sum_{i=l+1}^{m} B_i v_i(x) = 0$, and $\alpha\beta/\delta$ gives us $\sum_{i=l+1}^{m} B_i u_i(x) = 0$, and these reduce the term to

$$\sum_{i=l+1}^{m} B_i w_i(x) + t(x)B_h(x).$$

But notice that the coefficient of α/δ actually includes a term of the form $A(x)\sum_{i=l+1}^{m} B_i\frac{\alpha v_i(x)}{\delta}$, so summing these three coefficients does not give us that this term is zero, it gives us that the term is equal to $-A(x)\sum_{i=l+1}^{m} B_i\frac{\alpha v_i(x)}{\delta}$. Is this expression itself provably equal to 0? Yes, but the justification is not obvious. One finds that the terms with coefficient $1/\delta$ give $\left(\sum_{i=l+1}^{m} B_i w_i(x) + t(x)B_h(x)\right)A(x) = 0$, and after doing casework on whether or not $A(x) = 0$, one sees that either way, the full Equation 4 can indeed be reduced to 0, saving the theorem. Similar reasoning appears immediately after this with "The terms in $\alpha\frac{\sum_{i=l+1}^{m} B_i(\beta u_i(x)+\alpha v_i(x)+w_i(x))+B_h(t)}{\gamma} = 0$", but this can be fixed in the same way.

Helpful in resolving this confusion was Baghery et al. [6] Theorem 2, which carefully lists the equations from each coefficient, and explicitly clarifies that the proof uses cases on $A(x) = 0$. They do this for a version of the Groth '16 SNARK intended for use with type-III pairing friendly curves, but the proof carries over to the non-type-III case.

6 Evaluation of the Proof code

The initial development of the proofs for the six SNARKs was done in Lean 3 over the course of several months from 2021 through mid-2022. Our first proofs were relatively unsystematic, as we had not identified many of the commonalities from Sections 3 and 4. During 2023, the Lean community deprecated Lean 3 and ported `mathlib` to Lean 4. We ported our code as well, and took the opportunity to make our codebase more uniform by defining the `AGMProofSystemInstantiation` structure and recasting our SNARK definitions in terms of it. While this has codified the SNARK security definitions more strongly, it has made the translation effort more difficult – Our approach to the soundness proofs in Lean 3 was more organic, and used intermediate lemmas to shortcut some parts of the proof or to prepare the proof state before using the heavier tactics to make them run more smoothly. Our more automated approach

Name	Samples	Proof elements	SRS Components	Checks	Check Time	Check Time (Lean 4)
GGPR [42]	5	7 (6)	19	5 (4)	140.61 s	-
Pinocchio [59]	8	8	21	5	342.89s	-
Groth '16 [43]	5	3	8	1	13741.86s	-
Baghery et al. [6]	5	3	7 and 4	1	552.67s	162.74s
BabySNARK [55]	3	3	4	2	74.98s	-
Lipmaa [47]	2	3	7 and 4 and 1	1	81.82s	144.30s

Table 2: Data on different SNARK variants, including the number of toxic waste samples, number of proof elements, number of SRS components, number of verifier equality checks, and the time to check the soundness proofs in both Lean 3 and Lean 4. Note that GGPR includes in the paper a proof element and a check which are not strictly necessary for the soundness proof - we write the number used in our code in parentheticals.

in Lean 4 makes it harder to identify bugs when they exist. Currently we have only translated two of our six soundness proofs to Lean 4.

Summarizing our code by line count, we have:

- 150 LOC defining `AGMProofSystemInstantiation` and its soundness property.
- Approximately 3000 LOC of code in total defining and proving soundness across the six SNARK schemes.
- 80 LOC for `integral_domain_tactic` and other simplification tactics.
- 212 LOC for definitions and theorems about the transformations of SNARKs, including:
  - A theorem proving that, given a SNARK, it is possible to multiply through each SRS element by an existing (or new) toxic waste sample without affecting the soundness of the SNARK. This theorem justifies our treatment of the Laurent polynomials in Groth '16.
  - A theorem proving that if a toxic waste σ element appears to maximum degree $< n$ in a check polynomial, and τ is another toxic waste element, then it is possible to replace τ with σ^n is the maximum degree, thus reducing the number of toxic waste samples needed. One can view this transformation as one of the conceptual differences between the Lipmaa SNARK [47] relative to the Baghery et al. presentation.
  - A theorem proving that if the coefficient of a particular SRS element in a particular proof element is the same for any satisfying statement-witness pair, then we can treat that component as a constant and remove the dependence of the proof element linear combination on that SRS element. This could potentially be useful in optimizing SNARK circuits after the trusted setup.
- 419 LOC for supporting lemmas about polynomials and other datatypes intended for upstreaming to `mathlib`, as well as 53 LOC modifying existing `mathlib` definitions to make them more general and compatible with our development.

Our code is available on GitHub and is also listed on Reservoir, the Lean 4 package index.[3]

Table 2 shows data about the sizes of various parameters of the various SNARKs and the time it takes Lean to verify the proofs. In each case, the proof time is dominated by the mutual simplification phase, as one might expect, given that this phase requires recursive casework.

7 Future Work and Conclusion

To bring our discussion to a close, we mention a few topics that could make for good future research directions in the area of formal verification of SNARKs:

Integrating polyrith and independent verifiability The bulk of the work of our proof procedure happens in the recursive calls to `integral_domain_tactic`. While this is necessary in practice to reduce the size of the Gröbner basis computation instances passed to `polyrith`, a version of `polyrith` with unlimited proof size could dispatch this goal all on its own. One good direction for future work would be to integrate the `integral_domain_tactic` heuristic into `polyrith` to make this possible. One could then take advantage of other features of `polyrith`: In particular, `polyrith` by default logs to the user a description of the sequence of equational operations one carries out on the hypotheses to reach the conclusion. One could use this to create external tools, potential faster than our own, that check the certificates of SNARK soundness our code produces. One drawback is that this external verifier would then itself be part of the trusted computing base for our proof (working in Lean all proofs are

[3]The latest version of our code can be found at https://github.com/BoltonBailey/formal-snarks-project

checked by the kernel – vetting the proof only requires checking our formalization of the *theorem statement*).

Practical Application To bring the work further into practical relevance, one could use our framework to produce soundness proofs for real-world SNARK implementations, or modify it to provide explicit attacks if they exist. This would involve precisely replicating these implementations with our `AGMProofSystemInstantiation` structure. One could also try to use our system to search for new sound schemes. An obstacle here is performance: Each of our proof scripts takes at least several seconds to run, and since there a multitude of different values the generic `AGMProofSystemInstantiation` can take, it would take an unreasonable amount of time to search through them while simultaneously checking each of them for soundness. It would therefore be necessary to aggressively prune the search space, and perhaps also find algorithmic optimizations to our code.

Larger Cryptographic Frameworks Section 1.3 discusses a number of cryptographic frameworks in other languages [11, 14, 25]. Future work could create such a framework for Lean, which our work could then fit into. This would involve both replicating the work of these other frameworks, as well as ironing out some of the details, (such as the Schwartz-Zippel lemma applications) that we currently elide.

Other kinds of SNARKs Future work in formalizing SNARKs could move outside the linear PCP paradigm to newer SNARKs such as Sonic, PlonK, Marlin [27, 40, 49]. These "Polynomial IOP" SNARKs use the Fiat-Shamir paradigm [35], which involves the random oracle model [15], so a formalization here would have to cover this idea.

Conclusion We have presented our efforts to formalize the Groth '16 SNARK and similar constructions in Lean. Our work includes a variety of programs that help accomplish this task, and we have described a variety of pitfalls associated with this challenge and our strategies for overcoming them. Our work indicates that there is substantial facility to use automated proof techniques in the formalization of cryptography – going forward, we hope that tools with which security researchers can more easily analyze SNARKs continue to develop.

Acknowledgements

The authors would like to thank Bryan Parno for discussions on early drafts of this paper.

This material is based upon work supported by the National Science Foundation under the Graduate Research Fellowship Program with Grant No. DGE – 1746047, and additionally under NSF grant #1943499.

References

[1] Carmine Abate, Philipp G Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–15. IEEE, 2021.

[2] José Bacelar Almeida, Manuel Barbosa, Manuel L Correia, Karim Eldefrawy, Stéphane Graham-Lengrand, Hugo Pacheco, and Vitor Pereira. Machine-checked ZKP for NP relations: Formally verified security proofs and implementations of MPC-in-the-head. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2587–2600, 2021.

[3] David Kurniadi Angdinata and Junyan Xu. An elementary formal proof of the group law on Weierstrass elliptic curves in any characteristic. *arXiv preprint arXiv:2302.10640*, 2023.

[4] Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman. A verified algebraic representation of Cairo program execution. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 153–165, 2022.

[5] José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 488–500, 2012.

[6] Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. Another look at extraction and randomization of Groth's zk-SNARK. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I 25*, pages 457–475. Springer, 2021.

[7] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *2021 IEEE symposium on security and privacy (SP)*, pages 777–795. IEEE, 2021.

[8] Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. EasyPQC: Verifying post-quantum cryptography. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2564–2586, 2021.

[9] Gilles Barthe, Jan Cederquist, and Sabrina Tarento. A machine-checked formalization of the generic model and the random oracle model. In *Automated Reasoning: Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004. Proceedings 2*, pages 385–399. Springer, 2004.

[10] Gilles Barthe, Juan Manuel Crespo, Yassine Lakhnech, and Benedikt Schmidt. Mind the gap: Modular machine-checked proofs of one-round key exchange protocols. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 689–718. Springer, 2015.

[11] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 90–101, 2009.

[12] Gilles Barthe, Daniel Hedin, Santiago Zanella Béguelin, Benjamin Grégoire, and Sylvain Heraud. A machine-checked formalization of sigma-protocols. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 246–260. IEEE, 2010.

[13] David Basin, Andreas Lochbihler, Ueli Maurer, and S Reza Sefidgar. Abstract modeling of system communication in constructive cryptography using CryptHOL. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16. IEEE, 2021.

[14] David A Basin, Andreas Lochbihler, and S Reza Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *Journal of Cryptology*, 33:494–566, 2020.

[15] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[16] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 90–108. Springer, 2013.

[17] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 781–796, 2014.

[18] Dhruv Bhatia. A tactic using Sage to solve polynomial equalities with hypotheses, 2022. https://github.com/leanprover-community/mathlib/pull/14878.

[19] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. Cryptology ePrint Archive, Report 2020/352, 2020. https://ia.cr/2020/352.

[20] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In *Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part III 24*, pages 222–249. Springer, 2018.

[21] Bruno Buchberger. A criterion for detecting unnecessary reductions in the construction of Gröbner-bases. In *International Symposium on Symbolic and Algebraic Manipulation*, pages 3–21. Springer, 1979.

[22] David Butler, David Aspinall, and Adrià Gascón. Formalising oblivious transfer in the semi-honest and malicious model in CryptHOL. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 229–243, 2020.

[23] David Butler, Andreas Lochbihler, David Aspinall, and Adrià Gascón. Formalising σ-protocols and commitment schemes using CryptHOL. *Journal of Automated Reasoning*, 65(4):521–567, 2021.

[24] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[25] Ran Canetti, Alley Stoughton, and Mayank Varia. EasyUC: Using EasyCrypt to mechanize proofs of universally composable security. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 167–16716. IEEE, 2019.

[26] Yanju Chen, Clara Rodriguez, Yu Feng, and Bryan Tan. Picus, 2022.

[27] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 738–768. Springer, 2020.

[28] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A programming language for formally verified, zero-knowledge applications. *Cryptology ePrint Archive*, 2021.

[29] Alessandro Coglio. Ethereum's recursive length prefix in ACL2. *arXiv preprint arXiv:2009.13769*, 2020.

[30] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, and Bogdan Warinschi. Machine-checked proofs for electronic voting: privacy and verifiability for Belenios. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 298–312. IEEE, 2018.

[31] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015.

[32] Thomas W Dubé. The structure of polynomial ideals and Gröbner bases. *SIAM Journal on Computing*, 19(4):750–773, 1990.

[33] François Dupressoir, Konrad Kohbrok, and Sabine Oechsner. Bringing state-separating proofs to EasyCrypt a security proof for Cryptobox. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, pages 227–242. IEEE, 2022.

[34] Karim Eldefrawy and Vitor Pereira. A high-assurance evaluator for machine-checked secure multiparty computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 851–868, 2019.

[35] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1987.

[36] Denis Firsov and Dominique Unruh. Zero-knowledge in EasyCrypt. Cryptology ePrint Archive, Paper 2022/926, 2022. https://eprint.iacr.org/2022/926.

[37] Cédric Fournet, Chantal Keller, and Vincent Laporte. A certified compiler for verifiable computing. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 268–280. IEEE, 2016.

[38] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II 38*, pages 33–62. Springer, 2018.

[39] Ariel Gabizon. On the security of the BCTV Pinocchio zk-SNARK variant. Cryptology ePrint Archive, Report 2019/119, 2019. https://ia.cr/2019/119.

[40] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://ia.cr/2019/953.

[41] Steven D Galbraith, Kenneth G Paterson, and Nigel P Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008.

[42] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, pages 626–645. Springer, 2013.

[43] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 305–326. Springer, 2016.

[44] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. Computer-aided proofs for multiparty computation with active security. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 119–131. IEEE, 2018.

[45] Thomas Haines, Rajeev Goré, and Bhavesh Sharma. Did you mix me? formally verifying verifiable mix nets in electronic voting. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1748–1765. IEEE, 2021.

[46] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant a tutorial. *Rapport Technique*, 178, 1997.

[47] Helger Lipmaa. A unified framework for non-universal SNARKs. In *Public-Key Cryptography–PKC 2022: 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8–11, 2022, Proceedings, Part I*, pages 553–583. Springer, 2022.

[48] Andreas Lochbihler, S Reza Sefidgar, David Basin, and Ueli Maurer. Formalizing constructive cryptography using CryptHOL. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 152–15214. IEEE, 2019.

[49] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2111–2128, 2019.

[50] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. CanDID: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1348–1366. IEEE, 2021.

[51] The mathlib community. The Lean mathematical library. *CoRR*, abs/1910.09336, 2019.

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

[53] Catherine A Meadows and Catherine A Meadows. Formal verification of cryptographic protocols: A survey. In *Advances in Cryptology—ASIACRYPT'94: 4th International Conferences on the Theory and Applications of Cryptology Wollongong, Australia, November 28–December 1, 1994 Proceedings 4*, pages 133–150. Springer, 1995.

[54] Roberto Metere and Changyu Dong. Automated cryptographic analysis of the Pedersen commitment scheme. In *Computer Network Security: 7th International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2017, Warsaw, Poland, August 28-30, 2017, Proceedings 7*, pages 275–287. Springer, 2017.

[55] Andrew Miller, Ye Zhang, and Sanket Kanjalkar. Baby SNARK (do do dodo dodo, 2020.

[56] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[57] David Nowak. On formal verification of arithmetic-based cryptographic primitives. In *Information Security and Cryptology–ICISC 2008: 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers 11*, pages 368–382. Springer, 2009.

[58] Bryan Parno. A note on the unsoundness of vnTinyRAM's SNARK. Cryptology ePrint Archive, Report 2015/437, 2015. https://ia.cr/2015/437.

[59] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Communications of the ACM*, 59(2):103–112, 2016.

[60] Loïc Pottier. Nsatz: a solver for equalities in integral domains, 2021.

[61] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[62] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.

[63] Victor Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology—EUROCRYPT'97: International Conference on the Theory and Application of Cryptographic Techniques Konstanz, Germany, May 11–15, 1997 Proceedings 16*, pages 256–266. Springer, 1997.

[64] Nikolaj Sidorenco, Sabine Oechsner, and Bas Spitters. Formal security analysis of MPC-in-the-head zero-knowledge protocols. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–14. IEEE, 2021.

[65] Sabrina Tarento. Machine-checked security proofs of cryptographic signature schemes. In *ESORICS*, volume 5, pages 140–158. Springer, 2005.

[66] Søren Eller Thomsen and Bas Spitters. Formalizing Nakamoto-style proof of stake. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–15. IEEE, 2021.

[67] Franklyn Wang. Ecne: Automated verification of ZK circuits, 2022.

[68] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226. Springer, 1979.