FlashProve: an open-source library for provable inference

Roy Rinberg (Anthropic), **Alex Allain** (Anthropic), **Alessandro Chiesa** (EPFL, StarkWare), **Miranda Christ** (MIT, RESI), **Rahul Dalal** (Anthropic STEM Fellows Program, Succinct), **Giacomo Fenzi** (EPFL), **Ben Mayer** (Anthropic Fellows Program, MIT), **Guy Weissenberg** (EPFL)*

November 2026

Contents

1. Zero knowledge and succinctness 2
2. The power of cryptographic hash functions 4
3. Anatomy of a zero-knowledge succinct argument 4
4. Formalization in Lean 6
5. Technical overview: a look at the protocol 8
  5.1 Toolkit for IOPs 8
  5.2 Toolkit for zkML 10
  5.3 A simplified IOP for inference 12
  5.4 From IOP to FlashProve 15
6. Implementation: running a 400B model on one GPU 16
7. Limitations 17
Appendix: post-hoc verification & non-determinism 18
Bibliography 20

This report provides a technical description of FlashProve, a system for provable inference of large Mixture-of-Experts (MoE) language models. For discussion of the context and motivations for the development of the system, please see our companion blog post.

When a client sends a prompt to a model provider, the client sees only the generated tokens and has no guarantee that the provider uses the claimed model. The provider could instead use a smaller/weaker model, or even switch models between requests. Publishing the model architecture and weights would, in principle, enable verification. However, most providers keep this information secret, and even with access to it, most clients would lack the resources to run the model themselves and verify its output.

FlashProve addresses this problem by enabling provable inference for a Mixture-of-Experts model (specifically, Llama 4 Maverick). In an offline phase, the provider commits to the model's weights using a short hiding commitment that even "weak" clients can store. For each client request, the provider proves that it correctly evaluated the committed model on the client's input. The core of FlashProve is a cryptographic proof that is *zero-knowledge* (revealing no information about the model's weights) and *succinct* (allowing clients to verify the result much faster than running inference themselves).

*Author order: Anthropic authors first, then alphabetical order.

In a companion blog post, we discuss the broader motivation for this system. Here, we focus on the technical foundations of FlashProve. The system realizes a cryptographic protocol known as a **zero-knowledge succinct argument**, which builds on two active research areas lying at the intersection of coding theory, computational complexity, and cryptography: "zkSNARKs" and "zkML". Our protocol employs battle-tested cryptography that is widely deployed in peer-to-peer systems (e.g., L2 networks such as Starknet or Mantle for blockchains like Ethereum) and authentication systems (e.g., anonymous credentials in Google Wallet). In particular, the protocol relies on the security of hash functions and offers post-quantum security (plausible security even against future quantum attackers). Our protocol, its security properties, and corresponding security analyses are specified in Lean, a proof assistant that enables formal mathematical verification.

FlashProve is intended as a demonstration of what can be achieved today in provable inference; we chose not to incorporate in this demonstration all optimizations that are possible with current techniques, prioritizing an early release. We believe that there is compelling work to be done in this direction, and we encourage the community to pay (even more!) attention to this exciting crossover between cryptographic proofs and AI safety.

We describe FlashProve and its security guarantees by answering the following questions.

1. *Which security properties does the protocol provide?*
2. *Which cryptographic assumptions underlie these properties?*
3. *What does the protocol look like?*
4. *What formal coverage do we provide in Lean?*

After that we give:

5. a technical overview of the main ideas underlying the protocol,
6. a taste of the engineering work that enables the reported efficiency, and
7. the main limitations of our system.

1. Zero knowledge and succinctness

A cryptographic proof system consists of two algorithms: a **prover** and a **verifier**. The prover seeks to convince the verifier that a statement is true. In our setting, the statement takes the form: *"The (committed) model, given the user's input token(s), produces the claimed output tokens."*

The proof system must satisfy two fundamental properties:

- **Completeness.** If the prover follows the protocol and the statement is true, the verifier accepts. In our setting, the verifier accepts a proof of correct inference using the prescribed model weights.
- **Soundness**. Any malicious prover can convince the verifier of a false statement only with negligible probability. In our setting, the verifier rejects attempts to claim an incorrect output token at any checked position or use weights that differ from the prescribed ones.

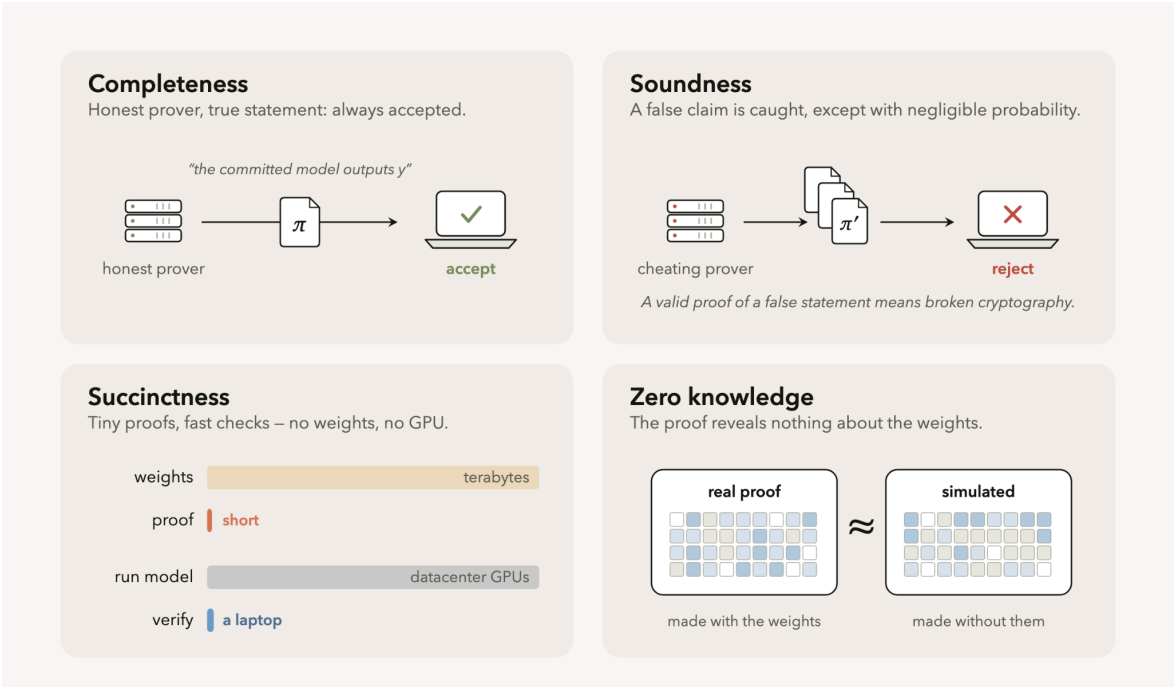
Our proof system provides **computational soundness**: false statements may admit convincing proofs, but finding them is computationally infeasible (it would require "breaking" the underlying

cryptography). This relaxation is necessary to achieve succinctness.[1] Cryptographers refer to proof systems with such computational security guarantees as *argument systems*.

FlashProve must account for committed model weights, which determine the computation being verified. Hence we use a **commit-and-prove** proof system that separates the prover's work into two phases. In the **offline phase**, the prover commits to the model weights and publishes a short digest. In the **online phase**, the prover proves the correctness of each inference relative to those committed weights.

The proof system must additionally satisfy the key properties we highlighted above.

- **Succinctness**. Proofs are much shorter than the computations they certify, and verification is much cheaper than executing those computations. This is crucial in our setting. Model weights occupy terabytes of storage, and inference requires powerful hardware such as an H100 GPU. Yet the verifier must run on consumer hardware without storing model weights or rerunning inference.
- **Zero knowledge**. The proof reveals nothing beyond the validity of the claimed inference. In particular, it hides both the model weights and the intermediate steps of the inference computation, even across repeated requests using the same committed weights.

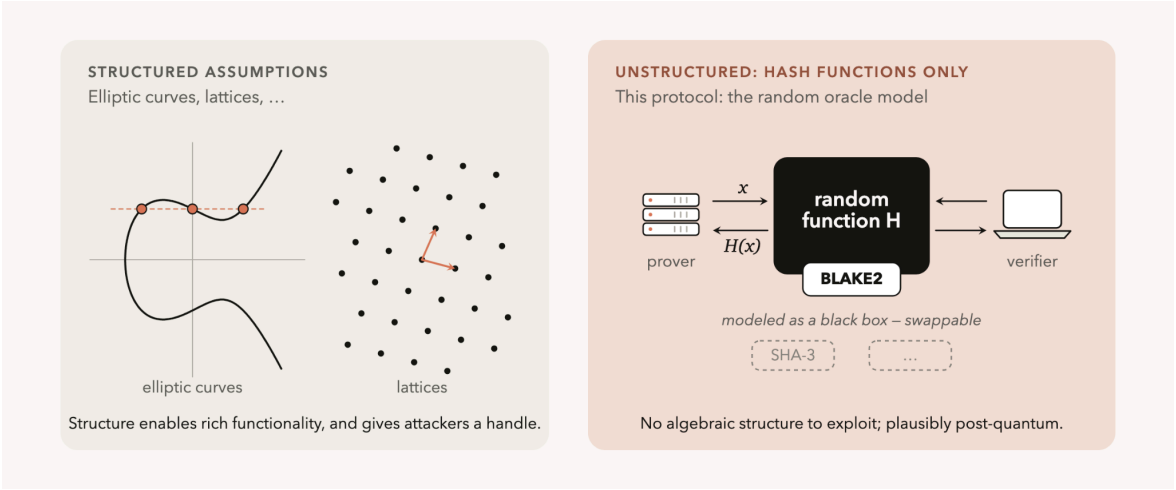


[1]The proof system provides an even stronger property: computational knowledge soundness. (This is common for proof systems of this type.) Informally, this means that any malicious prover that convinces the verifier with more than negligible probability must "know" the weights and a correct trace of computation that explain the claimed output token (given the user's input tokens).

2. The power of cryptographic hash functions

Much of cryptography relies on **structured hardness assumptions** involving algebraic objects such as elliptic curves and lattices. While these structures enable rich functionality (e.g., key agreement and encryption), future attackers might exploit such structures to solve the underlying problems more efficiently. In contrast, some cryptography can be based exclusively on **unstructured assumptions**, such as *hash functions*. Amazingly, this includes zero-knowledge succinct arguments! In particular, FlashProve relies on the assumption that the cryptographic hash function BLAKE2 (more precisely, the compression function inside BLAKE2) behaves like a random function.

More formally, we prove soundness and zero-knowledge in the random oracle model (ROM) [BR93], where the prover and verifier share access to a uniformly random function (modeling BLAKE2). Although the ROM is an idealized model, it provides a compelling basis for long-term security and precise quantitative security guarantees.[2] Our protocol can be instantiated with any hash function that is believed to behave like a random oracle, allowing us to replace BLAKE2 if it is weakened or a more efficient alternative arises.



3. Anatomy of a zero-knowledge succinct argument

Hash functions can be used to construct succinct arguments via an elegant recipe [Kil92, Mic00, BCS16]; these are known as *hash-based succinct arguments* (sometimes also as *STARKs* [BBHR19]).

[2]The random oracle is an idealized model: its security proofs do not give formal guarantees of security of the protocol when the random function is replaced with a concrete hash function (such as BLAKE2). The assumption that a concrete cryptographic hash function behaves "sufficiently like a random function" is called the random oracle heuristic. The random oracle heuristic underlies almost all modern applied cryptography protocols. This heuristic has counterexamples: protocols that can be proven secure in the ROM but are insecure for any concrete hash function. These rely on diagonalization arguments that do not seem to apply in our setting [CGH04, GK03, BBHMR19, KRS25, Fen26].

- **Step 1:** Construct a *probabilistic proof* for the desired statement.
- **Step 2:** Use *Merkle commitments* to compress the probabilistic proof into an interactive protocol.[3]
- **Step 3:** Use the *Fiat–Shamir transformation* to collapse the interaction into a single message.

We elaborate on each step. The discussion below focuses on succinctness; additionally achieving zero-knowledge requires adaptations in each step, and is comparatively less hard. Interested readers can learn more about this recipe in this video by PurpleMind, and in the [CY24] textbook.

Step 1: The starting point is an information-theoretic notion known as an **Interactive Oracle Proof** (IOP) [BCS16, RRR16]. This is an interactive protocol between a prover and a verifier in which the prover sends long message strings and the verifier accesses these long messages via so-called point queries (a query (i, j) retrieves the j-th symbol of the i-th prover message). A beautiful line of work starting in the 1990s, with notions such as Interactive Proofs [BM88, GMR89] and Probabilistically Checkable Proofs [BFLS91, FGLSS96, AS98, ALMSS98], tells us that every computation admits an IOP in which the prover is relatively efficient (compared to the proved computation) and the verifier is super fast (it makes very few queries and performs only cheap computations). Informally, the prover encodes the proved computation in a redundant way, and the verifier asks random questions that involve few queries to the prover's long messages; amazingly, even if the prover lies about the computation's result, the verifier will catch the lie with high probability with few queries! In a way, IOPs are to computation what error-correcting codes are to information.

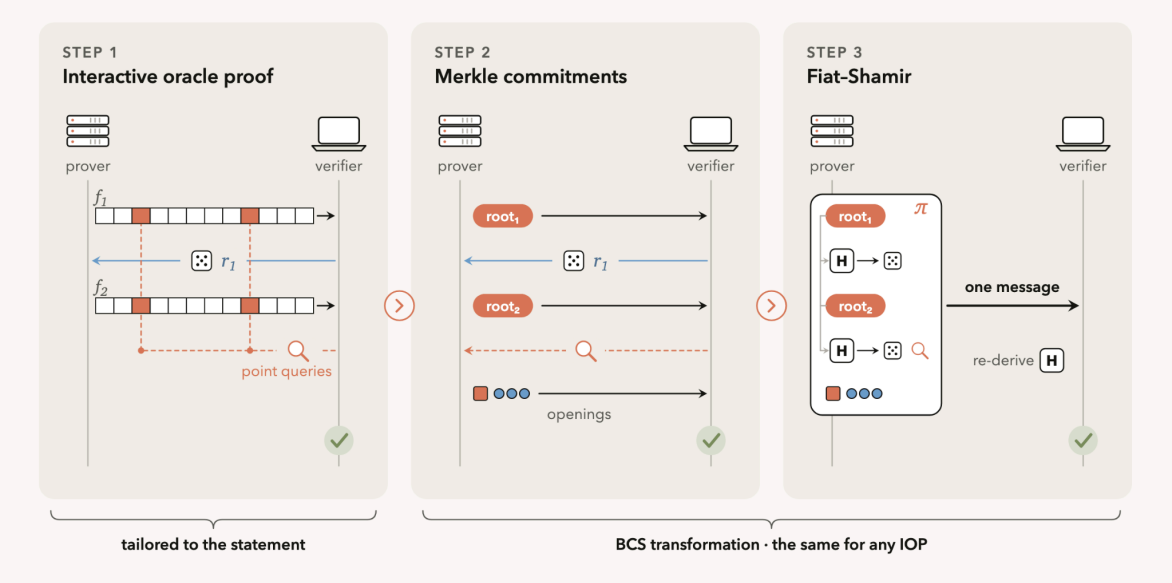
Step 2: Using an IOP in practice requires somehow "implementing" the query interface for the verifier. Simply sending the long prover messages to the verifier eliminates any succinctness gained by querying them; on the other hand, the verifier cannot trust the prover to truthfully answer queries. This tension can be resolved via a commit-then-open construction that relies on cryptography. Specifically, *the IOP is compiled into an interactive protocol where the prover sends only short messages.* This relies on hash trees, also known as **Merkle commitment schemes**. Informally, a hash tree enables anyone to compute a short commitment to a long vector, and subsequently reveal the values of any desired vector entries by supplying a corresponding authentication path to check against the prior commitment. Hash trees directly lead to a simple idea: the prover sends short commitments to oracles instead of the oracles themselves; later on, when the verifier wishes to query an oracle, the prover can reveal the queried locations along with opening proofs certifying the claimed values.

Step 3: The interactive protocol is compiled into a non-interactive protocol (i.e., a single prover-to-verifier message) by applying the **Fiat–Shamir transformation** [FS86, BCS16]. The starting point is that in the IOP from **Step 1**, and therefore also in the corresponding "compressed" interactive protocol from **Step 2**, the verifier's messages to the prover are all simple: uniform random challenges. We saw before that a (deterministic) hash function like BLAKE2 can be idealized as a truly random function, and we can use this intuition to have the prover emulate the interaction with the verifier on its own. Informally, the prover derives each verifier challenge by applying the hash function to the transcript of interaction so far. The prover can then send its (compressed)

[3] Specifically, an interactive argument, that is, an interactive proof system in which the soundness property is computational (holds against all efficient provers, in this setting assuming the hash function used to construct the hash tree is secure).

messages as a single message to the verifier. The verifier can rederive the challenges and check that the resulting transcript of interaction is an accepting one.

The step that requires careful adaptation to our setting (provable inference) is designing a suitable IOP (**Step 1**). The other two steps (**Step 2** and **Step 3**, known as the *BCS transformation* [BCS16, CY24]), while deserving careful optimization and engineering, are "orthogonal" to the specific choice of statement proved/verified. We discuss the IOP and other ideas in the technical overview further below.



4. Formalization in Lean

All mathematical analyses of our protocol are formalized in Lean, a "proof-assistant" programming language used to express formal statements of theorems and their demonstrations. This significantly increases the trust one can place in the protocol: *one must only ensure that the statement of the main theorem about the protocol is correct*, without any need to inspect the mathematical derivations that establish the theorem. (This is provided the Lean checker is correct, of course.)

We specify several items in Lean.

- A formal relation $\mathcal{R}$ that captures all correct inference computations according to the given model.
- The abstract properties of the cryptographic proof system that we construct: the completeness property; the soundness property; and the zero-knowledge property.[4] These properties, as explained in Section 2 above, are in an "idealized" world where the hash function is modeled as a truly random function, i.e., the random oracle model (ROM).
- The protocol, i.e., the prover and verifier functions.

The main theorem informally states "*The protocol is an argument system in the ROM for the relation $\mathcal{R}$ that satisfies the following error bounds: (i) the completeness error is at most ε_1; (ii) the soundness*

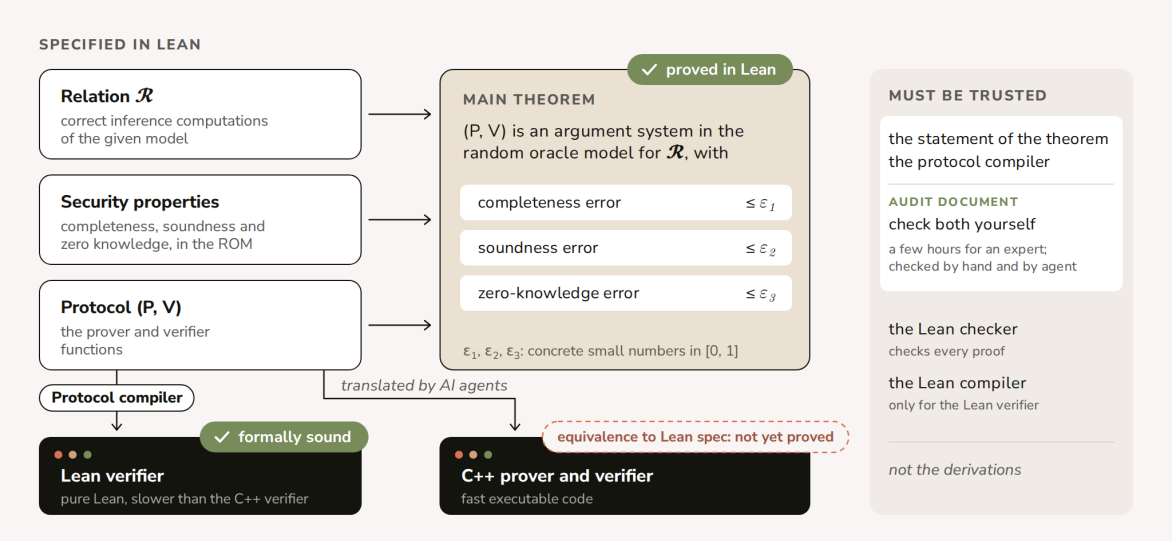
[4]The succinctness property can just be measured. There is no need to formalize it in Lean.

error is at most ε_2; and (iii) the zero-knowledge error is at most ε_3." Here $\varepsilon_1, \varepsilon_2, \varepsilon_3$ are concrete (small) numbers in $[0, 1]$ that quantify the security properties of the protocol implemented in FlashProve.

The executable code of the C++ prover and verifier are obtained by AI agents from their Lean representations. In addition, we provide a verifier that is purely written in Lean compiled from the description of the protocol; while it is slower than the C++ verifier, the executable Lean verifier is formally guaranteed to be sound up to additionally trusting the Lean compiler (used to execute Lean code). At this time, formal equivalence of the C++ code to the Lean specification is not proved.

The full formal security statements and protocol compiler are presented in an interactive viewer (download lean/STATEMENTS.html from the repository and open it in a browser). They have been checked for correctness both by hand and by an AI agent. We expect that it should not take an expert more than a few hours to independently audit them. Finally, for full trust, we point out the following standard guidelines for properly validating lean proofs.

There are broad efforts in the succinct-argument community to specify in Lean the entire "stack" of techniques in modern zkSNARKs. These include ArkLib (formally verified IOP-based zkSNARKs), VCVio [TH24, TDWHH26], CompPoly (computable polynomials and finite fields), part of the Ethereum Foundation's Verified zkEVM initiative and its Rust-to-Lean pipeline [KCT26]. The Proximity Prize [ABF26] motivated the autoresearch challenge better.codes (for a Reed–Solomon proximity problem), which spurred others such as zk.golf (for verified circuits) and snark.fast (for post-quantum proving speed). Earlier and parallel mechanizations include [BM24] (pairing-based SNARKs in Lean 3), [AGLST22, AGGLNST26] (Cairo and the S-two AIR) and Isabelle/STARK [Mar26]. These efforts aim to ultimately also include executable code, to enable end-to-end high-assurance realizations of these systems. Our Lean work builds on these previous ideas and infrastructure.



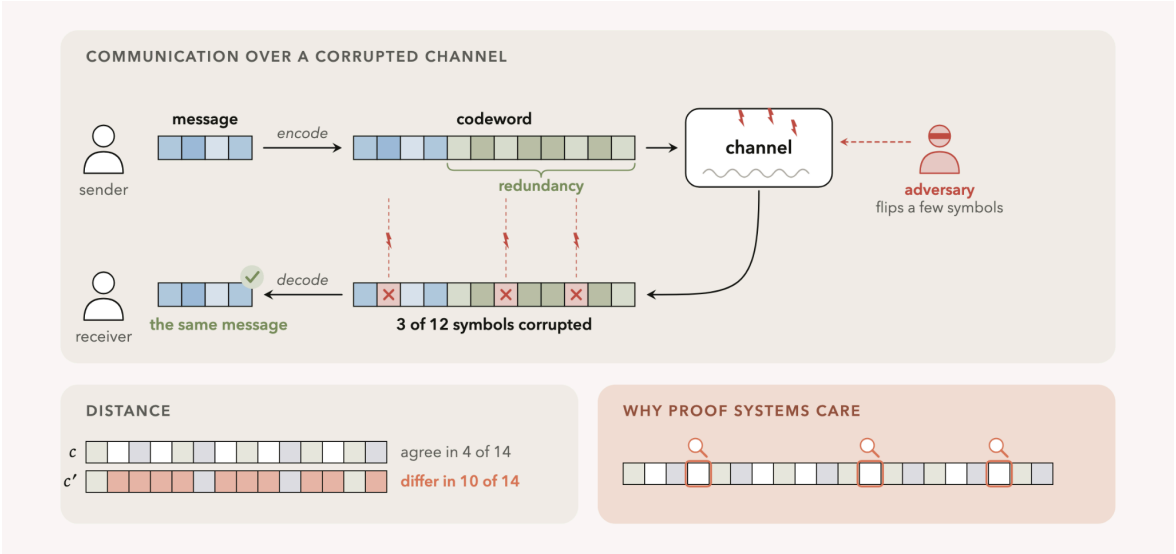
5. Technical overview: a look at the protocol

FlashProve does not claim to be a paradigm shift compared to prior art, nor does it claim to introduce completely new methods. In contrast, FlashProve can be viewed as a bespoke composition of state-of-the-art building blocks developed in the "zkSNARK" and "zkML" research areas. The composition is intricate (and the executable code highly optimized), but its salient feature is that FlashProve fits well-established paradigms. In particular, the majority of the system is stated and proved at the level of an IOP over a given linear code, while hash functions enter only through Merkle commitments and the Fiat–Shamir transformation (with some optimizations). Hence big parts of the system concern only information-theoretic constructs that are well-suited for automated formalization and optimization.

In the following technical overview we highlight some of the ideas borrowed from the IOP/zkSNARK literature as well as from the zkML literature, then explain how these are combined in FlashProve.

5.1 Toolkit for IOPs

IOP constructions crucially rely on error-correcting codes, the same concept underlying modern telecommunications. A code is a map that expands an input message to a longer output *codeword*. The added length creates redundancy so that, e.g., one can recover the original message even if a few entries of the codeword are corrupted.

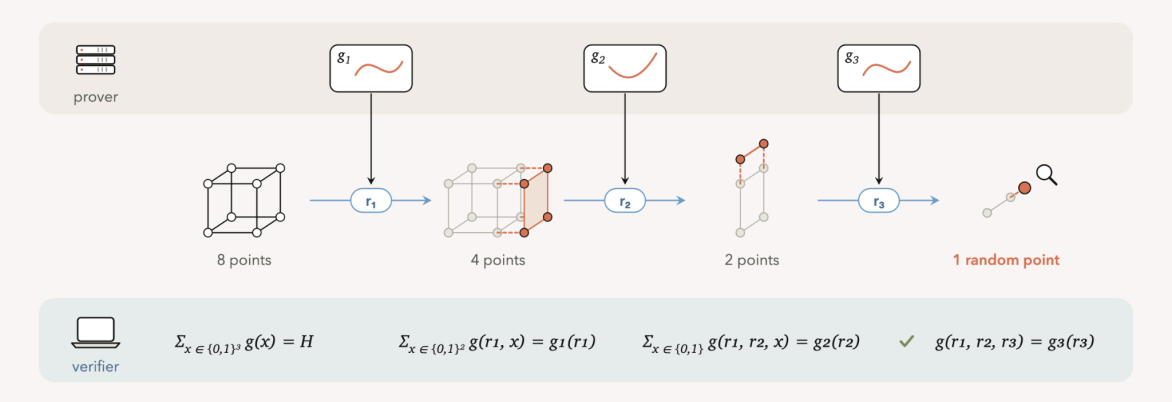


Similarly, instead of adding redundancy to a message, an IOP adds redundancy to a *computation*; the verifier of an IOP is then able to check the correctness of a large computation while making only a few queries to the prover messages and performing cheap tests. Codes play a prominent role in enabling such capabilities. Below we highlight some of the ideas from the IOP literature that we use in FlashProve.

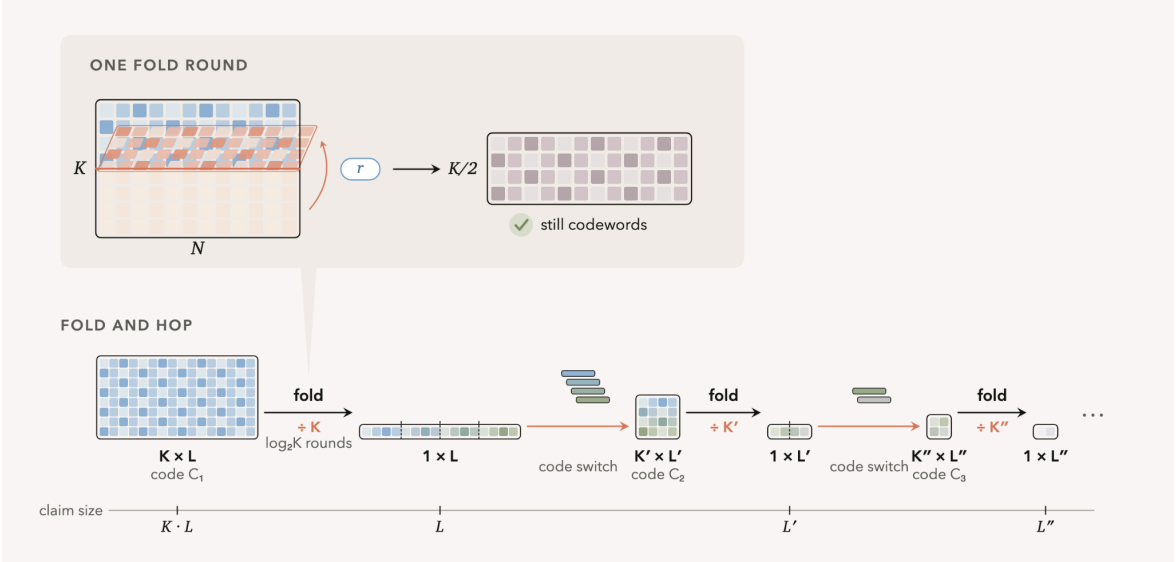
- **Zero-knowledge encoding.** The prover in an IOP typically starts by sending an oracle that encodes the (private) witness of the computation. Yet the answers to queries may leak

information about the private witness. To prevent this leakage, one uses a *zero-knowledge code* [ISVW13, BCL22, CFW26]: a code that encodes a message using randomness to ensure that a bounded number of locations of the codeword do not reveal information about the encoded message.

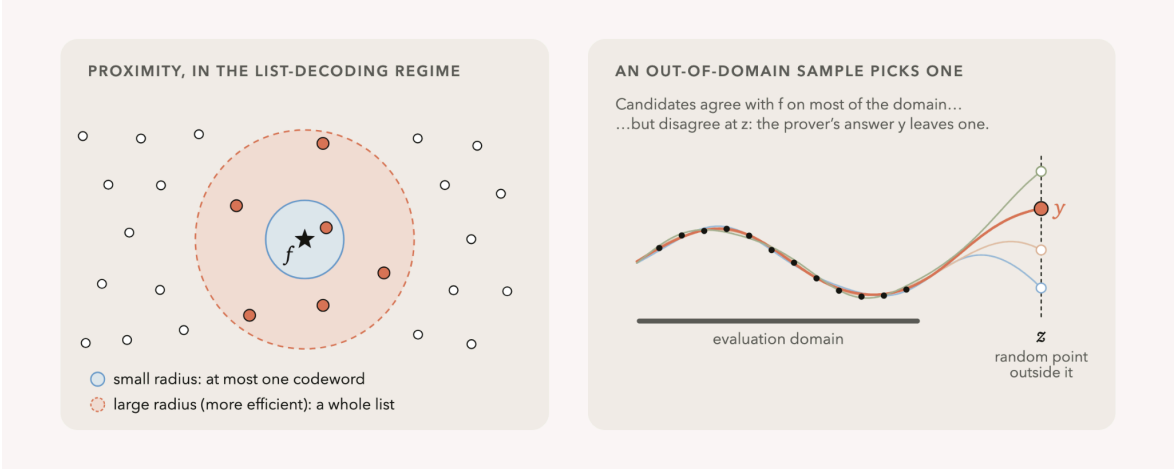
- **Sumcheck.** The *sumcheck protocol* [LFKN92] is the workhorse of probabilistic proofs. It is an elegant interactive protocol that reduces checking the sum of a multivariate polynomial over many points to the value of the polynomial at one random point. This simple primitive is used throughout FlashProve: in folding (next), lookup subroutines, and to reduce computation to claims on the message encoded by sent oracles. This results in two thousand sumchecks per inference request, accounting for 90-95% of all the interaction rounds of the IOP used in FlashProve. For privacy at a small overhead, we use a zero-knowledge variant with small masks [XZZPS19, CFW26].



- **Fold and hop.** Inference claims are reduced to linear claims on the oracles sent by the prover. These linear claims are as large as the computation itself, and to achieve succinctness we progressively reduce problem size. In line with state-of-the-art IOPs [ACFY25, NA25, CFW26], we combine three ingredients: sumcheck, interleaved codes, and code-switching. *Code interleaving* is a method to endow linear error-correcting codes with structure that allows shrinking the problem size [AHIV17, BCG20, GLSTW23, ACFY25, NA25, CFW26] by encoding a message by "stacking" multiple encodings of different message chunks via a small code. The codes we consider are linear, and the sumcheck protocol can be used to "squish" the stacked words together, reducing the size and creating a new claim on a single small encoding. This technique is followed by *code-switching* [RR20] to hop from one code to another: we transfer claims relative to a code to claims relative to another code of the same size. We use this to switch to another interleaved code, allowing us to reduce problem size further (and use codes better suited for small problem sizes via "code rate improvement" [ACFY24, ACFY25]). While the strategy is code-agnostic, we choose to use Reed–Solomon codes for their efficient encoding and zero-knowledge properties [AHIV17].
- **List-decoding and out-of-domain samples.** Succinctness depends on the analysis of the protocol, in particular on the *proximity parameter* that it allows: the (Hamming) distance that oracles can be from an honest codeword. The larger this parameter, the more efficient the protocol, but the harder its analysis. State-of-the-art IOPs operate in the list-decoding regime:



the proximity parameter is large enough that there might be many codewords close to an oracle. We are no exception, and achieving this efficiency requires using out-of-domain samples to uniquely identify a codeword within the list [BGKS20, ACFY24, ACFY25, BCFW25] and using state-of-the-art mutual correlated agreement theorems for Reed–Solomon codes [BCIKS20, BCHKS25, ABF26, ProxPrize].

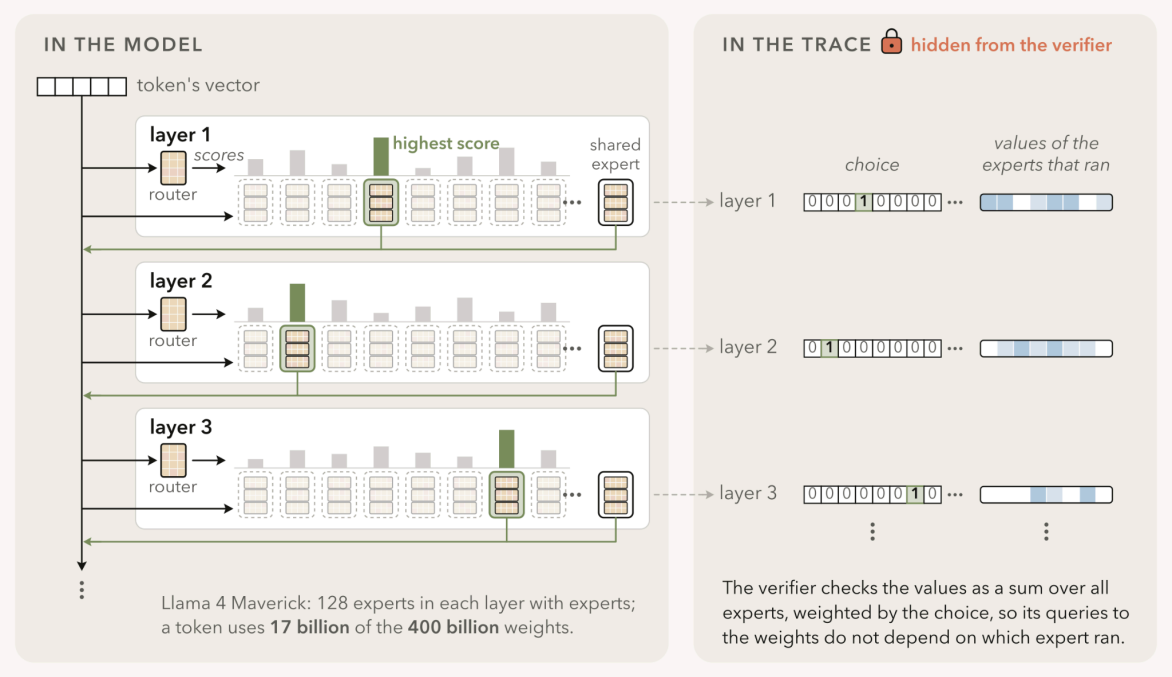


5.2 Toolkit for zkML

FlashProve relies on many ideas developed in the zkML literature. We highlight some of these ideas below in order to give a taste of how the specific structure of inference computations affects the design of a zero-knowledge succinct argument.

- **MoE benefits the prover.** Mixture-of-Experts (MoE) [JJNH91, SMMDLHD17] is a widely used architecture for large language models; such models include Mixtral [Jia24], DeepSeek-V3 [Dee24], Llama 4 [Met25], and gpt-oss [Ope25]. A model processes a token through a sequence of layers, each consisting of a few matrix products with weight matrices. In an MoE model, a

layer may hold many *experts* (groups of matrix products of the same shape) and a *router* (one small matrix product that scores the experts from the token's vector); only the highest-scoring experts run. In Llama 4 Maverick, one of 128 experts runs in each MoE layer, plus a shared expert, so a token uses 17 billion of the model's 400 billion weights. MoE is particularly suitable for IOPs, because *the prover's work per token can match the model's*: the trace (the values that the prover encodes as oracles) holds for each token only the values of the experts that ran and the choice itself. Checking the choice requires only the router's product and a comparison of scores, which is small compared with the products of a single expert. The choices, which depend on the weights, are hidden from the verifier by the zero-knowledge encoding of the trace. The trace holds each choice as an indicator vector over the experts, and the verifier checks the products of the expert that ran as a sum over all experts weighted by the vector's entries, so its queries to the weights do not depend on which expert ran.



- **Quantization via integer arithmetic.** Language models are commonly quantized: their weights and activations are scaled and rounded to small integers, which cuts memory and inference time with a small loss in accuracy [JKC18, XLS23]. Quantization also suits proof systems. Most IOPs work over a finite field $\mathbb{F}_p$, and integer arithmetic coincides with arithmetic in $\mathbb{F}_p$ as long as all input/output values stay in the interval $(-p/2, p/2)$. Many zkML systems prove such integer-quantized models [GGG17, FQZDC21, LXZ21, KHSS22, CWS24, SLZ24, QSL25], since their outputs are considered to be close enough to the true models being served. The additional complication this adds is that values must be kept in this range throughout the computation, and the verifier needs to be convinced of that fact.[5]

[5]The quantized model proved here was chosen to make zkML work. It says nothing about how inference is implemented in production at Anthropic or elsewhere.

- **Lookups.** Native ML computations, even quantized ones, require operations such as nonlinearities and comparisons (e.g. softmax, SiLU, ReLU), which are expensive to express as arithmetic over finite fields. *Lookup protocols* [BCGJM18, GW20, Hab22, STW24] can be used to address this: the nonlinear operations are known in advance and can be written down publicly, much like trigonometric tables. Given a table computing the function f, the prover claims that each pair (x, y) is a row of a public table of all pairs $(x, f(x))$. The verifier can check such a claim via a grand product argument [GWC19] or the sumcheck-based LogUpGKR protocol [GKR08, Tha13, Hab22, PH23]. The lookup method works for any f, regardless of its complexity, but one has to cleverly keep the table size under control, as it affects the complexity of the resulting proof system. In our setting the tables stay small because the model needs each function on only 2^{16} inputs; beyond that, for example, the sigmoid linear unit (SiLU) is cut off and the softmax exponential is rounded to zero.

5.3 A simplified IOP for inference

We explain how to combine the above ideas: we describe the flow of a **simplified IOP for inference**; after that we outline the main differences from the actual implementation.

A modular blueprint. The organizing idea of our IOP for inference is *modularity*. Each piece of the IOP is an **interactive oracle reduction** (IOR) [BCGGRS19, BMNW25, BCFW25, CFW26]: an interactive protocol that reduces a "difficult" claim about an oracle to a "simpler" claim. The sumcheck protocol [LFKN92] is the prototypical example: it reduces checking the sum of a polynomial's evaluations at *many* points to checking the value of the polynomial evaluated at a *single* point (chosen at random during the protocol).

Linear claims. All reductions in the IOP start from high-level objectives of the verifier (e.g., checking that an expert's computation is correct relative to the given model, or checking that values lie in a range) and end in linear claims on the encoded messages. A linear claim for an oracle asserts that the inner product of a vector (known to the verifier and the prover) and the alleged underlying message equals a given target value. Linear claims are convenient because one can batch them via random linear combinations. This final claim includes a proximity claim for the oracle and a linear claim on the message (allegedly) encoded in the oracle; this fits known techniques in the IOP literature [ACFY25, CFW26].

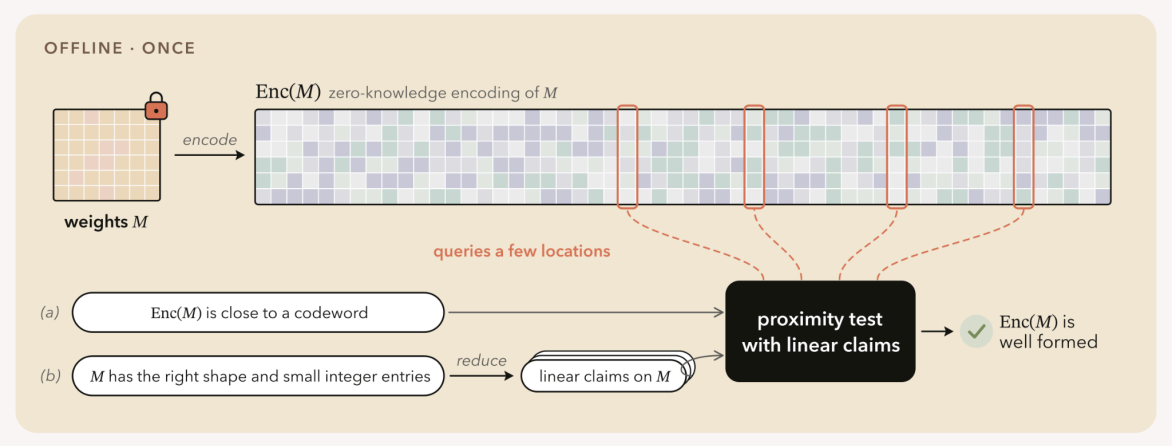
The simplified IOP. To give an idea of how the above components combine, we describe an IOP for proving the inference of a Mixture-of-Experts model given:

- M, a matrix of weights (a matrix of small integers in a fixed public layout); and
- Enc, a zero-knowledge encoding.

The model's architecture is a fixed sequence of steps, known to both parties, each consisting of either a matrix product with a slice of M whose result is requantized (a certain rounding right shift [JKC18]) back to small integers, or an elementwise nonlinearity, whose small-integer output is read from a public table.

- **Offline phase.** The one-time offline phase is to convince the verifier about properties of the committed weights, which are going to be assumed in subsequent online phases.
  1. The prover holds the weights M and sends $\mathrm{Enc}(M)$, the zero-knowledge encoding of M.

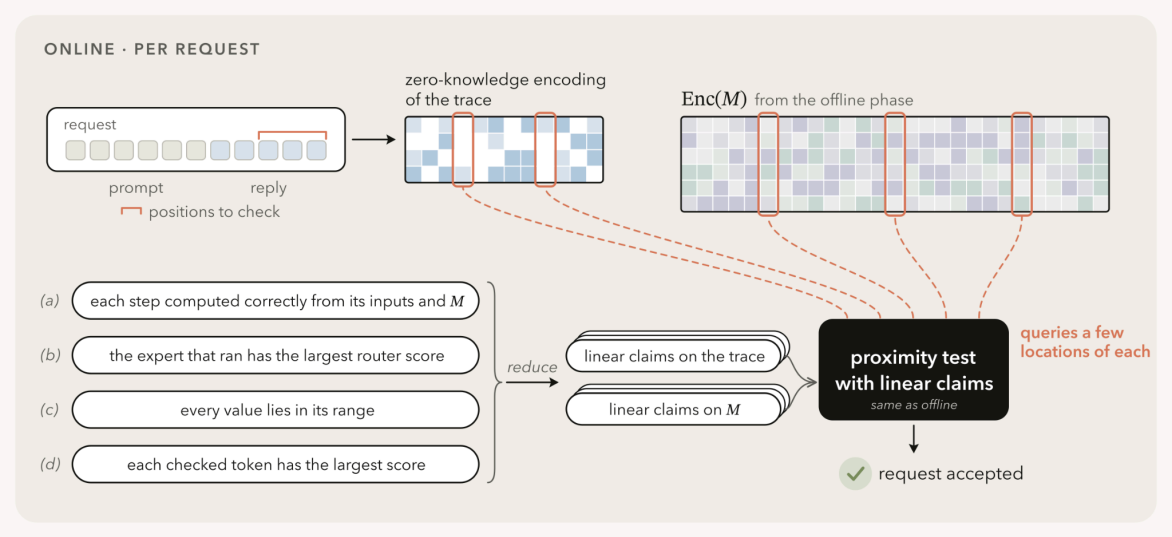
2. The prover and the verifier run an IOP to check that Enc(M) is "well-formed": (a) Enc(M) is close to a codeword; (b) that codeword encodes a matrix (of the prescribed shape) whose entries are small integers in a certain range.
3. The prover and the verifier interactively reduce these claims to multiple linear claims on the message underlying the encoding, and engage in proximity tests (with respect to the oracle Enc(M)) with these linear constraints.



- **Online phase**. The online phase can be run multiple times, for each inference request from the verifier on the model whose encoding was sent (as an oracle) in the offline phase.
  1. A request is a sequence of tokens (the prompt followed by the reply) and a range of positions in it. The claim to verify is that, for every position in the range, the next token is the committed model's *greedy choice* (the token with the largest score among all tokens of the vocabulary).
  2. The verifier sends the request to the prover, who responds with a zero-knowledge encoding of an inference evaluation trace (detailing the computation's value after each multiplication, rounding, or nonlinearity step in the architecture) such that: (a) each step's output is the relevant function of its inputs and of M; (b) in each expert layer, the expert that ran is the one with the largest router score; (c) all values of the trace lie in their ranges, so that arithmetic in the field coincides with arithmetic over the integers; and (d) at every position in the range, the token there has the largest score.
  3. The prover and the verifier again reduce the above assertions to linear claims. Each condition that the verifier would like to check can be formulated as a polynomial constraint between values of the trace and of M, or a lookup of such a polynomial into a public table.
  4. All linear claims of the request, on the trace oracles and on Enc(M), are checked by the same generic proximity tests with linear constraints as in the offline phase. During the execution of these protocols, the verifier queries Enc(M) and the encoding of the trace.

Security guarantees.

- **Soundness.** The soundness properties of the reductions of each of the verifier's claims guarantee that the probability, over the randomness sent by the verifier to the prover during the



protocol, that the verifier approves an incorrect request (some token in the checked range is not the greedy choice of the weights pinned down in the offline phase) is small.

- **Zero knowledge.** As long as the number of queries made by the verifier to the sent oracles is small, the encoded messages (i.e., the model's weights and the evaluation trace) remain hidden. In particular, the specific choice of expert at any step remains hidden (as this may reveal some information about the weights). One may wonder why we can assume that the verifier makes only a small number of queries to the oracles, while in reality the verifier may want to extract information on the weights. This is because the compilation to a non-interactive protocol "enforces" an honest behavior on the verifier's side [BCS16]. (A separate point altogether is how to ensure that the zero-knowledge encoding of the weights sent in the offline phase does not "wear out"; this is handled via *link proofs*, discussed further below.)

The implemented IOP vs. the above description. The above is a simplified version of the IOP in FlashProve, which differs somewhat for the sake of efficiency.

- **Batching.** The simplified IOP proves one request and checks its linear claims on $\mathrm{Enc}(M)$ by a proximity test on $\mathrm{Enc}(M)$. FlashProve proves a *batch* of requests together. Each request keeps its own trace oracles and its own reductions, but their claims on $\mathrm{Enc}(M)$ are combined, so $\mathrm{Enc}(M)$ is queried once per batch rather than once per request. This reduces the number of queries to the encoding, which also matters for the query budget of the zero-knowledge encoding. While a batch mixes requests of different clients, each client receives only its own part of the proof plus the shared part, revealing nothing about the other requests (not even how many there were).
- **Segments.** The simplified IOP sends the whole trace as one oracle, so the prover would hold the trace and its encoding until the proximity test at the end. That is far more than one GPU holds, and FlashProve is designed to run on one GPU. So it splits the model's layers into *segments*: the prover sends the trace of one segment, reduces its claims, and carries forward only the few values that the next segment reads, so GPU memory holds one segment at a time. The cost is more rounds, since the segments are proved one after the other.

- **Approximate range checks.** The simplified IOP checks that every value of the trace lies in its range, one lookup per value. This is expensive, and FlashProve instead checks random sums of the entries of a whole matrix at once. The cost is *imperfect completeness*: a sum of honest values can be large by chance, so an honest prover is sometimes rejected (with tiny probability). Imperfect completeness has implications for zero-knowledge as well: a rejected request might leak information about the weights. This can be fixed via retrying via a different source of randomness; we plan to address this in a future release.

5.4 From IOP to FlashProve

We use BLAKE2 to compile the IOP for inference outlined above into the non-interactive protocol used in FlashProve. This mostly follows the standard recipe outlined in Section 3, but we make some adjustments motivated by our setting. We highlight two: one motivated by hiding weights across many requests; and the other motivated by concurrent executions on a GPU.

Commit-and-prove hash-based arguments. Our setting differs from the usual one in the IOP literature: the weights are committed once and many requests are proved against the same commitment.[6] For security we need:

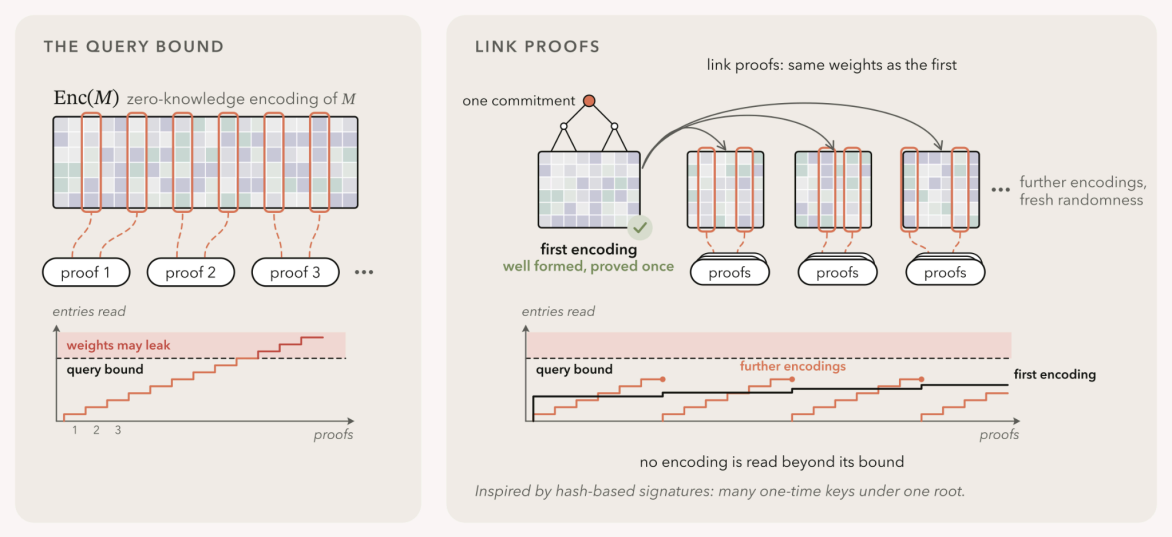
1. **Binding:** no prover can produce accepting proofs for weights other than the committed ones.
2. **Privacy:** the commitment can be reused for many proofs without revealing anything about the weights beyond the requests and their replies.

Using the tools discussed above, one natural approach would be to use a zero-knowledge encoding for the weights. The obstacle is the query bound: an encoding hides the weights only while a bounded number of its entries are queried, and every proof reads a few. Our solution to this is **link proofs**: The prover proves once that its first encoding is well-formed, then sends further encodings of the same weights with fresh randomness, each with a link proof that it holds the same weights as the first; proofs of requests use the further encodings, so no encoding is read beyond its bound, and one commitment supports as many proofs as there are linked encodings. This idea is inspired by similar solutions used in hash-based signatures (see [BHHLNPSSW15] and references therein), where a one-time (or few-time) signature scheme is "lifted" to a many-time signature scheme via a tree-like structure.

Fiat–Shamir on a directed acyclic graph. The Fiat–Shamir transformation (**Step 3**) derives each verifier challenge by hashing the previous challenge together with the prover's new message. This hash chain prevents a prover from choosing a message after seeing its challenge. But it is a strict serialization: round $i + 1$ cannot start until round i is hashed.

FlashProve derives challenges for the underlying IOP in a more general and flexible way. A session proves multiple requests at once. Each request runs its own preamble (its segments, lookups, and $\sim$2000 sumchecks), independently of the others; at the end their claims on $\mathrm{Enc}(M)$ meet in one shared proximity test, after which each request has a few final rounds of its own. The prover runs these preambles concurrently on one GPU, overlapping the serial hashing of one request with the

[6] The same setting, with pairing-based commitments, is studied for zkML in [LVAKH24]. Different techniques are needed for commit-and-prove hash-based SNARKs.



heavy commitment work of another. Each client must receive its own proof, learning nothing about the rest of the batch.

So the chain is **forked and joined**: one chain per request, a join into a shared chain for the shared rounds, and a fork back into per-request chains. The join is a small Merkle tree over the end values of the request chains; the shared chain starts from its root. A client's proof contains its own chain, the shared chain, and the opening of its own leaf. These dependencies form a *directed acyclic graph*.

Why is this secure? In a chain, a challenge binds the whole transcript before it. In a DAG, a challenge binds only the branches it depends on. A cheating prover can finish one request's chain, then choose another request's messages with that knowledge, or rederive one branch repeatedly while holding the others fixed. An IOP property that suffices to absorb this freedom is (a suitable extension of) **round-by-round soundness** [CCHLRR18, CMS19]. Hence we establish this property (with explicit error bounds) for all relevant IOR subprotocols, which is inherited by the final IOP.

6. Implementation: running a 400B model on one GPU

We measured an implementation of the protocol on Llama 4 Maverick, which has 400B parameters and 128 experts, using the publicly available 4-bit weights (converted to the integer model). Each request is a window of 1,024 tokens, and each proof checks one token, the one following the prompt: it certifies that this token is the committed model's greedy choice. We target 128 bits of security.

Prover. The prover runs on the provider's side, on a server with a single H100 and 24 CPU cores. The model's weights take up 232 GB and stay in host memory (the machine's ordinary RAM). The GPU (which has only 80GB of memory) streams the weights in and holds one segment of the trace at a time.

- *The offline phase* runs once per model and it encodes the weights, hashes the encoding into a commitment, and outputs a proof that the encoded weights are well formed. It takes about 39 minutes, and the provider stores 2.15 GB of files.

- *The online phase* batches up to 64 client requests, runs the quantized model, encodes and hashes the trace one segment at a time, runs the sumchecks and lookups, and opens the weight commitment once for the whole batch. **A batch of 24 requests takes about 223 seconds, which is 110 tokens per second. Each proof is 18.78 MB.** Here a token is any of the 1,024 tokens of a request, since the prover processes them all, even though the proof checks only one of them. We also tried checking every token of the reply instead; this made no measurable difference to the speed, which, counted in checked tokens only, comes to 39.1 tokens per second.

Since nobody is waiting on the proof, the prover works on many requests at once, and 110 tokens per second is the rate at which it gets through them; an individual request waits about five minutes, from the time its batch starts until the proofs of the whole batch are released. If we instead prove requests one at a time, a proof is ready in 58 seconds, but the prover only manages 17.6 tokens per second.

Comparison with plain inference: We ran vLLM, a widely used inference engine, with the FP8 version of the same model on eight H100s, and found that, per GPU, **proving a batch of 24 requests takes about 72 times longer (66 to 77 across runs) than simply running the prefill on them, and about 94 times longer than vLLM's prefill at the highest load we tried**.

Verifier. The verifier runs on the client's CPU. It needs no GPU and only has access to a commitment to the weights. The client downloads the commitment, the offline proof and one link once, **6.24 MB in all**, and then one proof per request. **Checking a proof takes 17 seconds on 24 threads (and 21 seconds on a single thread), including the offline phase's checks.**

7. Limitations

We summarize the limitations of FlashProve in this release. Some limitations are inherent to the choices we made (in cryptography, in architecture, and so on); others are a reflection of the fact that we are sharing intermediate progress on this project.

- **Protocol.** There are meaningful optimizations that we did not incorporate in the cryptographic protocol underlying FlashProve. These include optimizations in prover time, prover memory, argument size, and verifier time that would result from a more careful application of the IOP and zkML toolkits as well as fine-tuning the hash-based compilation.
- **Technical report.** The technical paper that describes the protocol was AI-generated, and we did not review it in detail. A carefully-reviewed paper will follow later.
- **Lean code.** The Lean code formally specifies the prover and the verifier algorithms at the information-theoretic (that is, IOP) layer; formal proofs of correctness and security are proved at this layer. Then, they are lifted to proofs about succinct arguments in the ROM via an (adaptation of) the BCS transformation. So far, zero knowledge is formally proved when requests are proved one at a time, and not yet when they are proved together in a batch. The Lean proofs depend (as always) on trusting the Lean kernel; moreover, if you run the Lean verifier, you are also implicitly trusting the compiler that interprets and executes this code.
- **Executable code.** The GPU code of the prover is generated from its Lean representation by AI (and is not proved equivalent to it). Any “mistake” in the prover code could affect completeness (even an honest prover may fail to convince the verifier) or zero-knowledge (the prover may

inadvertently leak information about the weights); crucially, however, any mistake in the prover code cannot affect soundness; indeed, the verifier needs to only trust itself to ensure correctness of the request answer. The CPU code of the verifier is also AI-generated from its Lean representation; for higher assurance, one could alternatively directly run the Lean verifier (thereby only trusting the Lean kernel and the process that executes Lean) or one could prove formal equivalence to a formally specified machine language (e.g., RISC-V) and then rely on a high-assurance execution stack for that machine language.

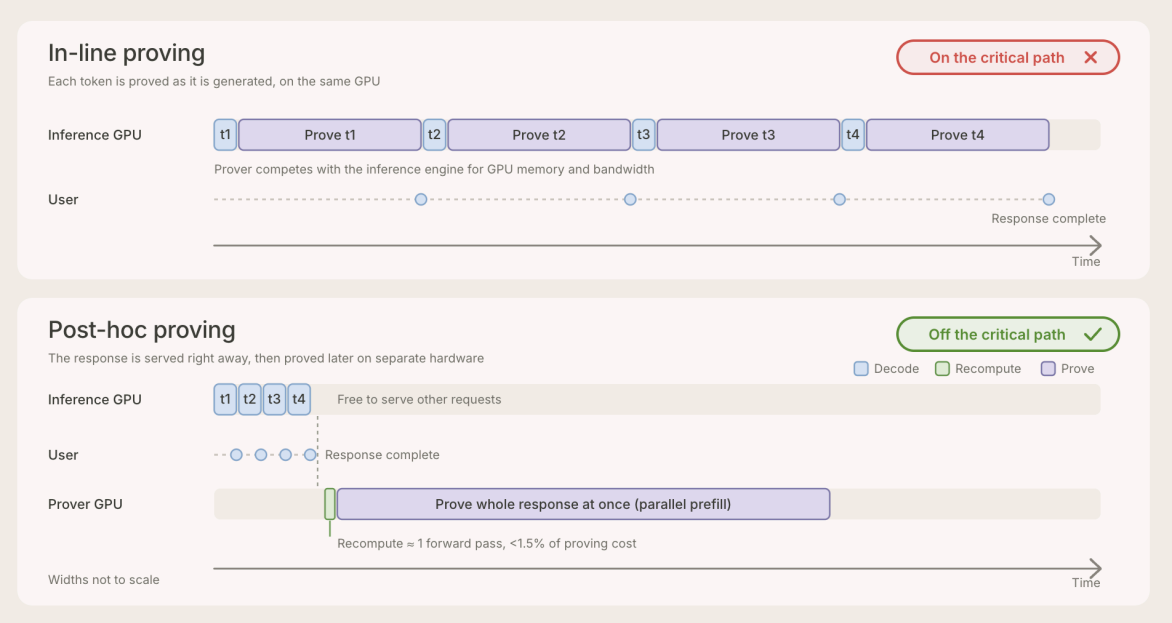
- **Integer model under greedy decoding or fixed-seed sampling**. Floating-point inference does not reproduce exactly across machines, so we prove an exact integer version of the published weights, asserting that each checked token is that model's greedy choice. A reply from a floating-point implementation, or sampled decoding, could depart at some position and no longer exactly match the computation proved. The integer version is generally expected and empirically tested to be within some small error rate of true floating-point inference, though this does not cover adversarially designed prompts. See for example [RKHRW25] for issues this might cause.
- **Parallelism**. The prover runs on one GPU: the weights are stored in host memory and are streamed to the GPU when generating a proof. Tensor parallelism and multi-GPU proving, as used in serving, are not currently supported. A multi-GPU prover, whose main gain would be lower latency for a single request, is plausible with the techniques we use but is left for future work.

Appendix: post-hoc verification & non-determinism

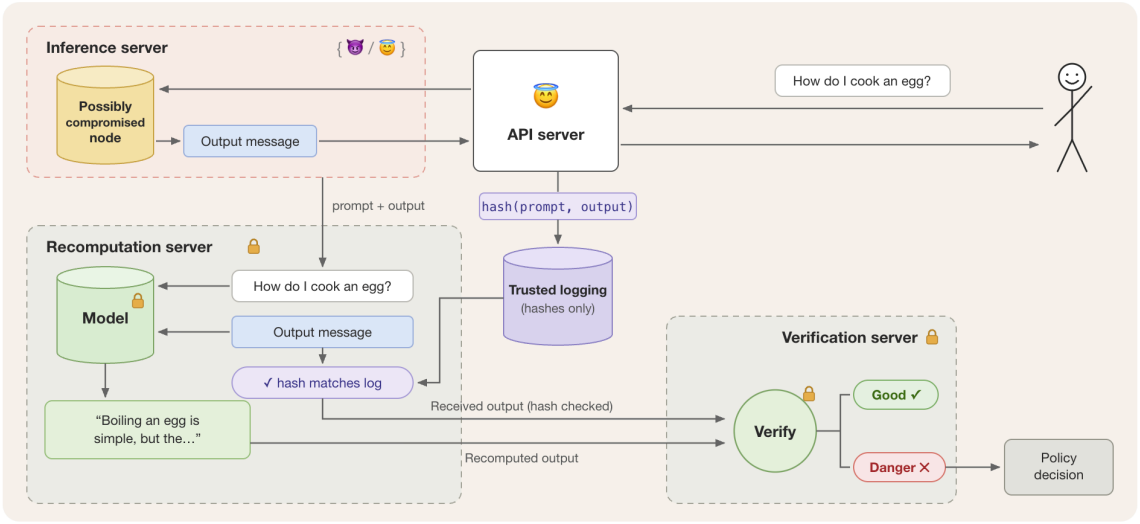
FlashProve is for post-hoc proof generation, where model responses are served to users immediately, and proofs are generated afterwards on separate proving servers via recomputation using the finished transcript. This has implications for security, efficiency, numerics and more, which we discuss below.

Efficiency & complexity. Post-hoc verification takes proving off the critical path for inference. Proving is much more expensive in terms of GPU time than inference and would contend with the inference engine for hardware resources like memory and bandwidth if run on the same machine. Decoupling inference from proving offers the benefits of proofs without degrading user experience, and ensures that inference engines do not have to concern themselves with co-optimizing around FlashProve's prover. Proving requests post-hoc also enables generating a proof for a whole response all at once and in parallel, which is much more efficient on GPUs due to their constrained memory bandwidth relative to flops. In inference engineering terms, this means we get to focus on proving *prefill* (which is highly parallelizable and compute bound) instead of *decode* (which is inherently sequential and typically memory bandwidth bound). In-line verification (generating proofs while tokens are generated) could let us avoid recomputing the model's activations, but the savings are small. Recomputation costs about one forward pass, while proving costs about 70 to 95 times more than that, so recomputing adds about 1 to 1.5% to the prover's work. This is typical of zkSNARKs, even specialized SOTA provers run 200+ times slower than the proved computations.

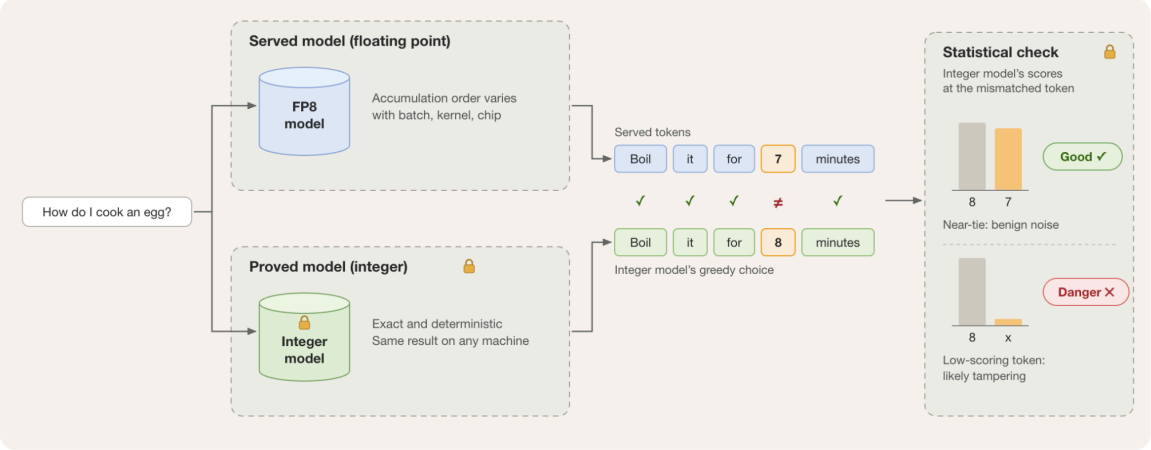
Security implications. Post-hoc proving does not *prevent* users from being served incorrect responses, it only ensures tampering will be *detected*. This can be remedied somewhat by waiting



until a proof is generated to view a response, but this is not sufficient for all use cases of provable inference. For example, if the proofs are being used to defend against data exfiltration from inference servers, an adversary could still leak key bits of information via a tampered response before a failed proof generation signals to the operator that a server is compromised. The operator would also have to trust their logging infrastructure is faithfully recording the contents of responses, as an adversary who compromises this could simply record the "correct" outputs of the model in the logs while sending tampered outputs to users/themselves. We see post-hoc verification as part of a defense in depth approach, which alongside other safeguards can help make a model provider more secure against this kind of exfiltration.



On numerics. The outputs of inference are often non-deterministic and subtly dependent on factors that are hard to pin down. This mostly comes down to floating point numbers being non-associative, so different accumulation orders across chips or operations that mix data across batches can alter the logits of a response. FlashProve addresses this by proving a statement about an *integer quantized* version of a model, as integers are associative and do not suffer from the same numeric issues as floats. This absolves the prover of any numerics issues, but since models are typically served with floats, a served response can diverge from the integer model's output even when it hasn't been tampered with. Luckily, there are practical statistical methods for separating such benign numerical noise from tampering attempts [KRRMGW25], which can be implemented to accommodate these differences while still not allowing much room for an attacker to hide information in responses [RKHRW25].



Bibliography

[ABF26] Gal Arnon, Dan Boneh, and Giacomo Fenzi, Open problems in list decoding and correlated agreement. Cryptology ePrint Archive, Paper 2026/680.

[ACFY24] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev, STIR: Reed–Solomon proximity testing with fewer queries. In the 44th Annual International Cryptology Conference (CRYPTO 2024), pp. 380–413.

[ACFY25] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev, WHIR: Reed–Solomon proximity testing with super-fast verification. In the 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2025), pp. 214–243.

[AGGLNST26] Jeremy Avigad, Anat Ganor, Lior Goldberg, David Levit, Ohad Nir, Yoav Seginer, and Alon Titelman, Formal verification of the S-two AIR. arXiv:2606.04311, 2026.

[AGLST22] Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman, A verified algebraic representation of Cairo program execution. In the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2022), pp. 153–165.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam, Ligero: lightweight sublinear arguments without a trusted setup. In the 24th ACM Conference on Computer and Communications Security (CCS 2017), pp. 2087–2104.

[ALMSS98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy, Proof verification and the hardness of approximation problems. Journal of the ACM 45(3), 1998, pp. 501–555.

[AS98] Sanjeev Arora and Shmuel Safra, Probabilistic checking of proofs: a new characterization of NP. Journal of the ACM 45(1), 1998, pp. 70–122.

[BBHMR19] James Bartusek, Liron Bronfman, Justin Holmgren, Fermi Ma, and Ron D. Rothblum, On the (in)security of Kilian-based SNARGs. In the 17th International Conference on Theory of Cryptography (TCC 2019), pp. 522–551.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev, Scalable zero knowledge with no trusted setup. In the 39th Annual International Cryptology Conference (CRYPTO 2019), pp. 701–732.

[BCFW25] Benedikt Bünz, Alessandro Chiesa, Giacomo Fenzi, and William Wang, Linear-time accumulation schemes. In the 23rd Theory of Cryptography Conference (TCC 2025), pp. 369–399.

[BCG20] Jonathan Bootle, Alessandro Chiesa, and Jens Groth, Linear-time arguments with sublinear verification from tensor codes. In the 18th Theory of Cryptography Conference (TCC 2020).

[BCGGRS19] Eli Ben-Sasson, Alessandro Chiesa, Lior Goldberg, Tom Gur, Michael Riabzev, and Nicholas Spooner, Linear-size constant-query IOPs for delegating computation. In the 17th Theory of Cryptography Conference (TCC 2019), pp. 494–521.

[BCGJM18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller, Arya: nearly linear-time zero-knowledge proofs for correct program execution. In the 24th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2018), pp. 595–626.

[BCHKS25] Eli Ben-Sasson, Dan Carmon, Ulrich Haböck, Swastik Kopparty, and Shubhangi Saraf, On proximity gaps for Reed–Solomon codes. Cryptology ePrint Archive, Paper 2025/2055.

[BCIKS20] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf, Proximity gaps for Reed–Solomon codes. In the 61st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2020). Journal version: Journal of the ACM 70(5), 2023.

[BCL22] Jonathan Bootle, Alessandro Chiesa, and Siqi Liu, Zero-knowledge IOPs with linear-time prover and polylogarithmic-time verifier. In the 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2022), pp. 275–304.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner, *Interactive oracle proofs*. In the 14th International Conference on Theory of Cryptography (TCC 2016-B), pp. 31–60.

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy, Checking computations in polylogarithmic time. In the 23rd Annual ACM Symposium on Theory of Computing (STOC 1991), pp. 21–32.

[BGKS20] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf, DEEP-FRI: Sampling outside the box improves soundness. In the 11th Innovations in Theoretical Computer Science Conference (ITCS 2020), pp. 5:1–5:32.

[BHHLNPSSW15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn, SPHINCS: practical stateless hash-based signatures. In the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2015), pp. 368–397.

[BM88] László Babai and Shlomo Moran, Arthur–Merlin games: a randomized proof system, and a hierarchy of complexity classes. Journal of Computer and System Sciences 36(2), 1988, pp. 254–276.

[BM24] Bolton Bailey and Andrew Miller, Formalizing soundness proofs of linear PCP SNARKs. In the 33rd USENIX Security Symposium (USENIX Security 2024), pp. 1489–1506.

[BMNW25] Benedikt Bünz, Pratyush Mishra, Wilson Nguyen, and William Wang, Arc: accumulation for Reed–Solomon codes. In the 45th Annual International Cryptology Conference (CRYPTO 2025), pp. 128–160.

[BR93] Mihir Bellare and Phillip Rogaway, Random oracles are practical: a paradigm for designing efficient protocols. In the 1st ACM Conference on Computer and Communications Security (CCS 1993), pp. 62–73.

[CCHLRR18] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, and Ron D. Rothblum, Fiat–Shamir from simpler assumptions. Cryptology ePrint Archive, Paper 2018/1004.

[CFS17] Alessandro Chiesa, Michael A. Forbes, and Nicholas Spooner, A zero knowledge sumcheck and its applications. Cryptology ePrint Archive, Paper 2017/305. 2017.

[CFW26] Alessandro Chiesa, Giacomo Fenzi, and Guy Weissenberg, Zero-knowledge IOPPs for constrained interleaved codes. Cryptology ePrint Archive, Paper 2026/391.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi, The random oracle methodology, revisited. In Journal of the ACM 51.4 (2004), pp. 557–594.

[CMS19] Alessandro Chiesa, Peter Manohar, and Nicholas Spooner, Succinct arguments in the quantum random oracle model. In the 17th Theory of Cryptography Conference (TCC 2019), pp. 1–29.

[CWS24] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang, ZKML: an optimizing system for ML inference in zero-knowledge proofs. In the 19th European Conference on Computer Systems (EuroSys 2024), pp. 560–574.
[CY24] Alessandro Chiesa and Eylon Yogev, Building Cryptographic Proofs from Hash Functions. Online textbook, 2024.
[Dee24] DeepSeek-AI, DeepSeek-V3 technical report. arXiv:2412.19437, 2024.
[Fen26] Giacomo Fenzi, How to prove more false statements: Fiat–Shamir limitations on (generated) R1CS. Cryptology ePrint Archive, Paper 2026/1838.
[FGLSS96] Uriel Feige, Shafi Goldwasser, László Lovász, Shmuel Safra, and Mario Szegedy, Interactive proofs and the hardness of approximating cliques. Journal of the ACM 43(2), 1996, pp. 268–292.
[FQZDC21] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu, ZEN: an optimizing compiler for verifiable, zero-knowledge neural network inferences. Cryptology ePrint Archive, Paper 2021/087. 2021.
[FS86] Amos Fiat and Adi Shamir, How to prove yourself: practical solutions to identification and signature problems. In Advances in Cryptology (CRYPTO 1986), pp. 186–194.
[GGG17] Zahra Ghodsi, Tianyu Gu, and Siddharth Garg, SafetyNets: verifiable execution of deep neural networks on an untrusted cloud. In Advances in Neural Information Processing Systems 30 (NeurIPS 2017).
[GK03] Shafi Goldwasser and Yael Tauman Kalai, On the (in)security of the Fiat–Shamir paradigm. In the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2003), pp. 102–113.
[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy Rothblum, Delegating computation: interactive proofs for muggles. In the 40th Annual ACM Symposium on Theory of Computing (STOC 2008), pp. 113–122.
[GLSTW23] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby, Brakedown: linear-time and field-agnostic SNARKs for R1CS. In the 43rd Annual International Cryptology Conference (CRYPTO 2023), pp. 193–226.
[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff, The knowledge complexity of interactive proof systems. SIAM Journal on Computing 18(1), 1989, pp. 186–208.
[GW20] Ariel Gabizon and Zachary J. Williamson, plookup: a simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Paper 2020/315.
[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru, PLONK: permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953. 2019.
[Hab22] Ulrich Haböck, Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530.
[ISVW13] Yuval Ishai, Amit Sahai, Michael Viderman, and Mor Weiss, Zero knowledge LTCs and their applications. In the 16th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and the 17th International Workshop on Randomization and Computation (APPROX-RANDOM 2013), pp. 607–622.
[Jia24] Albert Q. Jiang et al., Mixtral of experts. arXiv:2401.04088, 2024.
[JJNH91] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton, Adaptive mixtures of local experts. Neural Computation 3(1), 1991, pp. 79–87.
[JKC18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko, Quantization and training of neural networks for efficient integer-arithmetic-only inference. In the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2018), pp. 2704–2713.
[Kil92] Joe Kilian, A note on efficient zero-knowledge proofs and arguments. In the 24th Symposium on Theory of Computing (STOC 1992), pp. 723–732.
[KCT26] Natalia Klaus, Juan Conejero, and Palina Tolmach, A Rust-to-Lean verification pipeline with AI provers: an experience report. arXiv:2605.30106, 2026.
[KHSS22] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun, Scaling up trustless DNN inference with zero-knowledge proofs. arXiv:2210.08674, 2022.
[KRRMGW25] Adam Karvonen, Daniel Reuter, Roy Rinberg, Luke Marks, Adrià Garriga-Alonso, and Keri Warr, DiFR: inference verification despite nondeterminism. arXiv:2511.20621, 2025.
[KRS25] Dmitry Khovratovich, Ron D. Rothblum, and Lev Soukhanov, How to prove false statements: practical attacks on Fiat–Shamir. Cryptology ePrint Archive, Paper 2025/118.

[LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan, Algebraic methods for interactive proof systems. In Journal of the ACM 39.4 (1992), pp. 859–868.
[LVAKH24] Hidde Lycklama, Alexander Viand, Nikolay Avramov, Nicolas Küchler, and Anwar Hithnawi, Artemis: efficient commit-and-prove SNARKs for zkML. arXiv:2409.12055, 2024.
[LXZ21] Tianyi Liu, Xiang Xie, and Yupeng Zhang, zkCNN: zero knowledge proofs for convolutional neural network predictions and accuracy. In the 28th ACM Conference on Computer and Communications Security (CCS 2021), pp. 2968–2985.
[Mar26] Diego Marmsoler, Isabelle/STARK: a formalization of zk-STARK in Isabelle/HOL. arXiv:2608.01965, 2026.
[Met25] Meta AI, The Llama 4 herd: the beginning of a new era of natively multimodal AI innovation. Blog post, April 2025.
[Mic00] Silvio Micali, Computationally sound proofs. In SIAM Journal on Computing 30.4 (2000), pp. 1253–1298.
[NA25] Andrija Novakovic and Guillermo Angeris, Ligerito: A small and concretely fast polynomial commitment scheme. Cryptology ePrint Archive, Paper 2025/1187.
[Ope25] OpenAI, gpt-oss-120b & gpt-oss-20b model card. arXiv:2508.10925, 2025.
[PH23] Shahar Papini and Ulrich Haböck, Improving logarithmic derivative lookups using GKR. Cryptology ePrint Archive, Paper 2023/1284.
[ProxPrize] Ethereum Foundation, The Proximity Prize. 2025.
[QSL25] Wenjie Qu, Yijun Sun, Xuanming Liu, Tao Lu, Yanpei Guo, Kai Chen, and Jiaheng Zhang, zkGPT: an efficient non-interactive zero-knowledge proof framework for LLM inference. In the 34th USENIX Security Symposium (USENIX Security 2025).
[RKHRW25] Roy Rinberg, Adam Karvonen, Alexander Hoover, Daniel Reuter, and Keri Warr, Verifying LLM inference to detect model weight exfiltration. arXiv:2511.02620, 2025.
[RR20] Noga Ron-Zewi and Ron D. Rothblum, Local proofs approaching the witness length. In the 61st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2020), pp. 846–857.
[RRR16] Omer Reingold, Ron Rothblum, and Guy Rothblum, Constant-round interactive proofs for delegating computation. In the 48th Annual ACM Symposium on Theory of Computing (STOC 2016), pp. 49–62.
[SLZ24] Haochen Sun, Jason Li, and Hongyang Zhang, zkLLM: zero knowledge proofs for large language models. In the 31st ACM Conference on Computer and Communications Security (CCS 2024).
[SMMDLHD17] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean, Outrageously large neural networks: the sparsely-gated mixture-of-experts layer. In the 5th International Conference on Learning Representations (ICLR 2017).
[STW24] Srinath Setty, Justin Thaler, and Riad Wahby, Unlocking the lookup singularity with Lasso. In the 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2024), pp. 180–209.
[Tha13] Justin Thaler, Time-optimal interactive proofs for circuit evaluation. In the 33rd Annual International Cryptology Conference (CRYPTO 2013), pp. 71–89.
[TH24] Devon Tuma and Nicholas Hopper, VCVio: a formally verified forking lemma and Fiat–Shamir transform, via a flexible and expressive oracle representation. Cryptology ePrint Archive, Paper 2024/1819.
[TDWHH26] Devon Tuma, Quang Dao, James Waters, Alexander Hicks, and Nicholas Hopper, VCVio: verified cryptography in Lean via oracle effects and handlers. Cryptology ePrint Archive, Paper 2026/899.
[XLS23] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han, SmoothQuant: accurate and efficient post-training quantization for large language models. In the 40th International Conference on Machine Learning (ICML 2023), PMLR 202.
[XZZPS19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song, Libra: Succinct zero-knowledge proofs with optimal prover computation. In the 39th Annual International Cryptology Conference (CRYPTO 2019), pp. 733–764.