

Zippel: A Language, Compiler, Runtime, and Analyzer for ZK Proof Systems

Sebastian Angel Sydnie-Shea Cohen Eleftherios Ioannidis* Pratyush Mishra
Alex Ozdemir† Yuxiang Qiu Alireza Shirzad

*University of Pennsylvania *Microsoft Research †Georgia Tech*

***Abstract*—This paper introduces Zippel, a language, compiler, runtime, and analyzer for cryptographic proof systems. Zippel lets developers write proof systems with ease using syntax that mimics pseudocode in cryptography papers. It also eliminates entire classes of vulnerabilities by construction owing to its type system and runtime. For example, Zippel automatically synchronizes protocol transcripts and correctly applies the Fiat–Shamir transform. Zippel also has fully automated static analyses that confirm whether a protocol is complete and sound given only the protocol's description. We have implemented 35 proof systems in Zippel including Sumcheck, Schnorr, KZG, Groth16, Bulletproofs, Hyrax, Spartan, Dory, HyperPlonk, DeKART, and Pari. Our evaluation shows that Zippel programs are competitive with hand-written implementations on multi-core hardware, so Zippel's security and implementation benefits do not come at high cost.**

1. Introduction

In a *cryptographic proof system*, a *Prover* wants to convince a *Verifier* that a particular statement is true. For example, that some data hashes to a specific digest, that executing a program on a given input produces a certain output, or that evaluating a polynomial over all points in a domain and summing the results yields a claimed value. The key guarantees are *completeness* and *soundness*: if the statement is true, the verifier accepts the proof; if the statement is false, the verifier rejects with high probability.

In the last few years, research in cryptographic proof systems has surged. Dozens of new protocols are proposed each year, offering different trade-offs in proof size, prover time, verification time, memory usage, setup type, cryptographic assumptions, and additional properties (e.g., *zero knowledge* [64], *succinctness* [63]). While it is great that practitioners have many options, this volume of proposals presents a challenge, as implementing a proof system in a secure and performant way and adapting it to the needs of an application demands hard work and deep expertise in both cryptography and systems programming.

Our proposal. In this paper we introduce Zippel, our language, compiler, runtime, and analyzer specifically designed for cryptographic proof systems. With Zippel, developers can write proof systems like Schnorr [89], Groth16 [66], Bulletproofs [43], Spartan [90], HyperPlonk [47], and many others, using a *choreographic* programming pattern [82]. This pattern allows the programmer to specify the protocol as a single, unified description that mirrors mathematical pseudocode that one would find in an academic paper.

Zippel features a powerful type system consisting of *visibility types* that statically track which data is visible only to the prover or to both parties, and *sized types* that track the sizes of vectors and the degree and number of variables of polynomials. These types help Zippel prevent out-of-bounds accesses, interpolation errors, and common vulnerabilities such as masking polynomials with insufficient randomness. Zippel also comes with formal syntax and semantics.

Zippel leverages the choreographic pattern and type system to construct a computation graph, called *Graph IR*, that serves as an intermediate representation for three critical purposes. First, it helps the runtime automatically parallelize the protocol's execution on a multi-core CPU.

Second, Zippel implements a *projection algorithm* that leverages the nodes in the Graph IR that correspond to verifier challenges and prover messages to infer the reachability of other nodes (e.g., what the verifier can see), and splits the unified graph into two subgraphs: one for the prover and one for the verifier. From these subgraphs, Zippel automatically manages the transcript, correctly handles the application of the *Fiat–Shamir transform* [61] to eliminate common mistakes [39, 56, 83], and produces high-performance prover and verifier Rust binaries backed by `arkworks` [2]. Our evaluation shows that Zippel's code is competitive with existing implementations of the same proof systems (§9), so the benefits of Zippel do not come at a high cost.

Third, the graph allows Zippel to perform automated completeness and soundness analyses. These analyses are very fast (typically tens of milliseconds) and help developers and AI agents confirm that the protocol that they implemented (or adapted) actually proves the purported statement and is secure. In effect, these analyses act as ground truth "oracles" that catch subtle bugs that are virtually impossible to find with testing alone.

We have implemented 35 proof systems in Zippel, ranging from Schnorr [89] and other Σ-protocols [51], to Bulletproofs [43], Hyrax [98], Dory [76], KZG [71], PST13 [87], KZH [69], DeKART [40], and various Sumchecks [78], Zerochecks [47], and Hadamard product checks [65]. We also implemented full (zk)SNARKs including Groth16 [66], Spartan [90], HyperPlonk [47], and Pari [59]. Our experi-

ence in this process is diverse: we manually implemented a third of these systems, with the rest implemented by AI agents that leveraged Zippel's type system and analyses to find bugs. We reviewed these implementations.
In summary, our main contributions are:

- The Zippel language, a choreographic language designed for cryptographic proof systems (§4).
- A compiler that transforms a Zippel program into a labeled computation graph over which analysis can be done and a projection algorithm that transforms the graph into separate prover and verifier programs (§5).
- A high-performance runtime that auto-parallelizes the execution of these graphs on multi-core machines, manages the transcript, and guarantees the correct application of the Fiat–Shamir transform (§6).
- Static analyses for completeness and special soundness based on polynomial ideals and Gröbner bases (§7).
- Implementation of 35 proof systems in Zippel and evaluation of our analyses and runtime performance (§9).

Limitations. Our current implementation supports univariate, multivariate, and multilinear polynomials, as well as finite fields, elliptic curve groups, and pairings. We do not yet support arbitrary error-correcting codes, hash-based proof systems, lattices, or RSA groups, nor can we leverage GPU acceleration; we are working to add support for these.

Our static analyses cover only completeness and $(\ell_1, \ldots, \ell_\mu)$-*special soundness* [32–34]. We do not have a static analysis for the more general *knowledge soundness* property or to certify that a protocol is *zero-knowledge*.

It is also worth noting that the analyses that we support are not perfect: they produce no false positives (i.e., a protocol confirmed to satisfy a property indeed satisfies it), but they can fail or time out, which gives the developer no information. Furthermore, the analyses perform *bounded model checking*, meaning that they verify properties up to concrete input sizes (e.g., vector lengths, polynomial degrees) rather than for all inputs. This approach is still effective because cryptographic proof systems rarely alter their behavior based on input size, though there are of course exceptions.

Despite these limitations, Zippel is the first system capable of performing fully automated analyses of these properties for many cryptographic proof systems.

Source Code: https://github.com/eniac/zippel.

2. Background and related work

This section reviews cryptographic proofs, explains why writing a proof system from scratch or adapting an existing one is hard, and then discusses works that relate to Zippel.

2.1. Cryptographic Proofs

A *cryptographic proof* is a protocol between two parties: a *Prover* $\mathcal{P}$ and a *Verifier* $\mathcal{V}$. The objective of $\mathcal{P}$ is to convince $\mathcal{V}$ that a specific statement is true. Depending on the proof system, this statement may be expressed as a circuit, or it may be implicitly defined by the structure of the protocol itself. To construct the proof, the prover relies on auxiliary information known as the *witness*.

In an *interactive proof*, the protocol is a multi-round dialogue. $\mathcal{P}$ starts by sending some initial public data (e.g., public values, cryptographic commitments). $\mathcal{V}$ then replies with a random *challenge*. $\mathcal{P}$ uses this challenge, along with the witness, to compute a *response*. This back and forth repeats for some number of rounds. Finally, $\mathcal{V}$ uses the *transcript* of the protocol (i.e., all messages exchanged) to verify the prover's claim.

A cryptographic proof satisfies two properties:

- **Completeness:** An honest $\mathcal{P}$ can always convince $\mathcal{V}$ of a true statement.
- **Soundness:** A malicious $\mathcal{P}$ cannot convince $\mathcal{V}$ of a false statement except with negligible probability.

In addition, some protocols have extra properties:

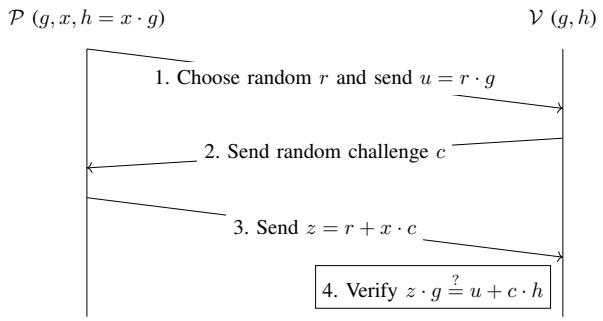
- **Knowledge soundness:** If $\mathcal{P}$ can convince $\mathcal{V}$ of a statement with non-negligible probability, then $\mathcal{P}$ must have knowledge of the witness. Formally, this is shown by the existence of an *extractor* algorithm that can interact with $\mathcal{P}$ (e.g., by rewinding $\mathcal{P}$ to answer different challenges) and efficiently recover the witness.
- **Zero-Knowledge:** $\mathcal{V}$ learns nothing about the witness from the transcript. More formally, there exists a *simulator* with no knowledge of the witness that can generate a "fake" transcript that is indistinguishable from a real transcript. Some protocols require $\mathcal{V}$ to be honest and generate valid challenges. The resulting property is known as *honest verifier zero-knowledge* (HVZK).
- **Succinctness:** $\mathcal{V}$'s computation and communication complexity are sublinear in the size of the witness.

Protocols that satisfy completeness, soundness, and zero-knowledge are called *Zero-Knowledge Proofs* (ZKPs). If they also satisfy knowledge soundness, the proofs are actually *proofs of knowledge*. If soundness holds only against computationally bounded adversaries, they are called *arguments* instead of proofs. And if additionally they are non-interactive and succinct, they are called zkSNARKs. Zippel's language supports writing all of these.

2.2. Non-interactive Cryptographic Proofs

The multi-round dialogue that comes with interactive proofs is often impractical because $\mathcal{P}$ and $\mathcal{V}$ might not be online at the same time, or there may be hundreds or thousands of verifiers who may wish to check the proof (e.g., transaction aggregation in blockchains and decentralized systems [77, 91, 92, 99]). A *non-interactive* cryptographic proof collapses this dialogue into a single proof object π that contains the entire transcript and that *anyone* can use to validate $\mathcal{P}$'s claim without further interaction.

To achieve this, the challenge that $\mathcal{P}$ uses must be derived from public information without $\mathcal{V}$'s help, must be deterministic (so that others can also derive the same challenge), and must be bias-resistant (so that $\mathcal{P}$ cannot



(a) Interactive Schnorr protocol.

```
proto schnorr<G: Group, F: Scalar<G>>
  (witness x: F, instance g: G, instance h: G)
  where h == x*g
{
  let r = random<F>;
  u <- r*g;
  c <- challenge<F>;
  z <- r + x*c;
  verify(z*g == u + c*h);
}
```

Specification relation

Blinding factor

Random oracle challenge

Verifier assertion

(b) Zippel implementation of Schnorr.

Figure 1: Figure (a) shows the Schnorr protocol for proving knowledge of a discrete logarithm typically seen in papers. The arguments g and h are public **instance** parameters and x is a private **witness**, with prime $q \in \mathbb{N}$, $g \in \mathbb{G}, x \in \mathbb{F}_q, h \in \mathbb{G}$. Figure (b) shows the corresponding Zippel code, which closely matches the protocol formulation.

get a challenge that is beneficial to them). The most common technique is the *Fiat–Shamir transform* [61], whereby $\mathcal{P}$ simulates $\mathcal{V}$'s challenge by applying a *random oracle* (heuristically instantiated as a hash function) to the portion of the transcript that would be seen in the interactive proof prior to the challenge (e.g., the initial messages and public values). This transformation from an interactive to a non-interactive protocol is a key technique and has the added benefit of making HVZK protocols resistant to malicious verifiers precisely because $\mathcal{V}$ is no longer the one generating the challenges. Given the prevalence and practicality of this transformation, Zippel must provide first-class support for transcript management and random oracle-based challenges.

2.3. Concrete Example: Schnorr's Protocol

We make the above concepts of challenge, response, commitment, proof, transcript, and the Fiat–Shamir transform concrete with an example. Schnorr [89] devised a ZKP to allow a smart card (or any other device) to authenticate to a server by showing that it knows a secret stored in the smart card without revealing the secret.

Notation. In this paper and also in Zippel's syntax we use additive group notation. So the group operation is addition (e.g., if g is a generator of an elliptic curve group then $x \cdot g$ denotes repeated addition of g to itself x times).

Interactive Schnorr. Given a generator g of a group $\mathbb{G}$ of prime order q, and a public commitment $h \in \mathbb{G}$, the goal of $\mathcal{P}$ is to prove knowledge of a secret scalar $x \in \mathbb{F}_q$ such that $h = x \cdot g$. The protocol is in Figure 1.

This protocol is *complete*, as an honest prover will always pass the verification check:

$$z \cdot g = (r + c \cdot x) \cdot g = (r \cdot g) + (c \cdot x) \cdot g = u + c \cdot (x \cdot g) = u + c \cdot h$$

It is trivially *sound* because the underlying statement (that h is in the group generated by g) is always true as long as h and g have the same type. More importantly, it is *knowledge sound*—or more precisely *special sound* [53] which implies *knowledge soundness*: if $\mathcal{P}$ can make $\mathcal{V}$ accept with probability greater than $1/|\mathbb{F}_q|$, then we can extract the witness. The extraction proceeds by running $\mathcal{P}$ to obtain commitment u and response z_1 to challenge c_1. Then, the extractor rewinds $\mathcal{P}$ to obtain a second response z_2 to a distinct challenge $c_2 \neq c_1$ (for the same commitment u). Since $z_i = r + c_i \cdot x$, the extractor recovers $x = (z_1 - z_2)(c_1 - c_2)^{-1}$.

Finally, Schnorr's protocol is HVZK because a simulator can produce an indistinguishable transcript without knowing x (by picking c and z first, then setting $u = z \cdot g - c \cdot h$). Schnorr's ZKP is not succinct.

Non-interactive Schnorr. The protocol is made non-interactive and resistant to malicious verifiers via the Fiat–Shamir transform. Instead of receiving a random challenge c from $\mathcal{V}$, $\mathcal{P}$ generates c by hashing the public values and the commitment. Specifically, $c = \text{Hash}(q, g, h, u)$, where Hash is modeled as a random oracle. $\mathcal{P}$ then computes z as before. The final non-interactive proof is $\pi = (u, z)$. A verifier can check π by computing $c = \text{Hash}(q, g, h, u)$ and confirming that $z \cdot g = u + c \cdot h$.

2.4. Why Implementing Proof Systems is Hard

Implementing cryptographic proof systems is hard for three main reasons. First, getting good performance requires understanding the cryptographic operators in sufficient detail to know which ones can leverage parallelism (e.g., multi-scalar multiplications of large vectors), exploit structure (e.g., inverse NTT versus Lagrange interpolation of polynomials), or be optimized for special settings (e.g., Sumcheck optimizations [54, 55, 57]), in addition to knowing how to write the corresponding code in a fast language (though AI coding agents can now help here).

Second, it is easy to violate zero-knowledge and soundness. For example, many implementations originally failed to properly blind commitments and polynomials, or would add the witness directly into the transcript, violating zero knowledge [9, 25]. Other examples include incorrect applications of the Fiat–Shamir transform [39, 56, 72, 79, 81], and failure to verify that points belong to the correct elliptic curve group, or that scalar variables are non-zero [100].

Third, while cryptographers spend a lot of time proving the security of their constructions, practitioners often need

to adapt these published works in order to incorporate them into real applications. The challenge is that even the slightest incorrect modification can turn a complete and sound protocol into an insecure implementation, and many practitioners lack the expertise to perform the necessary analysis.

2.5. Related Work

Zippel is related to, but different from, languages that compile statements to circuits that are then proven with ZKPs, languages for Σ-protocols, and works that use choreographies as a programming pattern. We explain below.

Circuit languages. Frameworks like Circom [38], Noir [16], Cairo [62], CirC [85], ZoKrates [22], xJsnark [74], Otti [29], Reef [31], Coral [30], and zkStruDul [75] operate at the circuit generation layer. These languages are designed to let a developer write application logic (e.g., "prove my balance is greater than 100") and compile that logic into a system of arithmetic constraints (e.g., R1CS) that is then used with a ZKP. In contrast, Zippel is not a language for writing applications or circuits; instead, it is a language for implementing the proof systems themselves (e.g., Spartan [90], Groth16 [66], HyperPlonk [47]). As a result, Zippel is complementary to circuit compilers.

Σ-protocol languages. Several compilers and languages exist for Σ-protocols [27, 35, 36, 80]. While these systems inspired our work, Zippel differs in three key aspects. First, Zippel targets a much broader class of protocols, including multi-round protocols, Sumchecks, polynomial commitments (e.g., KZG [71], Dory [76], Hyrax [98]), and full zkSNARKs (e.g., Groth16 [66], Spartan [90], HyperPlonk [47]). Second, Zippel offers a different programming and verification model. The certifying compilers [27, 35] take a declarative relation (e.g., in Camenisch–Stadler notation [45]) and synthesize a standard three-move Σ-protocol together with its correctness and security proofs. If a developer designs a novel protocol (e.g., by modifying the message sequence, adding new rounds, or using different primitives), they cannot use these compilers. In contrast, Zippel allows developers to express a broad class of proof systems and run analyses on them (though our analyses have limitations, see Section 7). Finally, Zippel produces auto-parallelized executables that achieve excellent performance.

Choreographies. Choreographic programming [82] is a paradigm where the developer writes the behavior of a distributed system as a single, unified program. Then, through a mechanism known as *projection*, the compiler automatically extracts individual programs that run on each machine. This approach has been applied to general distributed systems [67] as well as domains like Secure Multiparty Computation (MPC) to guarantee deadlock freedom and simplify reasoning about distributed message flows [93]. Zippel uses choreographic programming in a novel way to match the style of code that cryptographers already write, enable our analyses, and ensure that both prover and verifier have the same view of the transcript and that the Fiat–Shamir transform is correctly applied by construction.

3. Overview

Figure 2 gives a high-level overview of Zippel. First, the developer writes a `.zippel` program, exactly like what we show in Figure 1b. This program describes the protocol with abstract types like **Group** or **Field**, and **witness** or **instance** variables like `x`, `g`, and `h`. Of course, an abstract protocol is not something one can run; we need to know both the concrete groups and fields that we will use in practice, in addition to the instance and witness inputs. These are specified in a wrapper written in Rust as shown in Appendix A. The prover's input is independent of Zippel and developers can specify it at runtime however they like by constructing a Rust hash map that contains the names of variables in the `.zippel` file and their assigned values.

Zippel then compiles the program into an intermediate representation that we call *Graph IR*. This graph has nodes that represent operations in the protocol and edges that show dependencies between them. Zippel uses this graph for three purposes. First, it helps Zippel decide how to schedule the various operations. Second, it allows Zippel to understand which operations should be performed by $\mathcal{P}$ and which by $\mathcal{V}$. Finally, it allows Zippel to perform its completeness and soundness analyses on this single graph, rather than having to reason about a distributed protocol with multiple parties.

In the rest of this paper we describe Zippel's language in more detail (§4), the Graph IR and projection algorithm (§5), the runtime that executes the graph of operations (§6), the static analyses that we do on this graph (§7), and our experience implementing and running 35 proof systems (§9).

4. The Zippel Language

We designed Zippel to make it easy to write cryptographic proof systems by mirroring the syntax typically seen in cryptography papers. In this section we summarize Zippel's syntax and the operators and objects that we support. The full grammar is available in Appendix B.

4.1. Syntax and core language constructs

We will explore Zippel's syntax and core language constructs through a few examples.

Choreographies. As can be seen in Figure 1b, Zippel is a *choreographic* language. That is, the operations of both $\mathcal{P}$ and $\mathcal{V}$ are written in the same file. In Section 5 we will discuss how Zippel automatically determines which operations are performed by each party.

Types. Zippel's type system includes booleans, integers, finite fields, elliptic curve groups, pairing-friendly groups, univariate, multivariate, and multilinear polynomials, as well as vectors of these objects. For example, Figure 3 shows the KZG protocol [71], which uses the **pair** operator over pairing-friendly groups `G1` and `G2` to prove that a degree $N-1$ univariate polynomial `p` of type `Uni<F, N-1>` evaluated at point `x` (of type `F: Scalar<G1,G2>`) outputs `y`.

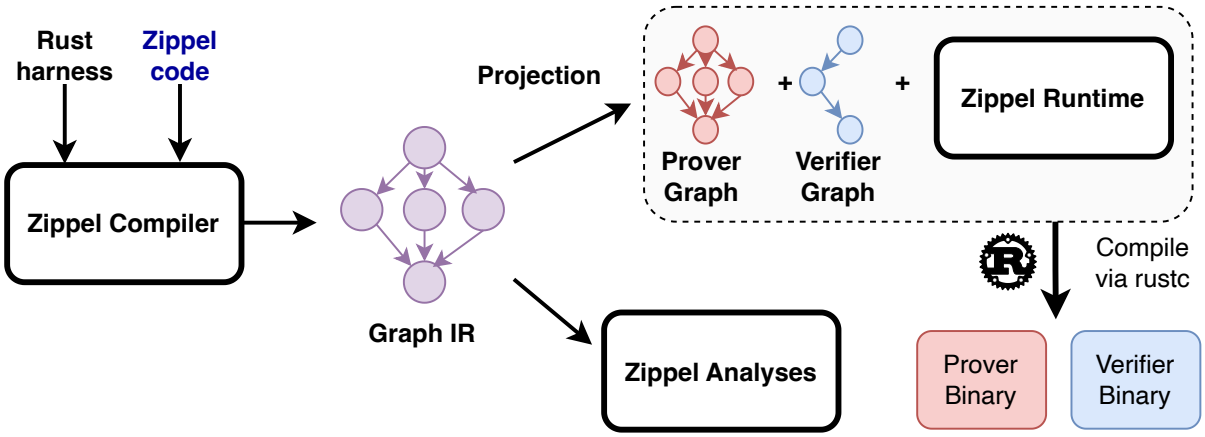


Figure 2: Zippel compiles code into Graph IR and then projects it into separate subgraphs for $\mathcal{P}$ and $\mathcal{V}$. These subgraphs are then lowered into separate binaries that are backed by `arkworks` and Zippel's runtime. The Graph IR supports analyses.

```
proto kzg<N: Size, G1: Group, G2: Group,
  GT: Pairing<G1, G2>, F: Scalar<G1, G2>>
( witness p: Uni<F, N - 1>,
  instance com: G1, instance x: F,
  instance y: F, instance gen1: G1,
  instance gen2: G2, instance gen2_tau: G2,
  extra gen1_taus: [G1; N]
) where
  let tau = random<F>;
  com == dot(coef(p), gen1_taus) &&
  p(x) == y &&
  gen2_tau == tau * gen2 &&
  gen1_taus == [tau^i * gen1 for i in 0..N]
{
  let q = (p - y) / poly([-x, 1]);
  let q_coeffs = coef(q);
  proof <- dot(q_coeffs[0..N - 1],
               gen1_taus[0..N - 1]);
  let lhs = pair(proof, gen2_tau - gen2 * x);
  let rhs = pair(com - y * gen1, gen2);
  verify(lhs == rhs)
}
```

Figure 3: Implementation of the KZG opening proof [71] in Zippel, which proves that polynomial `p` committed by `com` evaluated at point `x` outputs `y`. The generators over different groups are `gen1` and `gen2`. There is a trusted setup (we show the relation in the **where** clause) assumed to have produced the structured reference string objects `gen1_taus` and `gen2_tau` for a secret value τ not known to either $\mathcal{P}$ or $\mathcal{V}$. The expression `tau = random<F>` in the **where** clause should be read as $\exists\tau$. We qualify `gen1_taus` as **extra** because it is not a witness but it is also not part of the transcript in order for the protocol to be succinct.

Indeed, the above already gives an example of one of Zippel's key features: its *sized type system* for containers like vectors (e.g., `[F; N]`) and polynomials (e.g., **Uni**$\langle F, N\rangle$, **Mle**$\langle F, N\rangle$, **Poly**$\langle F, N, D\rangle$). Sized containers combined with sized indices (e.g., `i: Fin<N>`, the type of indices bounded between `0` and `N-1`, which is elided in the code) allow for safe array access and eliminate errors like truncation, incorrect handling of polynomial degrees, working with empty vectors, and blinding with too few random terms, which are common vulnerabilities [21, 23, 24].

Expressions. Zippel supports standard group, field, and polynomial ring arithmetic. In addition, Zippel supports uniform random sampling (**random** `<T>`) and random oracle challenges (**challenge** `<T>`) over a type `T`, with optional exclusions (e.g., **challenge** `<F - [0]>` to indicate sampling of non-zero elements). Zippel also features extensive support for polynomial manipulations. A polynomial in coefficient form can be constructed from a vector of coefficients using **poly**(`c`), and conversely, its coefficients can be extracted via **coef**(`p`). Polynomials can be evaluated on an FFT domain/grid predefined by `arkworks` using **eval**(`p`), or at an explicit point or batch of points via **eval**(`p, x`) or with the syntax `p(x)`. Polynomials can also be reconstructed from evaluations over an FFT grid or explicit coordinate points using Lagrange interpolation via **interpolate**(`x_vec, y_vec`). For multilinear settings, Zippel provides **mle**(`evals`) to compute the multilinear extension of a vector of hypercube evaluations.

Zippel provides many collection manipulation primitives: vector construction (`[...]`), random access (`v[i]`), vector concatenation (`v1 ++ v2`), list comprehensions (`[e for x in v]`), and reductions (**reduce**(`op, v`)) like summing a vector via **reduce**(`+, v`). Additionally, Zippel supports inline polynomial function definitions using the keyword **fun** (e.g., **fun**`(x)=> x*x + 2*x + 1`).

Variable assignment and transcript. Zippel has two types of variable assignment operators. The **let** keyword in combination with the = operator assigns the value of an expression to the variable. The <- operator does three things: (1) it defines the variable; (2) it assigns a value to this variable; and (3) it places the variable in the protocol's transcript. That is, it is available to both $\mathcal{P}$ and $\mathcal{V}$. For example, in Figure 1b the variables u, c, and z are available to both $\mathcal{P}$ and $\mathcal{V}$, with the caveat that $\mathcal{V}$ will not use variables in the transcript that come from the **challenge** operator (e.g., c); it

will recompute them from scratch instead.

Qualifiers. Function and protocol arguments can be marked with **instance**, **witness**, or **extra** qualifiers. These qualifiers control the visibility of inputs and define properties like knowledge soundness, zero-knowledge, and succinctness.

Arguments qualified as **instance** represent the common public inputs visible to both $\mathcal{P}$ and $\mathcal{V}$. Arguments qualified as **witness** represent the prover-only witnesses. Finally, arguments qualified as **extra** are prover-only parameters, but they are not considered part of the witness and are not expected to be extractable by a knowledge extractor. An example is the vector of *powers of tau* (`gen1_taus`) in KZG (Figure 3), which is the output of a trusted setup (not done in Zippel). This vector is large; $\mathcal{V}$ is not expected to use it for its computations even though it is public. The parts of the trusted setup that are relevant to $\mathcal{V}$ and to the correctness of the Fiat–Shamir transform are added as instance variables (`gen1`, `gen2`, `gen2_tau`).

Specification relations. A central feature of Zippel is the ability to specify what the proof system is meant to prove. This is done with the **where** clause in the protocol signature (e.g., **`where`** `h == x*g` in Figure 1). The **where** clause defines the *specification relation* (or precondition) that is assumed to hold on the inputs. Unlike the statements in the protocol body, the **where** clause is declarative and not executed at runtime by $\mathcal{P}$ or $\mathcal{V}$. Instead, Zippel leverages this clause to analyze the properties of the protocol (§7).

Recursion. Zippel does not support control flow constructs like `if` statements or unbounded loops in order to keep type checking and the static analyses simple. However, many proof systems like Sumcheck [78] and Inner Product Arguments [41, 43] are recursive. To support these protocols in Zippel we provide *primitive recursion* via type-level size overloading. In this setting, recursion is structured inductively: the developer defines a base case specialized for a minimal container or parameter size, and a general inductive step defined over a generic size parameter that must be specified at compile time (in Rust, not in Zippel). Because this size parameter strictly decreases at each step and terminates at a concrete base case, the recursion is guaranteed to terminate, enabling fully static, compile-time inlining. Appendix C gives the Sumcheck protocol [78] in Zippel and describes recursion in more detail.

4.2. Type Concretization

To bridge the gap between abstract, size-parameterized choreographic descriptions and executable code, Zippel performs two key concretization transformations:

Size monomorphization. This step lowers recursive, size-parameterized programs into concrete execution pipelines. When the developer supplies the Rust harness (Appendix A), they supply a concrete compile-time initial size (e.g., $N = 10$ variables). Zippel then generates a specialized copy of the recursive function for each integer value within the size range. The compiler then resolves all recursive applications by fully inlining each copy—substituting the recursive call with the copy specialized for the next smaller size—which naturally terminates at the base case. This process simplifies type equality to concrete integer equality and eliminates the need for complex symbolic constraint solving at compile time or dynamic branching at runtime.

Arkworks code specialization. As discussed in Section 3, a Zippel protocol is abstract and generic over elliptic curve groups and finite fields. To execute the protocol, however, the developer specifies the concrete curve/field configuration (e.g., BLS12-381) in the Rust harness using `arkworks` types [2] as shown in Appendix A. Zippel then maps all abstract algebraic types to these concrete Rust types.

4.3. Type Soundness

A benefit of Zippel is its *type soundness* guarantee. Because developers write proof systems using Zippel's sized algebraic types and leave transcript management to Zippel, they avoid many sharp corners (e.g., handwritten transcript indexing, manual array bound checks, polynomial and vector conversions, algebraic type handling). To state this guarantee, we say that a protocol is *well-typed* if it type checks after type concretization (§4.2).

Theorem 4.1 (Type soundness, informal)**.** *Let P be a well-typed Zippel protocol whose inputs have their declared types. Running P always terminates, either with a value of the expected type or with an error caused by a division by zero or a failed* **verify** *assertion. In particular, P never mixes incompatible algebraic types (e.g., adding a field element to a group element), operates on vectors of mismatched lengths, or accesses a container out of bounds.*

The type system, formal semantics, and proof appear in Ioannidis's PhD thesis [68, §7]. The proof does not cover $\mathcal{V}$ running on malformed values sent by $\mathcal{P}$. Zippel's runtime must ensure that values supplied at runtime are valid.

5. Graph IR and Projections

Zippel is a choreographic programming language in which both the prover $\mathcal{P}$ and verifier $\mathcal{V}$ are specified together within a single protocol [82]. While this unified approach is convenient, practical deployment requires extracting two separate executable programs: one for $\mathcal{P}$ and one for $\mathcal{V}$.

This section explains how we map a Zippel program, such as the examples we showed earlier (Figures 1 and 3), into an intermediate representation that we call *Graph IR*, and how we project this graph to get the two programs.

5.1. From Expressions to Graph IR

We construct the Graph IR by lowering the size-monomorphized Zippel declarations (§4.2) into a directed acyclic graph (DAG) that makes all data flows, algebraic and cryptographic operations, and transcript interactions explicit. To do so, the compiler flattens nested AST expressions

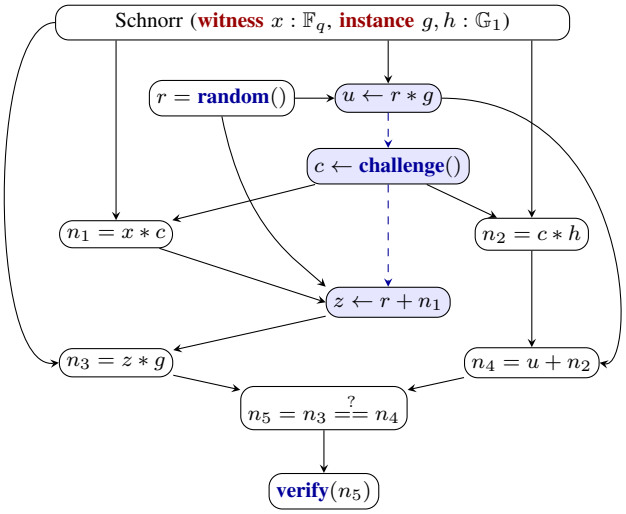


Figure 4: Graph IR for the Schnorr protocol in Figure 1b. Each node represents a flattened algebraic or cryptographic operation, labeled with its variable assignment (using $\leftarrow$ for transcript variables and $=$ for standard local variables, with internal references like n_1 or n_2 to denote dependency inputs). Transcript nodes are highlighted in blue, and dashed blue arrows indicate transcript sequencing. We omit dashed edges between input arguments to keep the figure simple.

into a sequence of simple, single-operator steps where each intermediate value is represented by a unique node. This is effectively *Three-Address Code* (TAC), a standard intermediate representation in compiler design [26, §6.2]. For example, the compiler lowers the expression $r \cdot g + c \cdot h$ into three separate nodes: (1) $n_1 = r \cdot g$, (2) $n_2 = c \cdot h$, and (3) $n_3 = n_1 + n_2$.

The Graph IR for Schnorr's protocol from Figure 1b is shown in Figure 4; we explain this process in detail below.

Definition 5.1 (Graph IR)**.** Let $G = (V, E)$ be a DAG:

- V is a set of nodes representing input arguments, operations (e.g., group or field arithmetic, pairings, polynomial evaluations), random sampling (**random**), challenges (**challenge**), and assertions (**verify**).
- $E \subseteq V \times V$ is a set of directed edges representing data flow, where an edge (v_1, v_2) indicates that the output of node v_1 is consumed as an input by node v_2.

Each node $v \in V$ is annotated with the following metadata:

- **Operation** ($\text{op}(v)$): The operation performed by the node (e.g., $+$, $\cdot$, **pair**, **random**, **challenge**, **verify**). For nodes representing input arguments, $\text{op}(v)$ represents the input's qualifier: **witness**, **instance**, or **extra**.
- **Data type** ($\text{type}(v)$): The sized algebraic type of the node's output (e.g., **Field**, **Group**, **Uni**$\langle F, N\rangle$, **Mle**$\langle F, N\rangle$, **Poly**$\langle F, N, D\rangle$, vectors $[F; N]$).
- **Transcript status** ($\text{tran}(v) \in \{\text{true}, \text{false}\}$): Tracks whether the node's value is part of the protocol transcript. This is true if $\text{op}(v) =$ **challenge** or if the value is explicitly appended to the transcript via the <– operator, and false otherwise.
- **Visibility** ($\text{vis}(v) \in \{\text{private}, \text{public}\}$): Tracks whether the node's value is private to the prover or visible to both parties. Since G is a DAG, the visibility of each node $v \in V$ is uniquely determined by a topological traversal of the graph according to the following rules:
  1) **Transcript rule**: If $\text{tran}(v) = \text{true}$, then $\text{vis}(v) = \text{public}$. This rule takes precedence over all others.
  2) **Source rule**: If v is a source node (no incoming edges in E), then $\text{vis}(v) = \text{private}$ if $\text{op}(v) \in \{$**witness**, **extra**, **random**$\}$, else $\text{vis}(v) = \text{public}$.
  3) **Taint propagation rule**: If v is an operation node (has incoming edges in E): $\text{vis}(v) = \text{private}$ if $\exists$ parent $u \in V$ (where $(u, v) \in E$) such that $\text{vis}(u) = \text{private}$; $\text{vis}(v) = \text{public}$ otherwise.

To guarantee correctness during code generation, the Graph IR must satisfy two well-formedness properties:

Instance and transcript sequencing. The sequential order in which elements are added to the cryptographic sponge is critical to the correctness of the Fiat–Shamir transform. To preserve this ordering, we introduce implicit dependency edges among input **instance** arguments and transcript nodes.

Input arguments are ordered based on their definition order in the Zippel protocol. Transcript nodes are sequenced based on program order. For example, if $t_1, t_2, \ldots, t_k$ are the transcript nodes ($\text{tran}(t_i) = \text{true}$) in program order, we inject edges (t_i, t_{i+1}) for each $1 \leq i < k$. We denote the set of all sequencing edges by E_s and add them to E. In the Graph IR, these sequencing edges are drawn as dashed blue arrows to distinguish them from solid black data-flow edges. To keep Figure 4 simple, we group all input arguments into a single node, and omit their individual nodes and dashed sequencing edges, but they exist.

Acyclicity. Since Zippel lacks arbitrary control-flow loops and resolves all recursive calls at compile time via size monomorphization, the graph G is acyclic by construction.

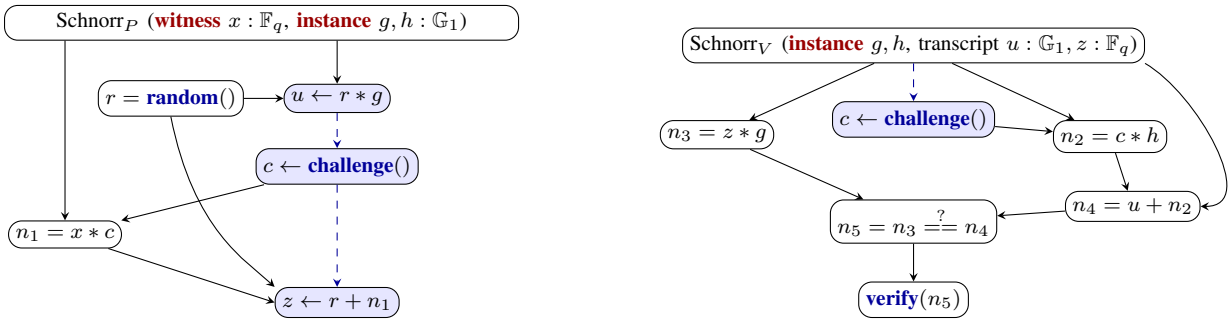
5.2. Projection Algorithms

This section describes how Zippel projects the joint graph $G = (V, E)$ into separate prover and verifier subgraphs, which can then be run as separate programs.

Prover projection. The objective of the prover program is to compute all values that must be transmitted to the verifier. Let $T_{\mathcal{P}} \subseteq V$ denote the set of *prover target nodes* in G, representing all variables that are written to the transcript:

$$T_{\mathcal{P}} = \{v \in V \mid \text{tran}(v) = \text{true}\}$$

To evaluate these target nodes, the prover must compute all of their transitively required data dependencies. This dependency structure is captured by the ancestor closure in the DAG. Let $\text{Ancestors}_G(v) = \{w \in V \mid w \xrightarrow{*} v\}$ denote the set of all nodes in V from which there is a directed path of edges (data-flow or sequencing) to v (the relation is reflexive, meaning $v \in \text{Ancestors}_G(v)$).



(a) Projected prover subgraph.

(b) Projected verifier subgraph.

Figure 5: Projections of Figure 4 to prover and verifier subgraphs. Dashed edges between inputs exist but are omitted.

Definition 5.2 (Prover subgraph)**.** Given $G = (V, E)$, the projected prover subgraph $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}})$ is defined by:

$$V_{\mathcal{P}} = \bigcup_{t \in T_{\mathcal{P}}} \text{Ancestors}_G(t)$$
$$E_{\mathcal{P}} = E \cap (V_{\mathcal{P}} \times V_{\mathcal{P}})$$

As shown in Figure 5, the resulting prover subgraph $G_{\mathcal{P}}$ contains all inputs, random blinds, and intermediate computations required to generate the transcript. The prover program executes this subgraph in topological order.

Verifier projection. The verifier's goal is to check **verify** nodes and replay the entire transcript, without access to nodes that have private visibility. The verifier target nodes are therefore $T_{\mathcal{V}} \subseteq V$:

$$T_{\mathcal{V}} = \{v \in V \mid \text{op}(v) = \textbf{verify} \vee \text{tran}(v) = \text{true}\}$$

The inputs to the verifier's graph are the **instance** arguments and the transcript. However, we need to exclude **challenge** values because it is not safe for the verifier to blindly trust the challenges in the transcript; instead, $\mathcal{V}$ must recompute them from scratch. We therefore define the verifier's *source set* $S_{\mathcal{V}} \subseteq V$ as:

$$S_{\mathcal{V}} = \left\{ v \in V \;\middle|\; \begin{array}{c} \text{op}(v) = \textbf{instance} \\ \vee \\ \text{tran}(v) = \text{true} \wedge \text{op}(v) \neq \textbf{challenge} \end{array} \right\}$$

When traversing backwards from the verifier targets $T_{\mathcal{V}}$, this source set $S_{\mathcal{V}}$ acts as a vertex cut that terminates the reachability analysis along data-flow edges. To capture this cut, we define a modified graph $G' = (V, E')$ by removing all incoming data-flow edges to the source set $S_{\mathcal{V}}$ (sequencing edges are kept so that $\mathcal{V}$ processes the transcript in the same order as $\mathcal{P}$):

$$E' = E \setminus \big((V \times S_{\mathcal{V}}) \setminus E_s\big)$$

Definition 5.3 (Verifier subgraph)**.** Given $G = (V, E)$, the projected verifier subgraph $G_{\mathcal{V}} = (V_{\mathcal{V}}, E_{\mathcal{V}})$ is defined by:

$$V_{\mathcal{V}} = \bigcup_{t \in T_{\mathcal{V}}} \text{Ancestors}_{G'}(t)$$
$$E_{\mathcal{V}} = E' \cap (V_{\mathcal{V}} \times V_{\mathcal{V}})$$

Under this definition, the nodes $S_{\mathcal{V}} \cap V_{\mathcal{V}}$ serve as the input arguments to $\mathcal{V}$'s execution. For instance, in the projected Schnorr verifier graph shown in Figure 5, the source inputs are the **instance** arguments g, h and the transcript variables u, z, while the challenge c is recomputed locally. $\mathcal{V}$'s graph contains only the verifier-side intermediate computations, the locally recomputed **challenge**, and the target nodes $T_{\mathcal{V}}$; private nodes ($\text{vis}(v) = \text{private}$) are omitted.

5.3. Information-Flow Analysis

Section 7 will cover novel analyses for establishing completeness and (special) soundness. But here we want to provide a much simpler example that conveys why Graph IR is an excellent computational model and analysis substrate.

The goal of this analysis is to detect whether $\mathcal{P}$ is directly sending **witness** or **extra** variables to $\mathcal{V}$. Note that **this is not a zero-knowledge analysis** (or even close!). Instead, it is a simple analysis that flags common mistakes. Without this analysis, AI agents would often produce code that accessed the witness directly from $\mathcal{V}$, since that is the easiest way to satisfy both completeness and soundness!

Analysis. $\mathcal{V}$ must not rely on private inputs to complete its verification, so Zippel checks the visibility of nodes in $V_{\mathcal{V}}$:

$$\text{Safe-Visibility}(G_{\mathcal{V}}) \iff \forall v \in V_{\mathcal{V}},\ \text{vis}(v) \neq \text{private} \wedge \text{op}(v) \notin \{\textbf{witness}, \textbf{extra}, \textbf{random}\}$$

If a developer mistakenly adds a **witness**, **extra**, or **random** variable (such as x or r in Schnorr) to the transcript or this variable is reachable by $\mathcal{V}$'s ancestor traversal without being blocked by a node in $S_{\mathcal{V}}$ (e.g., if $\mathcal{V}$ directly references x in one of its operations), this indicates an information-flow violation. In such cases, Zippel outputs an error.

Our implementation extends this analysis to track the distribution of values sent from $\mathcal{P}$ to $\mathcal{V}$ (e.g., uniform, non-uniform) to flag potential leaks of private information. We omit the details since we cannot prove anything formal about this extension, but we have found it to be helpful in practice.

6. Runtime System

Zippel's projection produces graphs for $\mathcal{P}$ and $\mathcal{V}$. In this section, we describe how Zippel's runtime schedules the

nodes in each graph and how it evaluates them. Appendix A covers how to integrate Zippel into a larger application.

6.1. Node Scheduling

At a high level, every node in the graph is either ready to run or waiting for some dependency to be fulfilled. Ready nodes have no incoming (data or sequencing) edges. Zippel dispatches ready nodes in parallel using a combination of Kahn's topological sort algorithm [70] and a work-stealing thread pool [17]. As nodes complete their execution in the graph, Zippel's runtime decrements the in-degree of their successors using lock-free atomic instructions, eventually making more nodes ready for processing.

Correctness and efficiency. Since the Graph IR contains explicit sequencing edges (t_i, t_{i+1}) between transcript nodes and projection preserves this (as detailed in Section 5.2), transcript nodes lie on a single path. Consequently, any valid topological sort guarantees that transcript nodes are executed in program order. Parallelism does not compromise correctness because a subsequent transcript node t_{i+1}'s in-degree only drops to zero (making it ready) after the preceding transcript node t_i has completed execution. Finally, our scheduling implementation inherits the optimal $O(|V|+|E|)$ scheduling overhead of Kahn's algorithm, with execution time dominated by the concrete operations in the nodes.

6.2. Node Evaluation and Automated Fiat–Shamir

When a node is ready and is dispatched to a worker thread for processing, Zippel executes it as follows.

Non-transcript nodes. Zippel evaluates the algebraic operations by invoking the corresponding routines from the `arkworks` library. To avoid copying large field or group vectors across threads, our execution engine stores node results in write-once, read-many cells that successor nodes access via read-only shared pointers. For finite-field ($\text{op}(v) \in \{+, \cdot, \text{inv}, \dots\}$) and elliptic-curve group arithmetic, we use the `ark-ff` and `ark-ec` libraries. We generally lower operations over vector inputs to standard Rust `Iterators`, except in performance-critical cases: for example, inner products of field and group vectors use `arkworks`' optimized multi-scalar multiplication (MSM), and scalar vector inversions use batch inversion. Finally, we lower operations over polynomials (including univariate addition, multiplication, division, single/multi-point evaluation via FFTs, and multivariate/multilinear operations) to the specialized subroutines in `arkworks`' `ark-poly` library.

Input and transcript nodes. Zippel automates the application of the Fiat–Shamir transform [61] by managing the transcript and the cryptographic sponge.

- *Sponge initialization:* Prior to executing the graph, both $\mathcal{P}$ and $\mathcal{V}$ initialize their own cryptographic duplex sponge [20, 49] and absorb all **instance** inputs in program order (dictated by the sequencing edges).
- *Prover execution:* When $\mathcal{P}$ evaluates a node v where $\text{tran}(v) = \text{true}$, Zippel performs the node's operation, absorbs the resulting value into the sponge, and appends the value to the transcript. If $\text{op}(v) =$ **challenge**, Zippel instead squeezes the sponge.
- *Verifier execution:* When $\mathcal{V}$ evaluates a node v where $\text{tran}(v) = \text{true}$, Zippel reads the corresponding value from the transcript and absorbs it into the sponge. If $\text{op}(v) =$ **challenge**, Zippel instead squeezes the sponge to recompute the challenge.

Since both parties execute their sponge operations in the order specified by the Graph IR's sequencing edges, they include all **instance** inputs and maintain synchronized sponge states that capture all transcript nodes. This eliminates weak Fiat–Shamir attacks [39, 56, 83] by construction.

7. Analysis

Prior works use computer algebra to automatically analyze cryptographic assumptions and signatures [28, 37]. This led us to ask: could we also use similar techniques to analyze properties of cryptographic proof system protocols, like completeness and soundness, which prior works had not considered? The answer turns out to be yes. This section describes how Zippel confirms that a protocol satisfies perfect completeness and $(\ell_1, \dots, \ell_\mu)$-special soundness [32].

Both of our novel analyses take a common approach: encoding Zippel's semantics as the zeros of multivariate polynomials and reasoning about them with Gröbner bases. This approach is suitable because both perfect completeness and special soundness are properties that are neither probabilistic nor reliant on computational assumptions. We begin with some background and then describe the analyses.

7.1. Polynomial Ideals and Gröbner Bases

This section covers the math background for our analyses at a high level; the details can be found elsewhere [50].

Varieties. Let $\mathbb{F}$ be a finite field, let $\mathbf{v} = (v_1, \dots, v_n)$ be variables, and let $\mathbb{F}[\mathbf{v}]$ be the set of polynomials in $\mathbf{v}$ with coefficients from $\mathbb{F}$. A set of polynomials G defines a *polynomial system* of equations which is satisfied at an input $\mathbf{v}$ when every polynomial in G is zero. Let $\mathbb{V}(G)$ be the set of zeros of G: $\mathbb{V}(G) = \{\mathbf{v} \in \mathbb{F}^n : \forall p \in G, p(\mathbf{v}) = 0\}$. To repeat: the variety $\mathbb{V}(G)$ is the set of all *solutions* to G.

In Zippel's static analyses, completeness and special soundness will be expressed as *variety containment* questions. A variety containment $\mathbb{V}(G) \subseteq \mathbb{V}(G')$ means that every solution to the polynomial system G is also a solution to G'. However, checking variety containment requires solving systems of polynomial equations, which is computationally hard. We sidestep this cost by looking instead at *ideals*.

Ideals. Given a set of polynomials $G = \{p_1, \dots, p_s\}$, the *ideal* generated by G, denoted $\langle G \rangle$, is the set of polynomials:

$$\langle G \rangle = \left\{ \textstyle\sum_{i=1}^{s} r_i p_i \;\middle|\; r_i \in \mathbb{F}[\mathbf{v}] \right\}$$

That is, $\langle G \rangle$ contains all polynomials constructible by scaling the elements of G by arbitrary polynomials and adding

them. The key fact is that G and its generated ideal $\langle G \rangle$ have exactly the same solutions: $\forall G, \mathbb{V}(G) = \mathbb{V}(\langle G \rangle)$. Thus, we can view $\langle G \rangle$ as the *algebraic consequences* of G: other polynomials that are also zero on G's solutions.

This correspondence between varieties and ideals allows us to reduce questions about variety containment to ideal containment. Specifically, to show that $\mathbb{V}(G) \subseteq \mathbb{V}(G')$, it suffices to show that $\langle G' \rangle \subseteq \langle G \rangle$. Unlike solving polynomial systems, checking ideal containment does not require finding any solutions. Instead, because $\langle G' \rangle$ is generated by G', showing $\langle G' \rangle \subseteq \langle G \rangle$ is equivalent to showing that each polynomial $p \in G'$ is in $\langle G \rangle$. This is known as the *ideal membership problem*, and it is a simpler algebraic check.

The downside is that ideal containment is a *conservative approximation* of variety containment. Because finite fields are not algebraically closed, a polynomial system can satisfy variety containment ($\mathbb{V}(G) \subseteq \mathbb{V}(G')$) even when the corresponding ideal containment ($\langle G' \rangle \subseteq \langle G \rangle$) does not hold. As a result, Zippel may fail to prove a property (e.g., completeness, soundness) for a protocol that is correct.

Gröbner bases (GBs). One way to solve the ideal membership problem and show that $p \in \langle G \rangle$ is to repeatedly use polynomials in G as rewrite rules. At each step, we subtract a suitable multiple of some polynomial in G to cancel a term of p. If this process eventually rewrites p to 0, then p is an algebraic consequence of G. This rewriting process is called *reduction by* G, and is written as $\texttt{reduce}(p, G) \to p'$. Unfortunately, when G is an arbitrary set, reduction can fail to reach 0 even when p belongs to the ideal: reduction by G is a sound but incomplete test for ideal membership.

A Gröbner basis is a specially chosen set of polynomials B that generates the same ideal as G ($\langle B \rangle = \langle G \rangle$) and guarantees that reduction by B is a complete test: any polynomial p reduces to 0 if and only if it belongs to the ideal $\langle B \rangle$. Formally, if a polynomial set B is a Gröbner basis, then $p \in \langle B \rangle$ implies $\texttt{reduce}(p, B) = 0$. Furthermore, there exists an algorithm, $\texttt{GBasis}(G) \to B$, that returns a Gröbner basis for any $\langle G \rangle$. One caveat is that neither `reduce` nor `GBasis` is polytime in the worst case, but in many settings they are efficient enough in practice [86, 96].

Elimination. In Zippel's analyses, we often care about a subset of the variables in a polynomial system. For example, we may wish to learn the constraints on $\mathcal{V}$'s checks while abstracting away $\mathcal{P}$'s internal variables. Elimination helps us do this. Let $\mathbf{v}'$ be a subset of the variables $\mathbf{v}$. For set G,

$$\langle G \rangle \cap \mathbb{F}[\mathbf{v} \setminus \mathbf{v}']$$

is the *elimination ideal* obtained by eliminating $\mathbf{v}'$. It consists of all algebraic consequences of G expressible without the variables $\mathbf{v}'$. Thus, elimination is a lossy projection: information involving eliminated variables is discarded, while all consequences expressible in the remaining variables are preserved. Computing elimination ideals directly is difficult. Fortunately, Gröbner bases expose them automatically when computed using a suitable *monomial order*. Unlike univariate polynomials (where terms are ordered by exponent degree such as $x^3 > x^2 > \dots$), multivariate terms have no single natural order. A monomial order is a rule that systematically decides which term in a polynomial is the largest (the leading term) during reduction.

An *elimination order* for $\mathbf{v}'$ is a monomial order in which monomials containing variables in $\mathbf{v}'$ are larger than those that do not. If a GB B for $\langle G \rangle$ is computed using such an order, then the polynomials in B that do not contain variables from $\mathbf{v}'$ form a GB for the elimination ideal.

One common elimination order is the *lexicographic* order. For example, if the variables are ordered $v_1 > v_2 > \dots > v_n$, then a lexicographic GB exposes the elimination ideal obtained by eliminating any prefix $v_1, \dots, v_k$.

7.2. From Graph IR to Ideals

We encode a Graph IR as a polynomial ideal, represented by a set of generator polynomials $G \subseteq \mathbb{F}[\mathbf{v}]$. The encoding is defined such that $\mathbb{V}(G)$ (the set of zeros of G) is in one-to-one correspondence with the successful executions of the graph. Thus, by proving properties about $\mathbb{V}(G)$, one proves properties about all executions of the graph.

To construct G, we map a node's value to one or more variables in $\mathbf{v}$, and each operation to a set of polynomials.

Variable Encoding. We map Graph IR values to polynomial variables according to their algebraic types:

- *Field elements* map to a single variable in $\mathbf{v}$.
- *Group elements* map to a single variable in $\mathbf{v}$ representing their discrete logarithm. Note that we will not need to find this discrete logarithm to analyze the graph.
- *Vectors and polynomials* map to a collection of variables in $\mathbf{v}$ representing their individual entries, coefficients, or evaluations.

Operation Encoding. Each Graph IR operation node is mapped into a set of polynomials whose zeros encode correct execution of that operator. For example:

- A field addition $z = x + y$ maps to $z - x - y$.
- A field multiplication $z = x \cdot y$ maps to $z - x \cdot y$.
- A scalar multiplication of a field element x and a group element g (with discrete logarithm y) yielding a group element h (with discrete logarithm z) maps to $z - x \cdot y$.
- A bilinear pairing of group elements g, h (with discrete logarithms x, y) yielding a group element p (with discrete logarithm z) maps to $z - x \cdot y$.
- A field inverse $y = \text{inv}(x)$ maps to $x^2 y - x$ and $xy^2 - y$.
- A verifier assertion $\text{Verify}(x = y)$ maps to $x - y$.
- An FFT operation mapping inputs $\mathbf{v}_{\text{in}}$ to outputs $\mathbf{v}_{\text{out}}$ is encoded by the rows of $M_n \cdot \mathbf{v}_{\text{in}} - \mathbf{v}_{\text{out}}$, where M_n is the constant FFT matrix.

All Zippel operations can be encoded as polynomial sets, sometimes with auxiliary variables that are added to $\mathbf{v}$.

Notation. We group our variables $\mathbf{v}$ into sub-vectors depending on the type of the node that they encode.

- $\mathbf{x}$ denotes nodes v where $\text{op}(v) \in \{\textbf{instance}, \textbf{extra}\}$.
- $\mathbf{w}$ denotes nodes v where $\text{op}(v) = \textbf{witness}$.
- $\mathbf{r}$ denotes nodes v where $\text{op}(v) = \textbf{random}$.
- $\mathbf{m}$ denotes transcript messages (appended via <–).

- $\mathbf{c}$ denotes verifier challenges (**challenge** outputs).
- $\mathbf{t}$ denotes intermediate (temporary or auxiliary) variables. We subdivide these into prover-only ($\mathbf{t}_\mathcal{P}$), verifier-only ($\mathbf{t}_\mathcal{V}$), and relation-only ($\mathbf{t}_R$) variables.

We let P, V, and R denote the polynomials representing the prover graph, verifier graph, and specification relation (**where** clause), respectively. The variety $\mathbb{V}(R)$ contains the inputs $(\mathbf{x}, \mathbf{w}, \mathbf{t}_R)$ that satisfy the relation. The variety $\mathbb{V}(P)$ contains the variable values $(\mathbf{x}, \mathbf{w}, \mathbf{r}, \mathbf{c}, \mathbf{m}, \mathbf{t}_\mathcal{P})$ that encode successful prover executions. And the variety $\mathbb{V}(V)$ contains the public and transcript values $(\mathbf{x}, \mathbf{c}, \mathbf{m}, \mathbf{t}_\mathcal{V})$ that encode successful verifier executions.

7.3. Perfect Completeness

This section describes Zippel's analysis for perfect completeness, which holds if $\mathcal{V}$ always accepts when interacting with an honest $\mathcal{P}$ on inputs satisfying the relation. We can express this condition in the language of varieties:

$$\begin{aligned} &\forall \mathbf{x}, \mathbf{w}, \mathbf{m}, \mathbf{c}, \mathbf{r}, \mathbf{t}_\mathcal{P}, \mathbf{t}_R, \quad \exists \mathbf{t}_\mathcal{V}, \\ &\mathbf{v} \in \mathbb{V}(P) \cap \mathbb{V}(R) \implies \mathbf{v} \in \mathbb{V}(V) \end{aligned} \tag{1}$$

This says that for any valid prover execution trace ($\mathbb{V}(P)$) on inputs satisfying the relation ($\mathbb{V}(R)$), there exists a matching verifier execution trace ($\mathbf{t}_\mathcal{V}$) that is accepting ($\mathbb{V}(V)$). $\mathbf{t}_\mathcal{V}$ is existentially quantified because it records values computed inside $\mathcal{V}$. For example, suppose $\mathcal{V}$ receives transcript values x and y, sets z to be $x + y$, and checks that $x = y$ and $z = 2$. Its polynomial encoding contains three equations:

$$z - x - y = 0, \qquad x - y = 0, \qquad z - 2 = 0.$$

Here, x and y are supplied by the transcript, while z is computed by the verifier and is therefore an intermediate variable ($\mathbf{t}_\mathcal{V} = (z)$). When the verifier receives $x = 1$ and $y = 1$, it computes $z = 2$ and accepts: there exists a value of z satisfying all three equations. Universally quantifying z would require these equations to hold also for $z = 3$ (not possible), even though the verifier computes $z = 2$.

Although (1) resembles a variety containment condition, it is not one, because of the existential quantifier over the verifier's intermediate variables ($\mathbf{t}_\mathcal{V}$). This makes it hard to check directly. We must eliminate this existential.

We do this by leveraging Zippel's determinism. The polynomials in V come from two kinds of verifier nodes: nodes that compute an intermediate value (e.g., $z - x - y$ above) and **verify** assertions (e.g., $x - y$ and $z - 2$). Let $I_V \subseteq V$ be the defining polynomials of the intermediate variables $\mathbf{t}_\mathcal{V}$. In the example, $I_V = \{z - x - y\}$. Because every Zippel operation is deterministic, I_V fixes one value for each intermediate variable once the variables it depends on are fixed; it is the verifier's computation, written as equations. That is, I_V defines a *Skolem function* for $\exists \mathbf{t}_\mathcal{V}$.

We add I_V to the hypothesis, so the implication only requires acceptance when $\mathbf{t}_\mathcal{V}$ agrees with the verifier's computation. With this restriction, we can replace $\exists \mathbf{t}_\mathcal{V}$ with a universal quantifier:

$$\begin{aligned} &\forall \mathbf{x}, \mathbf{w}, \mathbf{m}, \mathbf{c}, \mathbf{r}, \mathbf{t}_\mathcal{P}, \mathbf{t}_R, \mathbf{t}_\mathcal{V}, \\ &\mathbf{v} \in \mathbb{V}(P) \cap \mathbb{V}(R) \cap \mathbb{V}(I_V) \implies \mathbf{v} \in \mathbb{V}(V) \end{aligned} \tag{2}$$

Since $I_V \subseteq V$, the defining polynomials in the conclusion already appear in the hypothesis (and hence they hold trivially), so Condition (2) really tests the assertions: whenever the prover is honest, the relation holds, and the verifier computes its intermediate values correctly, every **verify** passes. Further, (2) has only universal quantifiers, so it is a variety containment condition:

$$\mathbb{V}(P \cup R \cup I_V) \subseteq \mathbb{V}(V) \tag{3}$$

Following Section 7.1, we turn this into ideal containment:

$$\langle V \rangle \subseteq \langle P \cup R \cup I_V \rangle \tag{4}$$

We check this by computing $B = \texttt{GBasis}(P \cup R \cup I_V)$ and verifying that for all $p \in V$, $\texttt{reduce}(p, B) = 0$.

This analysis is sound: if the ideal containment test succeeds, the protocol is guaranteed to be perfectly complete. However, the test can fail or time out, in which case the developer learns nothing. This can happen if the protocol being analyzed has non-zero honest-prover failure probability, as this analysis applies to perfect completeness. It can also fail when ideal containment is too conservative compared to variety containment (§7.1), or when the verifier's intermediate variables cannot be represented as polynomial Skolem functions. Additionally, the worst-case complexity of `GBasis` and `reduce` can cause timeouts. Nevertheless, this analysis works well in practice (§9.3).

Optimizations. We reduce the polynomial system before checking ideal containment using two optimizations.

First, we reuse computations shared between $\mathcal{P}$ and $\mathcal{V}$. Since Zippel is choreographic, both Graph IRs share many nodes. These nodes perform the same computations on the same inputs, so determinism guarantees the same outputs. We therefore encode each shared node once, using the same variables and polynomials in P and V. The shared variables are universally quantified in (1): the prover's equations already constrain their values, which the verifier also uses.

Second, we eliminate variables by solving equations before checking ideal containment in (4). Suppose a generator has the form $g = ax + h(\mathbf{y})$, where x is the variable to eliminate, $\mathbf{y}$ lists the remaining variables, $a \in \mathbb{F} \setminus \{0\}$ is a constant, and $h \in \mathbb{F}[\mathbf{y}]$ is a polynomial depending only on $\mathbf{y}$. The equation $g = 0$ then determines $x = -h(\mathbf{y})/a$. We remove g and substitute this expression for x in both the remaining generators and the polynomials whose membership we test. This removes x and its defining equation without changing the result of the membership tests:

$$p \in \langle G \rangle \iff \phi(p) \in \langle \phi(G \setminus \{g\}) \rangle,$$

where ϕ denotes the substitution $x \mapsto -h(\mathbf{y})/a$.

7.4. Generalized Special Soundness

Recall from Section 2.3 that Schnorr is an ℓ-special sound protocol because we can extract a witness from $\ell = 2$ accepting transcripts that have the same commitment but distinct challenges. To automatically analyze this property, we instantiate ℓ copies of the verifier's constraints V_i for $i \in \{1, \ldots, \ell\}$. The public inputs $\mathbf{x}$ and witnesses $\mathbf{w}$ are shared across all copies. We partition the transcript messages $\mathbf{m}$ into the first message $\mathbf{m}_{\text{shared}}$ (e.g., commitment) that is shared across all copies and the second message $\mathbf{m}_i$ (e.g., the response), which is not shared. The challenges $\mathbf{c}_i$ and verifier intermediate variables $\mathbf{t}_{\mathcal{V},i}$ are also copy-specific. In what follows, we assume that the challenge is a single variable c_i for simplicity, but Appendix D.1 lifts this restriction to allow for any challenge type.

To enforce the requirement that the challenges in these copies are distinct, we define the polynomial constraint set:

$$D = \{d_{i,j}(c_i - c_j) - 1\}_{1 \le i < j \le \ell} \tag{5}$$

where $d_{i,j}$ are auxiliary variables representing the multiplicative inverses of the differences between challenges.

Our analysis aims to construct a candidate extractor, represented as a set of polynomials $E = \{p_j\}_j$ (one for each witness variable $w_j \in \mathbf{w}$). Specifically, we want each extraction polynomial p_j to be of the form:

$$w_j - f_j(\mathbf{v}_\mathcal{V})$$

where $\mathbf{v}_\mathcal{V}$ are all verifier-visible variables (i.e., they exclude $\mathbf{w}$ and $\mathbf{t}_R$).

For the candidate extractor to be correct, it must reconstruct a valid witness $\mathbf{w}$ for any tuple of ℓ accepting transcripts with distinct challenges. In the language of varieties:

$$\begin{aligned} &\forall \mathbf{x}, \mathbf{w}, \mathbf{m}_{\text{shared}}, c_i, \mathbf{m}_i, d_{i,j}, \mathbf{t}_{\mathcal{V},i}, \quad \exists \mathbf{t}_R, \\ &\mathbf{v} \in \mathbb{V}(V_1 \cup \cdots \cup V_\ell \cup D \cup E) \implies \mathbf{v} \in \mathbb{V}(R) \end{aligned} \tag{6}$$

This says that for any set of ℓ accepting transcripts ($\mathbb{V}(V_i)$) with distinct challenges ($\mathbb{V}(D)$), for the witness computed by the candidate extractor ($\mathbb{V}(E)$), there must be a trace ($\mathbf{t}_R$) that satisfies the protocol's relation ($\mathbb{V}(R)$). It remains for us to construct a candidate extractor E and verify that it satisfies the above condition.

To find the candidate extractor E, we attempt to solve for the witness variables $\mathbf{w}$ using a reduced GB B for the ideal $\langle V_1 \cup \cdots \cup V_\ell \cup D \cup R \rangle$.[1] We compute B under an elimination order for the relation's intermediate variables $\mathbf{t}_R$ and the witness variables $\mathbf{w}$ that is lexicographic among these variables (i.e., $\mathbf{t}_R > \mathbf{w} >$ others). We eliminate $\mathbf{t}_R$ too because these variables are not visible to the verifier.

Within B, we search for polynomials that match the form above for all witness variables $w_j \in \mathbf{w}$. That is: $w_j - f_j(\mathbf{v}_\mathcal{V})$. Alternatively, we also match polynomials of the form $m(w_j - f_j(\mathbf{v}_\mathcal{V}))$, where m is any group-typed monomial. These polynomials arise when a verification condition over group elements is used to check a scalar equality "in the exponent". In this case, we divide out m and take $p_j = w_j - f_j(\mathbf{v}_\mathcal{V})$. Then, we also require f_j to be evaluable without computing any discrete logarithms. (For example, if w_j represents a field element, f_j cannot be group-typed.) Ultimately, if for any w_j, no p_j can be found, then the analysis fails.

Finally, our analysis must verify that the candidate extractor E actually reconstructs a valid witness. That is, we must check (6). Just like in our completeness analysis, we simplify this to an ideal containment check by eliminating the existential quantifier over $\mathbf{t}_R$:

$$\langle R \rangle \subseteq \langle V_1 \cup \cdots \cup V_\ell \cup D \cup E \cup I_R \rangle \tag{7}$$

where I_R represents Skolem functions for the relation's intermediate variables $\mathbf{t}_R$. This check concludes the analysis.

This analysis is sound, but it can fail on correct protocols for the same reasons as the completeness analysis. Further, it may fail to find an extractor even if one exists. This is possible because the analysis only finds polynomial extractors—a subset of all efficient algorithms.

Generalization. Our analysis generalizes beyond 3-move Σ-protocols to $(2\mu + 1)$-move protocols that satisfy $(\ell_1, \ldots, \ell_\mu)$-special-soundness [32]. An example of this would be a sequentially iterated version of Schnorr. Appendix D.2 gives the details.

7.5. Encoding Failures

So far, our encoding has assumed that every operation succeeds. However, some operations can abort: inversion and polynomial division fail on a zero divisor, and interpolation fails on repeated points. Consider this toy protocol with a trivial relation:

```
proto toy<F: Field>() where 1 == 1 {
  x <- challenge<F>;
  y <- 1 / x;
  verify(x * x * y == x)
}
```

When the challenge x is zero, which happens with probability $1/|\mathbb{F}|$, the honest prover aborts, so the protocol is not perfectly complete. Yet the analysis of Section 7.3 certifies it. To see why, recall that the encoding of `y <- 1 / x` (Section 7.2) consists of two equations:

$$x^2 y - x = 0, \qquad xy^2 - y = 0.$$

When $x \neq 0$, they force $y = 1/x$; when $x = 0$, they force $y = 0$. The encoding thus treats $1/0$ as 0 rather than as an abort. The verifier's assertion is $x^2 y - x = 0$, which is one of the prover's two equations, so Condition (4) holds.

Definedness conditions. To detect aborts, Zippel attaches a *definedness condition* to each operation that can fail. This is a polynomial that is zero on the solutions of the encoding where the operation succeeds, and nonzero on the spurious solutions that stand in for an abort. For $y = \text{inv}(x)$, the definedness condition is $xy - 1$: it is 0 when $y = 1/x$, and -1 at the spurious solution $(x, y) = (0, 0)$. Polynomial

1. In a *reduced* GB, no monomial in any polynomial is divisible by another polynomial's leading term.

division needs a divisor that is not the zero polynomial, so its definedness condition is $s - 1$, where s is an auxiliary variable that the encoding sets to 1 when some coefficient of the divisor is nonzero, and to 0 otherwise. Interpolation through points $x_1, \ldots, x_n$ divides by every difference $x_i - x_j$. Its definedness condition therefore checks that the points are distinct, using $(x_i - x_j)u_{ij} - 1$ with a fresh variable u_{ij} for each pair. We write DC_P, DC_V, and DC_R for the sets of definedness conditions of the operations in the prover graph, the verifier graph, and the relation, respectively.

Checking completeness. Condition (4) has two sides. The right-hand side, $P \cup R \cup I_V$, describes what completeness takes as given: the prover runs honestly, the inputs satisfy the relation, and the verifier computes its intermediate values. The left-hand side, V, describes what completeness must show: every **verify** assertion passes. Each set of definedness conditions goes on the right-hand side if completeness may take the success of its operations as given, and on the left-hand side if completeness must show it:

- The prover's conditions, DC_P, go on the left, because completeness requires the honest prover never to abort.
- The verifier's conditions, DC_V, also go on the left, because a verifier that aborts does not accept.
- The relation's conditions, DC_R, go on the right, because completeness considers only inputs that satisfy the relation, and a relation such as `where y == 1 / x` holds only if its divisions succeed (here, only if $x \neq 0$).

Condition (4) becomes:

$$\langle V \cup DC_P \cup DC_V \rangle \subseteq \langle P \cup R \cup DC_R \cup I_V \rangle \qquad (8)$$

In the toy protocol, $DC_P = \{xy - 1\}$, and the other two sets are empty. Every polynomial on the right-hand side vanishes when $x = y = 0$, so every polynomial in their ideal does too. But $xy - 1$ is -1 there, so it is not in the ideal, and the analysis correctly refuses to certify the protocol.

Checking soundness. The extractor validation check of Section 7.4, Condition (7), also has two sides, but the verifier and the relation trade places. The right-hand side describes what soundness takes as given: ℓ accepting transcripts with distinct challenges, the witness that the candidate extractor computes from them, and the relation's intermediate values. The left-hand side, R, describes what soundness must show: the extracted witness satisfies the relation. The same rule places the definedness conditions:

- The verifier's conditions go on the right, because soundness considers only accepting transcripts, and a verifier that aborts does not accept. Each transcript i gets its own copy, DC_{V_i}, just as it gets its own copy V_i of the verifier's polynomials.
- The relation's conditions, DC_R, go on the left, because the extracted witness must satisfy the relation, and the relation holds only if its divisions succeed.
- The prover's conditions, DC_P, do not appear, just as P does not: the transcripts may come from a malicious prover that does not run the honest prover's code.

Condition (7) becomes:

$$\langle R \cup DC_R \rangle \subseteq \langle \bigcup_{i=1}^{\ell} (V_i \cup DC_{V_i}) \cup D \cup E \cup I_R \rangle \qquad (9)$$

The search for a candidate extractor (Section 7.4) follows the same rule, but it has only one side: it takes everything it uses as given, including the relation. So each verifier copy and the relation bring their definedness conditions along, and the search computes B for the ideal generated by

$$\bigcup_{i=1}^{\ell} (V_i \cup DC_{V_i}) \cup D \cup R \cup DC_R.$$

The candidate extractor E then has to pass Condition (9).

For example, a relation that divides by k, such as `where h == g * (x / k)`, holds only if $k \neq 0$. The search may assume $k \neq 0$ when it solves for the witness, but Condition (9) holds only if its right-hand side implies $k \neq 0$. If the verifier computes `1 / k`, for instance, it rejects every transcript with $k = 0$, and the definedness condition of this division, in each DC_{V_i}, supplies $k \neq 0$. But if the verifier accepts ℓ transcripts with distinct challenges when $k = 0$, Condition (9) does not hold, and the analysis refuses to certify the protocol. This is right: no witness x satisfies the relation when $k = 0$, so there is nothing to extract from these transcripts, and the protocol is not special sound.

8. Implementation

Zippel is written in ≈84K lines of Rust. The parser, optimizer, and type checker account for ≈11K lines, the Graph IR and projection algorithm require ≈3K lines, the analyses require ≈7K lines, and the runtime ≈10K lines. The rest are tests, benchmarks, examples, and tooling such as our syntax formatter.

We use Zippel to implement many cryptographic proof systems. These range from simple, discrete-logarithm-based Σ-protocols such as Schnorr [89] and Chaum-Pedersen [46], to SNARKs for NP such as Groth16 [66], Spartan [90], HyperPlonk [47], and Pari [59].

We also implement key components of these protocols, including univariate and multilinear polynomial commitment schemes such as KZG [71], PST13 [87], KZH [69], and Dory [76]; inner product arguments such as BCCGP [41], Bulletproofs [43], and Hyrax [98]; and univariate and multilinear PIOPs such as Sumcheck [78], Zerocheck [47], and Hadamard product checks [65].

A list of protocols we have implemented is provided in Figure 6. Many of these protocols were implemented with AI agents using simple prompts that provided: (1) Zippel's grammar; (2) examples of protocols implemented by hand in Zippel; (3) the paper containing the new protocol we wanted to implement; and (4) commands for running the code with Zippel, as well as its static analyses.

In all cases, we manually reviewed the generated code and iterated as needed to ensure the results were correct. For many of the protocols that had baseline implementations, we also ensured that proofs produced by Zippel's code were verifiable in those separate baseline implementations.

Proof System	LoC	Complete	Spec. Sound
Sumcheck [78]	87	✓ 15.1 ms	
Schnorr [89]	7	✓ 0.7 ms	✓ 27.9 ms
Multi-Schnorr [89]	19	✓ 1.4 ms	✓ 13.4 sec
Pedersen-Eq [88, 89]	13	✓ 0.8 ms	✓ 29.1 ms
Okamoto [84]	15	✓ 1.1 ms	×
Oka.-ElGamal [60, 84]	18	✓ 1.9 ms	✓ 146.1 ms
Chaum-Pedersen [46]	15	✓ 1.1 ms	✓ 54.2 ms
CDS Disjunction [52]	50	✓ 43.9 ms	×
Hadamard [65]	13	✓ 62.2 ms	
E-Cash Coin [80]	95	✓ 20.8 ms	×
KZG [71]	24	✓ 2.5 ms	
MLE Sum [94, 97]	78	✓ 15.5 ms	
PST13 [87]	81	✓ 14.0 ms	
BCCGP IPA [41]	61	✓ 12.9 ms	
Groth16 [66]	76	✓ 31.5 ms	
Bulletproofs [43]	77	✓ 13.6 ms	
Dot Product [98]	56	✓ 2.8 ms	×
Product Proof [98]	31	✓ 3.5 ms	×
Hyrax IPA [98]	88	✓ 19.2 ms	
Hyrax PCS [98]	51	✓ 37.5 ms	
Membership [42]	50	✓ 16.5 ms	
Marlin-KZG [48]	31	✓ 4.7 ms	
Spartan [90]	424	×	
Dory IPA [95]	305	✓ 257.4 ms	
Dory PCS [76]	304	✓ 12.5 sec	
R1CS Σ [44]	54	✓ 19.1 ms	
Multiset [47]	152	✓ 23.0 sec	
Permutation [47]	157	✓ 35.7 sec	
Zerocheck [47]	114	✓ 9.3 sec	
Product-Check [47]	142	✓ 11.1 sec	
HyperPlonk [47]	433	×	
Zeromorph-KZG [73]	31	✓ 4.4 ms	
KZH [69]	50	✓ 41.3 ms	
DeKART [40]	124	✓ 88.8 ms	
Pari [59]	109	✓ 113.6 ms	

Figure 6: Proof systems implemented in Zippel ordered by year of publication. A ✓ means that Zippel automatically confirmed the property. A × means that Zippel was unable to confirm the property even though the protocol satisfies it. The value next to the mark indicates analysis time.

9. Evaluation

Our evaluation aims to answer three questions:

1) How long does it take to compile protocols, and how large are the Graph IRs (§9.1)?
2) What is the performance and what is the memory usage of protocols in Zippel, and how does that compare with existing baselines (§9.2)?
3) How well do our static analyses work and how expensive are they to run (§9.3)?

Experimental testbed. We run all of our experiments on an AWS `m4.4xlarge` instance, which consists of 16 vCPUs (2.4 GHz) and 64 GB of RAM.

Baselines. To put Zippel's performance and resource cost results in context, we compare protocols that are compiled and run in Zippel with equivalent protocols that were previously implemented in Rust by other authors. Our goal is to show that Zippel does not add considerable overhead, and might sometimes even speed up protocols. We found 14 existing implementations that we could use as baselines: Sumcheck, Schnorr, KZG, PST13, Groth16, Bulletproofs, Hyrax, two variants of Spartan (one written in `arkworks` and one written with custom optimizations), Dory, HyperPlonk, KZH, DeKART, and Pari. We had to update several of these implementations because they were using old versions of `arkworks` and the latest version (0.6), which is the one we use, is considerably faster; it would not have been fair to compare Zippel against these older implementations.

Overall, we think these baselines are representative as they range from small proof systems to complex SNARKs.

9.1. Compilation Performance and Graph Size

Compilation times in Zippel vary widely, from 1.2 ms for Schnorr (which has no size parameter) and 7.5 ms for Groth16 (instance size $n = 2^{18}$) to 22.0 seconds for HyperPlonk ($n = 2^{18}$). This variation depends on whether Zippel can represent a protocol symbolically or must unroll compile-time constructs. For systems like Groth16, nodes in Graph IR are symbolic dot products and pairings, whose representation is constant and independent of the instance size. In contrast, protocols that use list comprehensions or primitive recursion (e.g., Spartan's Sumchecks) force Zippel to unroll these operations so they depend on instance size.

Figure 7 reports the size of the compiled Graph IR (the prover and verifier graphs that Zippel actually executes) for each of the protocols that we compare with existing baselines. Each node in the graph is 96 bytes. Even the largest of these graphs stays modest: HyperPlonk's $\mathcal{P}$ graph has 2,296 nodes, well under a megabyte in total.

Note that compilation only depends on instance size and not on inputs, so in principle it only needs to be done once and can be reused (though our current implementation does not yet support saving Graph IRs to disk). Further, during development and local performance testing one can use small instances, such as Spartan with $n = 2^4$, which compile very quickly. The next section discusses how long it takes Zippel to execute the prover and verifier programs.

Takeaway. Compiling programs in Zippel takes milliseconds to tens of seconds. The one drawback is that Zippel materializes the Graph IR which depends on instance sizes for some operations, so as the instances get larger compilation times increase. Nonetheless, these costs are reasonable and in our experience have not impeded our ability to develop or run protocols in Zippel.

9.2. Runtime Performance and Memory Use

To assess Zippel's runtime performance, we compare each of the 13 selected protocols against the baselines. Except for Schnorr, we generate a random satisfying instance of size 2^{18} for each protocol using each baseline's own test-vector construction: random size-2^{18} vectors and

polynomials for the IPA and PCS evaluations, a random satisfying assignment for a square R1CS or Plonkish instance with 2^{18} constraints for the SNARKs, and a degree-3 virtual polynomial over 18-variable multilinear extensions for Sumcheck. We evaluate the prover with 1–8 threads and the verifier with 1 thread. We use one thread for the verifier since the majority of proof systems have fast verification that would not materially benefit from parallelism, and the baselines often do not parallelize the verifier.

Performance results. The middle columns of Figure 7 give the results. A value larger than 1 means that Zippel is faster than the baseline; below 1 means slower. In general, Zippel is comparable to existing implementations across the board and in some cases (e.g., Sumcheck, PST13) it is faster owing to Zippel using different optimizations than these systems.

There are two extremes where Zippel does worse. The first is Schnorr, which is very fast in absolute terms (baseline and Zippel) so Zippel's scheduling and other runtime overheads can make a difference. The second is Spartan's original Microsoft implementation by Srinath Setty [19]. This version of Spartan uses an elliptic-curve arithmetic library specialized for Curve25519/Ristretto255, which provides highly optimized arithmetic operators that leverage x86-64 SIMD instructions like AVX2. Spartan also implements specialized Sumcheck routines, including dedicated quadratic, cubic, and batched Sumcheck operations. In contrast, Zippel uses the more general-purpose `arkworks` library, which aims to support multiple elliptic curves and finite fields at the cost of giving up some of the benefits of specialization. Indeed, when we compare Zippel with another existing implementation of Spartan [18] based on `arkworks` we see that Zippel performs better.

Memory use. For both Zippel and the baseline, we measure the peak heap memory allocated while generating a proof (excluding keys, witness, and other inputs that both sides already hold), using a single thread to isolate the difference between the two implementations.

The last column of Figure 7 gives the results. As with performance, Zippel is mostly on par with existing implementations, suggesting that Zippel's design is not inherently more memory intensive. In some cases it can even be more efficient. Zippel frees up memory as soon as it can, which is not always a key goal for some of the baselines. As a result, Zippel can be considerably more memory efficient, as in the case of Sumcheck and Hyrax.

Takeaway. Zippel's ease of implementation, security, and correctness benefits do not come at a high cost. Of course, highly tuned and specialized implementations can be more performant than Zippel, in much the same way that hand-written assembly can be faster or more memory efficient than the assembly produced by compilers like `gcc` or `clang`. But for most cases, Zippel's implementations are competitive, and any future improvements to Zippel's runtime directly improve all protocols that run on it.

9.3. Analysis Coverage

We evaluate Zippel's automated static analyses on all the systems in Figure 6 using a 20-minute timeout. Since our analyses perform bounded model checking, that means our result establishes the property only for protocols of a specific instance size, not for all sizes. To maximize the coverage this provides while keeping the analysis tractable, we pick the smallest instance size that exercises every code path for each protocol (e.g., both the base case and the recursive case of a size-overloaded function).

Results. All protocols are perfectly complete, so we evaluate our analysis on all of them. Figure 6 gives the results. Our current implementation of Zippel can automatically verify 33 of them (94%), all in under a minute, and most within tens of milliseconds. The exceptions are HyperPlonk and Spartan, which are simply too large and produce millions of equations and variables even for small instance sizes (each coefficient of each polynomial becomes a variable). This suggests that to support automated analyses for these large complex SNARKs we will need more modular verification.

Only ten protocols are special-sound: Schnorr, Multi-Schnorr, Pedersen-Equality, Chaum-Pedersen, Okamoto, Okamoto-ElGamal, CDS, E-Cash Coin, and the Proof of Product and Proof of Dot Product protocols in Hyrax.[2] Of these protocols, our analysis works on five of them. The failures are illustrative. Okamoto proves knowledge of a Pedersen opening. The issue is that Pedersen commitments are not perfectly binding so the witness is not unique. Meanwhile, our heuristic—finding polynomials whose leading term is the witness (§7.4)—can only extract uniquely determined witnesses. This is also why our soundness analysis fails on Proof of Product, Proof of Dot Product, and E-Cash Coin, all of which use Pedersen commitments.

In contrast, Zippel is able to verify special soundness for Okamoto-ElGamal, which is a variant of Okamoto that is very similar but uses the perfectly binding ElGamal construction, so our analysis is able to extract the witness. Our analysis also extracts the witness for Pedersen-Equality, which proves that two Pedersen commitments commit to the same value. It works because the protocol proves that the difference of the two commitments has a known discrete log with respect to a single base, and that discrete log is unique.

CDS is different. Its extractor must first find which subproofs received different challenge shares in the two transcripts, and extract witnesses only from those. Our analysis searches for extractors that are a single polynomial in the transcripts, so it cannot express this case split.

We plan to make our analyses more robust in the future.

Takeaway. Zippel is able to verify the completeness and special soundness of many protocols. A key benefit is that whenever the analyses work, they finish very quickly: most complete within tens of milliseconds. This is fast enough that both developers and AI coding agents can repeatedly

2. A few other protocols, such as Bulletproofs [43], are special-sound under computational assumptions, which our analysis does not apply to.

Proof System	LoC		Graph IR (nodes)		$\mathcal{P}$ Speedup (threads)				$\mathcal{V}$	Peak Memory Ratio for $\mathcal{P}$
	Zippel	Baseline	$\mathcal{P}$	$\mathcal{V}$	1	2	4	8	Speedup	Baseline / Zippel
Sumcheck [14]	87	1,559	180	182	1.27	1.35	1.36	1.94	1.12	4.64
Schnorr [3]	7	186	11	13	1.25	0.94	0.83	0.80	1.04	0.01
KZG [6]	24	527	21	18	1.00	0.99	0.99	1.00	0.95	0.94
PST13 [13]	81	250	221	91	1.79	1.80	1.81	1.85	0.59	1.16
Groth16 [4]	76	458	46	25	1.00	1.00	1.00	1.00	0.98	1.07
Bulletproofs [7]	77	167	656	421	0.98	0.97	0.98	0.98	0.98	1.00
Hyrax PCS [5]	51	277	1,167	142	0.99	0.99	1.00	1.00	2.06	21.49
Spartan (arkworks) [18]	424	2,037	1,931	1,269	0.89	1.01	1.27	1.48	0.88	0.95
Spartan (Microsoft) [19]	424	1,867	1,931	1,269	0.34	0.37	0.48	0.60	0.46	0.95
Dory PCS [8]	304	766	636	762	1.01	1.06	1.12	1.23	0.97	0.51
HyperPlonk [12]	433	2,828	2,296	1,074	1.02	1.06	1.08	1.12	0.98	1.59
KZH [15]	50	221	68	134	0.98	1.01	0.98	0.99	0.99	1.00
DeKART [1]	124	846	205	115	0.85	0.96	1.12	1.34	0.66	0.88
Pari [11]	109	1,141	71	39	0.93	0.94	0.92	0.90	0.78	0.99

Figure 7: Comparison of Zippel to existing implementations of the same protocols. We give lines of code (LoC) of both Zippel and baseline protocols, as well as the size of the Graph IRs for Zippel's prover ($\mathcal{P}$) and verifier ($\mathcal{V}$). We also show the relative prover performance with 1, 2, 4, and 8 threads, the relative verifier performance with 1 thread, and the relative peak memory use (1 thread). We compute these ratios by dividing the baseline number by Zippel's number: a value over 1 means that Zippel is better. All protocols except Schnorr use instances of size 2^{18}. Results are the mean across 10 runs.

use these analyses to confirm their implementation and iterate on their code in real time. The drawback is that whenever the analyses do not work, it is not obvious if they fail because the protocol is incomplete, unsound, too complex, or one is simply missing a statement in the **where** clause. In the future we hope to build analyses for *incompleteness* and *unsoundness*, which tell the developer that the protocol is wrong and explain why by providing a counterexample.

10. Discussion and Future Work

Zippel makes it easy to implement cryptographic proof systems at a high level of abstraction that eliminates entire classes of common bugs (e.g., polynomial size mismatch, weak Fiat–Shamir) by construction. This assurance comes at little cost: our evaluation (§9) shows that Zippel programs perform similarly to existing implementations.

Zippel expresses the completeness and special soundness of cryptographic proof systems as questions of polynomial ideal containment, enabling fully automated static analyses. Prior works have used Gröbner bases, but for different purposes (functional correctness). For example, Tsai–Wang–Yang [96] use GBs to check whether some assembly code implements the right field or EC operator, whereas Ozdemir et al. [86] use GBs to check whether a Boolean-to-arithmetic circuit compiler is correctly implemented.

Nevertheless, Zippel has limitations. It lacks support for proof systems based on hashing, lattices, and groups of unknown order. Further, our analyses perform only bounded model checking, can time out, and do not presently scale to all the protocols we implemented. These drawbacks are real, but they also present opportunities: Zippel is a strong foundation for a lot of exciting research directions.

Further automatic analyses. Zippel's Graph IR could support automated zero-knowledge analyses that construct a simulator or prove its existence. Additionally, Zippel's sized type system could automatically reason about transcript sizes to statically establish a protocol's succinctness. The analyses could use SMT solvers to produce counterexamples to confirm violations of properties as well.

Beyond automation. When automated analyses fall short, Zippel could also be an effective bridge to interactive theorem provers like Lean [58] and proof libraries like ArkLib [10]. Zippel exposes three levels of abstraction for extracting verification targets:

- *Ideals:* At the mathematical level, we could extract the variety and ideal containment assertions as lemmas to be discharged in Lean.
- *Graph IR:* At the intermediate representation level, we could extract Lean theorem statements about the security of Zippel's graphs, which have a richer type system, but still have fixed-size data.
- Zippel *Syntax:* At the language level, leveraging Zippel's formal operational semantics, we could extract size-generic Lean proof obligations. This would allow developers to prove size-agnostic protocol security.

We see Zippel as a foundation for understanding and implementing cryptographic proof systems going forward. Our experience is that giving these proof systems a language of their own unlocks novel analyses and optimizations. Further, by unifying how proof systems are expressed, executed, and analyzed, Zippel narrows the gap between the protocols we design and the implementations we can trust.

Acknowledgements

We thank the S&P 2027 reviewers for their thorough comments on our work. We also thank Mike Hicks, Stephanie Weirich, and Steve Zdancewic for feedback on the formal semantics and type preservation proofs of Zippel. This work was funded in part by NSF Award CNS-2045861 and gifts from the Ethereum Foundation and Ingonyama.

References

[1] aptos-dkg: DeKART univariate range proof implementation. https://github.com/aptos-labs/aptos-core/blob/main/crates/aptos-dkg/src/range_proofs/dekart_univariate_v2.rs.

[2] arkworks: An ecosystem for developing and programming with zkSNARKs. https://github.com/arkworks-rs.

[3] arkworks crypto-primitives: Schnorr signature implementation. https://github.com/arkworks-rs/crypto-primitives/blob/v0.6.0/crypto-primitives/src/signature/schnorr.

[4] arkworks Groth16 implementation. https://github.com/arkworks-rs/groth16/tree/v0.6.0/src.

[5] arkworks poly-commit: Hyrax implementation. https://github.com/arkworks-rs/poly-commit/tree/v0.6.0/poly-commit/src/hyrax.

[6] arkworks poly-commit: KZG10 implementation. https://github.com/arkworks-rs/poly-commit/blob/v0.6.0/poly-commit/src/kzg10/.

[7] Bulletproofs implementation. https://github.com/alex-ozdemir/bulletproofs/blob/master/src/reductions/bp_2ary_step.rs.

[8] dory-pcs: Dory implementation. https://github.com/a16z/dory/tree/5246da5.

[9] Dusk Network missing blinding factors. https://github.com/0xPARC/zk-bug-tracker#17-dusk-network-missing-blinding-factors.

[10] Formally verified arguments of knowledge in Lean. https://github.com/Verified-zkEVM/ArkLib.

[11] garuda-pari: Pari implementation. https://github.com/alireza-shirzad/garuda-pari/tree/main/pari.

[12] HyperPlonk implementation. https://github.com/EspressoSystems/hyperplonk/tree/2a3b55c.

[13] HyperPlonk: multilinear KZG (PST13) implementation. https://github.com/EspressoSystems/hyperplonk/tree/main/subroutines/src/pcs/multilinear_kzg.

[14] HyperPlonk: sum-check subroutine implementation. https://github.com/EspressoSystems/hyperplonk/blob/main/subroutines/src/poly_iop/sum_check.

[15] irondict: KZH implementation. https://github.com/alireza-shirzad/irondict/tree/bf2369d/subroutines/src/pcs/kzhk.

[16] Noir: The programming language for private apps. https://noir-lang.org/.

[17] Rayon: A data parallelism library for Rust. https://github.com/rayon-rs/rayon.

[18] Spartan. https://github.com/zhenfeizhang/ark-spartan.

[19] Spartan. https://github.com/microsoft/Spartan/tree/3a2c097cab39.

[20] spongefish: Fiat–Shamir for the masses. https://github.com/arkworks-rs/spongefish/.

[21] Zendoo: Missing polynomial normalization after arithmetic operations. https://github.com/0xPARC/zk-bug-tracker#zendoo-polynomial-1.

[22] ZoKrates: A toolbox for zkSNARKs on Ethereum. https://github.com/zokrates/zokrates.

[23] Axiom halo2 libraries. https://github.com/trailofbits/publications/blob/master/reviews/2023-06-axiom-halo2libraries-securityreview.pdf, 2023.

[24] Scroll zkEVM halo2 circuits. https://github.com/trailofbits/publications/blob/master/reviews/2023-04-scroll-zkEVM-wave1-securityreview.pdf, 2023.

[25] CVE-2024-45040: Gnark commitments to private witnesses in Groth16 as implemented break zero-knowledge property. https://advisories.gitlab.com/golang/github.com/consensys/gnark/CVE-2024-45040/, 2024.

[26] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[27] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on Σ-protocols. In *ESORICS*, 2010.

[28] M. Ambrona, G. Barthe, and B. Schmidt. Automated unbounded analysis of cryptographic constructions in the generic group model. In *EUROCRYPT*, 2016.

[29] S. Angel, A. J. Blumberg, E. Ioannidis, and J. Woods. Efficient representation of numerical optimization problems for SNARKs. In *USENIX Security*, 2022.

[30] S. Angel, S. Celi, E. Margolin, P. Mishra, M. Sander, and J. Woods. Coral: Fast Succinct Non-Interactive Zero-Knowledge CFG Proofs. In *S&P*, 2026.

[31] S. Angel, E. Ioannidis, E. Margolin, S. Setty, and J. Woods. Reef: Fast succinct non-interactive zero-knowledge regex proofs. In *USENIX Security*, 2024.

[32] T. Attema and R. Cramer. Compressed Σ-protocol theory and practical application to plug & play secure algorithmics. In *CRYPTO*, 2020.

[33] T. Attema, R. Cramer, and L. Kohl. A compressed Σ-protocol theory for lattices. In *CRYPTO*, 2021.

[34] T. Attema and S. Fehr. Parallel repetition of $(k_1, \ldots, k_\mu)$-special-sound multi-round interactive proofs. In *CRYPTO*, 2022.

[35] J. Bacelar Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Zanella Béguelin. Full proof cryptography: Verifiable compilation of efficient zero-knowledge protocols. In *CCS*, 2012.

[36] E. Bangerter, T. Briner, W. Henecka, S. Krenn, A.-R. Sadeghi, and T. Schneider. Automatic generation of Σ-protocols. In *European Workshop on Public Key Services, Applications and Infrastructures (EUROPKI)*, 2009.

[37] G. Barthe, E. Fagerholm, D. Fiore, J. C. Mitchell, A. Scedrov, and B. Schmidt. Automated analysis of cryptographic assumptions in generic group models. In *CRYPTO*, 2014.

[38] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing*, 20(6), 2022.

[39] D. Bernhard, O. Pereira, and B. Warinschi. How not to prove yourself: Pitfalls of the Fiat–Shamir heuristic and applications to Helios. In *ASIACRYPT*, 2012.

[40] D. Boneh, T. Datta, R. Fernando, K. Nazirkhanova, and A. Tomescu. DekartProof: Efficient vector range proofs and their applications. Cryptology ePrint, 2025/1159, 2025.

[41] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *EUROCRYPT*, 2016.

[42] W. Borgeaud. Membership proofs from polynomial commitments. https://solvable.group/posts/membership-proofs-from-polynomial-commitments/, 2020.

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

[44] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. Spooner. Proof-carrying data without succinct arguments. In *CRYPTO*, 2021.

[45] J. Camenisch and M. Stadler. Proof systems for general statements about discrete logarithms. Technical Report 260, Institute for Theoretical Computer Science, ETH Zurich, 1997.

[46] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO*, 1992.

[47] B. Chen, B. Bünz, D. Boneh, and Z. Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In *EUROCRYPT*, 2023.

[48] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *EUROCRYPT*, 2020.

[49] A. Chiesa and M. Orrù. A Fiat–Shamir transformation from duplex sponges. In *TCC*, 2025.

[50] D. A. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, 4th edition, 2015.

[51] R. Cramer. *Modular design of secure yet practical cryptographic protocols*. PhD thesis, University of Amsterdam, 1997.

[52] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, 1994.

[53] I. Damgård. On Σ-protocols. *Lecture Notes, University of Aarhus, Department for Computer Science*, 84, 2002.

[54] Q. Dao, A. Biswas, L. Eagen, A. Milson, S. Papini, and J. Thaler. The sum-check protocol over the monomial basis, and other optimizations. Cryptology ePrint, 2026/762, 2026.

[55] Q. Dao, Z. DeStefano, S. Bagad, Y. Domb, and J. Thaler. Speeding up sum-check proving. Cryptology ePrint, 2026/587, 2026.

[56] Q. Dao, J. Miller, O. Wright, and P. Grubbs. Weak Fiat–Shamir attacks on modern proof systems. In *USENIX Security*, 2023.
[57] Q. Dao and J. Thaler. More optimizations to sum-check proving. Cryptology ePrint, 2024/1210, 2024.
[58] L. De Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*. Springer, 2015.
[59] M. Dellepere, P. Mishra, and A. Shirzad. Garuda and Pari: Faster and smaller SNARKs via equifficient polynomial commitments. In *USENIX Security*, 2026.
[60] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4), July 1985.
[61] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, 1986.
[62] L. Goldberg, S. Papini, and M. Riabzev. Cairo–a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint, 2021/1063, 2021.
[63] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.
[64] S. Goldwasser and S. Micali. The knowledge complexity of interactive proof-systems. In *STOC*, 1985.
[65] J. Groth. Linear algebra with sub-linear zero-knowledge arguments. In *CRYPTO*, 2009.
[66] J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, 2016.
[67] A. K. Hirsch and D. Garg. Pirouette: Higher-order typed functional choreographies. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–27, 2022.
[68] E. Ioannidis. *Correct Programs, Executed Correctly: Verifying Specifications and Executions*. PhD thesis, University of Pennsylvania, 2025. https://repository.upenn.edu/handle/20.500.14332/61703.
[69] G. Kadianakis, A. Zapico, H. Hafezi, and B. Bünz. KZH-fold: Accountable voting from sublinear accumulation. In *CCS*, 2025.
[70] A. B. Kahn. Topological sorting of large networks. *CACM*, 5(11), 1962.
[71] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, 2010.
[72] D. Khovratovich, R. D. Rothblum, and L. Soukhanov. How to prove false statements: Practical attacks on Fiat–Shamir. In *CRYPTO*, 2025.
[73] T. Kohrita and P. Towa. Zeromorph: Zero-knowledge multilinear-evaluation proofs from homomorphic univariate commitments. *Journal of Cryptology*, 37(38), 2024.
[74] A. Kosba, C. Papamanthou, and E. Shi. xJsnark: A framework for efficient verifiable computation. In *S&P*, 2018.
[75] R. Krishnan, A. Samuelson, E. Yao, and E. Cecchetti. zkStruDul: Programming zkSNARKs with structural duality. *arXiv preprint arXiv:2511.10565*, 2025.
[76] J. Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In *TCC*, 2021.
[77] T. Liu, T. Xie, J. Zhang, D. Song, and Y. Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. In *S&P*, 2024.
[78] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. In *FOCS*, Oct. 1990.
[79] E. McMurtry, O. Pereira, and V. Teague. When is a test not a proof? In *ESORICS*, 2020.
[80] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash. In *USENIX Security*, 2010.
[81] J. Miller. Coordinated disclosure of vulnerabilities affecting Girault, Bulletproofs, and PlonK. https://blog.trailofbits.com/2022/04/13/part-1-coordinated-disclosure-of-vulnerabilities-affecting-girault-bulletproofs-and-plonk/, 2022.
[82] F. Montesi. *Choreographic Programming*. PhD thesis, IT University of Copenhagen, Denmark, 2014. https://pure.itu.dk/en/publications/choreographic-programming/.
[83] H. Nguyen, U. Ho, and A. Biryukov. Fiat–Shamir in the wild. In *Proceedings of the 8th International Symposium on Cyber Security, Cryptology, and Machine Learning (CSCML)*, pages 135–150, 2024.
[84] T. Okamoto. Provably secure and practical identification schemes and corresponding signature schemes. In *CRYPTO*, 1992.
[85] A. Ozdemir, F. Brown, and R. S. Wahby. CirC: Compiler infrastructure for proof systems, software verification, and more. In *S&P*, 2022.
[86] A. Ozdemir, G. Kremer, C. Tinelli, and C. Barrett. Satisfiability modulo finite fields. In *CAV*, 2023.
[87] C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In *TCC*, 2013.
[88] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*. Springer, 1991.
[89] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4, 1991.
[90] S. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO*, 2020.
[91] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. In *OSDI*, 2018.
[92] S. Setty, S. Angel, and J. Lee. Verifiable state machines: Proofs that untrusted services operate correctly. *ACM SIGOPS Operating Systems Review*, 54(1), 2020.
[93] I. Sweet, D. Darais, D. Heath, W. Harris, R. Estes, and M. Hicks. Symphony: Expressive secure multiparty computation with coordination. *arXiv preprint arXiv:2302.10076*, 2023.
[94] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, 2013.
[95] J. Thaler. Proofs, arguments, and zero-knowledge, 2020. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf.
[96] M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In *CCS*, 2017.
[97] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *S&P*, 2013.
[98] R. S. Wahby, I. Tzialla, abhi shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *S&P*, 2018.
[99] B. WhiteHat. roll_up. https://github.com/barryWhiteHat/roll_up, 2018.
[100] O. Wright. Specialized zero-knowledge proof failures. https://blog.trailofbits.com/2022/11/29/specialized-zero-knowledge-proof-failures, 2022.

Appendix A. Rust harness for Zippel programs

This section describes how a concrete proof system is instantiated, executed, and analyzed in Zippel. Recall that a developer writes a Zippel program (e.g., Figure 1b), but this program contains an abstract choreography. Executing this choreography in practice requires binding abstract groups, fields, and variable values to concrete Rust types and runtime objects. To do so, developers write a wrapper program in Rust that interacts with the Zippel runtime API. Figure 8 illustrates a simplified version of such a wrapper for the Schnorr protocol, defining the operations for both $\mathcal{P}$ and $\mathcal{V}$.

A.1. Prover Orchestration

As shown in the `zippel_prover()` function of Figure 8, executing $\mathcal{P}$'s side consists of five distinct steps:

Parsing: $\mathcal{P}$ parses the protocol given in the file path (e.g., `"schnorr.zippel"`).

Compilation: Calling `compile()` creates the Graph IR, and lowers size-parameterized structures. It also concretizes the abstract group and fields. If the protocol includes container sizes or recursion bounds that are generic in the choreography, they become type-level size variables (e.g.,

```
fn zippel_prover() {
  let proto = Parse::load("schnorr.zippel");
  let graph = proto.compile::<Bls12_381>();
  let prover = graph.project_prover();
  let inputs = inputs();
  let proof = prover.prove(inputs);
  // send proof to verifier
}

// Example prover input generation
fn inputs() -> Ctx<Vid, Value<Bls12_381>> {
  let mut rng = rand::rngs::OsRng;
  let x = <Bls12_381>::F::rand(&mut rng);
  let g = <Bls12_381>::G1::rand(&mut rng);
  let h = G1Ops::vec_mul(&g, &vec![x]);
  // assign values to Zippel variables
  Ctx::<Vid, Value<Bls12_381>>::from_iter([
    (Vid("x".to_string()), Value::Scalar(x)),
    (Vid("g".to_string()), Value::G1(g)),
    (Vid("h".to_string()), Value::G1(h)),
  ])
}

fn zippel_verifier() {
  let proto = Parse::load("schnorr.zippel");
  let graph = proto.compile::<Bls12_381>();
  let verifier = graph.project_verifier();
  let v_in = get_verifier_inputs();
  let result = verifier.verify(v_in);
}
```

Figure 8: Example Zippel wrappers in Rust for $\mathcal{P}$ and $\mathcal{V}$. The prover and verifier functions are meant to be written in different files as part of larger applications and the proof would be communicated over the network.

`N: Size` in the Sumcheck protocol of Figure 10, representing the number of variables of the multilinear polynomial). To resolve them, the developer provides a mapping from these size variable identifiers (e.g., a size identifier `Tid("N")`) to a concrete `usize` value (such as $N = 10$) using a size context map of type `Ctx<Tid, usize>` passed to the compiler. During this step, the handler performs size monomorphization (§4.2), generating a fully inlined, static graph specialized for these bounds.

Projection: Calling `project_prover()` projects the graph into the corresponding prover executable graph.

Input preparation: $\mathcal{P}$'s inputs (including **instance**, **extra**, and **witness**) are mapped to their respective variables using the `Ctx` environment map, which maps variable identifiers (`Vid`) to algebraic wrappers (`Value`). In the `inputs()` helper function, the witness x and generator g are sampled, and the public commitment h is computed. These are then wrapped into their corresponding algebraic types (e.g., `Value::Scalar(x)`, `Value::G1(g)`) and associated with their Zippel variable names. Of course, in an actual application these variables would not be generated at random but rather would come from somewhere else. It is nevertheless the same idea: add them to the `Ctx` map.

Execution: Invoking `prove()` executes the prover's projected subgraph to generate a serialized `proof` transcript. Under the hood, the Zippel runtime automatically parallelizes the computation across available CPU cores.

A.2. Verifier Orchestration

On $\mathcal{V}$'s side, shown in the `zippel_verifier()` function of Figure 8, execution proceeds similarly to $\mathcal{P}$. $\mathcal{V}$ loads and compiles the protocol with the concrete groups and fields, and projects the graph. However, $\mathcal{V}$ has no access to **witness** or **extra** variables (e.g., the discrete log x). Instead, it supplies the public **instance** inputs (e.g., the generator g and commitment h) and proof transcript which contains the prover messages. As we discuss in Section 6.2, the verifier computes **challenge** values from scratch.

Appendix B. Full Syntax

Figure 9 presents the full syntax grammar for Zippel. A Zippel program consists of a sequence of global declarations. These declarations include protocol definitions (**proto**) that capture choreographies, helper functions (**fn**), and algebraic type definitions (`type`). The language features a rich system of qualifiers (**instance**, **witness**, **extra**), algebraic kinds (**Field**, **Group**, **Pairing**, **Scalar**), sized types (such as vectors, univariate, multivariate, and multilinear polynomials), and domain-specific expressions tailored to expressing cryptographic proof systems.

Appendix C. Example Recursion in Zippel

Figure 10 shows an implementation of Sumcheck in Zippel. The recursive helper function `sum` is defined with two size-overloaded variants: a base case specialized for two active variables (`p`: **Poly**$\langle F, 2, 1\rangle$) and an inductive step overloaded on the number of active variables `A: 3..N+1` (`p`: **Poly**$\langle F, A, 1\rangle$). To make the recursive progression easy to follow, we highlight the changing active variable dimensions **A** and **2** in bold in the figure. Crucially, both variants are parameterized by the total size parameter `N: Size` representing the full variable dimension of the original polynomial `orig`: **Poly**$\langle F, N, 1\rangle$. At the end of the inductive step, the recursive call `sum(orig, residual, chals2, ...)` passes arguments with changing size dimensions: the active polynomial `residual` has one fewer variable (**Poly**$\langle F, A-1, 1\rangle$), while the accumulated challenge vector `chals2` grows by one element ($[F; N-(A-1)+1]$). The compiler typechecks this call by matching it against the signature of `sum` instantiated with the decremented active size parameter `A - 1`. When this active size decreases to 2, the call automatically routes to the overloaded base-case variant.

⟨x, T⟩ ::= ⟨identifier⟩
⟨module⟩ ::= ⟨decl⟩*
⟨decl⟩ ::= **proto** ⟨x⟩ ⟨ ⟨tvars⟩? ⟩ (⟨args⟩?) **where** ⟨exp⟩ { ⟨exp⟩? }
| **fn** ⟨x⟩ ⟨ ⟨tvars⟩? ⟩ (⟨args⟩?) (→ ⟨typ⟩)? { ⟨exp⟩? }
| `type` ⟨x⟩ = ⟨typ⟩ ;
⟨tvar⟩ ::= ⟨T⟩ : ⟨kind⟩ | ⟨T⟩ = ⟨sz⟩
⟨tvars⟩ ::= ⟨tvar⟩ (, ⟨tvar⟩)*
⟨arg⟩ ::= ⟨qualifier⟩? ⟨x⟩ : ⟨typ⟩
⟨args⟩ ::= ⟨arg⟩ (, ⟨arg⟩)*
⟨qualifier⟩ ::= **instance** | **witness** | **extra**
⟨kind⟩ ::= **Field** | **Group** | **Size**
| **Pairing** ⟨ ⟨T⟩ , ⟨T⟩ ⟩ | **Scalar** ⟨ ⟨T⟩ (, ⟨T⟩)* ⟩ | ⟨range⟩
⟨typ⟩ ::= **Unit** | **Bool** | ⟨T⟩ | [⟨typ⟩ ; ⟨sz⟩] | { ⟨recordfields⟩ }
| **Fin** ⟨ ⟨range⟩ | ⟨sz⟩ ⟩ | **Uni** ⟨ ⟨T⟩ , ⟨sz⟩ ⟩
| **Mle** ⟨ ⟨T⟩ , ⟨sz⟩ ⟩ | **Poly** ⟨ ⟨T⟩ , ⟨sz⟩ , ⟨sz⟩ ⟩
⟨recordfields⟩ ::= ⟨recordfield⟩ (, ⟨recordfield⟩)*
⟨recordfield⟩ ::= ⟨x⟩ : ⟨typ⟩
⟨sz⟩ ::= ℕ | ⟨x⟩ | (⟨sz⟩) | ⟨sz⟩ ⟨szop⟩ ⟨sz⟩
⟨szop⟩ ::= + | − | ∗ | / | ^
⟨range⟩ ::= ⟨sz⟩ .. ⟨sz⟩ | ⟨sz⟩ , ⟨sz⟩ .. ⟨sz⟩
⟨exp⟩ ::= **let** ⟨x⟩ (: ⟨typ⟩)? = ⟨exp⟩ ; ⟨exp⟩?
| ⟨x⟩ ← ⟨exp⟩ ; ⟨exp⟩? | ⟨exp⟩ ; ⟨exp⟩?
| ℕ | ⟨x⟩ | () | (⟨exp⟩) | ⟨range⟩
| − ⟨exp⟩ | ⟨exp⟩ ⟨op⟩ ⟨exp⟩
| **pair** (⟨exp⟩ , ⟨exp⟩) | [⟨exps⟩] | ⟨x⟩ [⟨exp⟩]
| **poly** (⟨exp⟩) | **coef** (⟨exp⟩) | **mle** (⟨exp⟩)
| **eval** (⟨exp⟩ (, ⟨exp⟩)?) | **dot** (⟨exp⟩ , ⟨exp⟩)
| **eval** ⟨ ⟨range⟩ | ⟨sz⟩ ⟩ (⟨exp⟩ , ⟨exp⟩)
| **interpolate** (⟨exp⟩ (, ⟨exp⟩)?)
| **random** ⟨ ⟨T⟩ ⟨excl⟩? ⟩
| **challenge** ⟨ ⟨T⟩ ⟨excl⟩? ⟩
| [⟨exp⟩ **for** ⟨x⟩ **in** ⟨exp⟩] | **reduce** (⟨op⟩ , ⟨exp⟩)
| **fun** (⟨x⟩ (, ⟨x⟩)*) => ⟨exp⟩ | ⟨x⟩ (⟨exps⟩?)
| **verify** (⟨exp⟩)
| {| ⟨recordexps⟩ |} | ⟨exp⟩ . ⟨x⟩
| ⟨exp⟩ . `set` (⟨x⟩ , ⟨exp⟩)
⟨excl⟩ ::= − ⟨exp⟩ | − **roots_of_unity** (⟨sz⟩)
⟨exps⟩ ::= ⟨exp⟩ (, ⟨exp⟩)*
⟨recordexps⟩ ::= ⟨recordexp⟩ (, ⟨recordexp⟩)*
⟨recordexp⟩ ::= ⟨x⟩ : ⟨exp⟩
⟨op⟩ ::= + | − | ∗ | / | % | ++ | ^ | == | &&

Figure 9: Zippel language syntax, specifying declarations, qualifiers, kinds, types, sizes, ranges, and expressions.

Appendix D. Generalized Special Soundness

In this appendix, we generalize our special soundness analysis to multi-variable challenges (§D.1) and multiple rounds (§D.2).

D.1. Multi-variable challenges

Our special soundness analysis also generalizes to challenges that are not single variables, but rather vectors of variables. The analysis is the same as in Section 7.4 and Appendix D.2, except that the set D must be modified to express vector distinctness. Consider the case of ℓ-special-soundness. Let $\mathbf{c}_i$ and $\mathbf{c}_j$ be the vector challenges for two accepting transcripts. To express the requirement that $\mathbf{c}_i$ and $\mathbf{c}_j$ are distinct, we can use the following polynomial:

$$\prod_{k=1}^{|\mathbf{c}_i|}(d_{i,j,k} \cdot (c_{i,k} - c_{j,k}) - 1) \tag{10}$$

where the $d_{i,j,k}$ are auxiliary variables representing the multiplicative inverse of the element-wise differences. Note that this polynomial has roots only when $\mathbf{c}_i$ and $\mathbf{c}_j$ differ in at least one element, as desired.

D.2. $(\ell_1, \ldots, \ell_\mu)$-Special Soundness

Our special soundness analysis in Section 7.4 applies to 3-move Σ-protocols. However, we can naturally generalize it to $(2\mu + 1)$-move protocols (which have $\mu + 1$ prover messages and μ challenges).

The property for these many-move protocols is called $(\ell_1, \ldots, \ell_\mu)$-special-soundness [32]. It requires an extractor that computes a witness from $\prod_i \ell_i$ accepting transcripts of a special structure called a *tree*. The structure is that all transcripts must share the first $\mathcal{P}$ message, and then must be partitioned into ℓ_1 equally sized groups based on the value of their first $\mathcal{V}$ challenge. Then, each group must share its second $\mathcal{P}$ message and be partitioned into ℓ_2 equally-sized subgroups based on the second $\mathcal{V}$ challenge. And so on, through ℓ_μ.

Our analysis applies to this protocol by generalizing the set D to express the tree structure of message and challenge (dis)equality. We index transcripts by a vector $\mathbf{i} \in [\ell_1] \times \cdots \times [\ell_\mu]$, whose μ components correspond to the μ protocol rounds. For each $k \in [\mu]$, the verifier issues challenge $c_{\mathbf{i},k}$, after which the prover responds with message $\mathbf{m}_{\mathbf{i},k}$. The initial prover message, which precedes all indexed rounds, is shared across all transcripts and is not part of this indexing. All variables and polynomial sets that were previously subscripted by a scalar index i are now subscripted by our vector index $\mathbf{i}$. Our analysis works exactly as for ℓ-special-soundness, but the transcript structure set D is defined differently. Now it expresses both the equalities and disequalities that define the transcript tree. It contains these polynomials:

- For all $k \in [\mu]$, for all $\mathbf{i}, \mathbf{i}'$ which share their first k subindices, include $\mathbf{m}_{\mathbf{i},k} - \mathbf{m}_{\mathbf{i}',k}$ (message equality).
- For all $k \in [\mu]$, for all $\mathbf{i}, \mathbf{i}'$ which share their first $k-1$ subindices but not their k^{th} subindex, include $d_{\mathbf{i},\mathbf{i}',k}(c_{\mathbf{i},k} - c_{\mathbf{i}',k}) - 1$, for fresh $d_{\mathbf{i},\mathbf{i}',k}$ (challenge disequality).
- For all $k \in [\mu]$, for all $\mathbf{i}, \mathbf{i}'$ which share their first k subindices, include $c_{\mathbf{i},k} - c_{\mathbf{i}',k}$ (challenge equality).

With this D, we can apply our same extractor computation and validation routines to multi-round special-soundness. As in our completeness analysis, our implementation optimizes the presentation above by sharing variables and polynomials among the $V_{\mathbf{i}}$. We can share the variables between different verification conditions for graph nodes that do not depend on challenges that differ between those conditions.

```
fn round_poly<N: Size, F: Field, K: 2..N + 1>(
  p: Poly<F, K, 1>
) -> Uni<F, 1> {
  reduce(+, [
    eval<0>(p, tail)
    for tail in [
      [i / 2 ^ j % 2 for j in 0..K - 1]
      for i in 0..2 ^ (K - 1)
    ]
  ])
}
// Recursive case
fn sum<N: Size, F: Field, A: 3..N + 1>(
  orig: Poly<F, N, 1>,
  p: Poly<F, A, 1>,
  chals: [F; N - A + 1],
  claim: F,
  ch: F
) -> Bool {
  let residual = eval(p, [ch]);
  let rp = round_poly(residual);
  evs <- [rp(0), rp(1)];
  verify(claim == evs[0] + evs[1]);
  let g = interpolate([0, 1], evs);
  r <- challenge<F>;
  let chals2 = chals ++ [r];
  sum(orig, residual, chals2, g(r), r)
}
// Base case
fn sum<N: Size, F: Field>(
  orig: Poly<F, N, 1>,
  p: Poly<F, 2, 1>,
  chals: [F; N - 2 + 1],
  claim: F,
  ch: F
) -> Bool {
  let residual = eval(p, [ch]);
  evs <- [residual(0), residual(1)];
  verify(claim == evs[0] + evs[1]);
  let g = interpolate([0, 1], evs);
  r <- challenge<F>;
  let full_chals = chals ++ [r];
  eval(orig, full_chals) == g(r)
}
proto sumcheck<N: Size, F: Field>(
  instance claim: F,
  instance orig: Poly<F, N, 1>
) where
  let rp = round_poly(orig);
  claim == rp(0) + rp(1)
{
  let rp = round_poly(orig);
  evs <- [rp(0), rp(1)];
  verify(claim == evs[0] + evs[1]);
  r <- challenge<F>;
  let g = interpolate([0, 1], evs);
  verify(sum(orig, orig, [r], g(r), r))
}
```

Figure 10: Implementation of the Sumcheck protocol [78] in Zippel, which proves that a multilinear polynomial `orig` on `N` variables summed over the Boolean hypercube produces the claimed value `claim`.