FlashProve: Hash-Based Proofs of LLM Inference at 400 Billion Parameters

This paper is based in part on a collaboration with Alessandro Chiesa, Giacomo Fenzi, and Guy Weissenberg, who provided ideas and direction towards the current protocol.

Abstract

FlashProve is a hash-based commit-and-prove argument for language-model inference: a provider commits to a model's weights, then proves its greedy outputs correct. Three properties of the protocol are machine-checked in Lean, at exact match, in the random-oracle model, for a configuration fixed before the oracle is sampled, each under the conditions stated in Section 1.3: index-binding knowledge soundness (error about $q/2^{124} + 2^{-101}$ after q queries) and, for one request per proof, completeness on served requests (error at most 2^{-40}) and zero knowledge (error below 2^{-154}, up to 688,127 replies and 2^{64} hash queries); one bit per request, which depends on the weights, is revealed. C++ and CUDA programs produce proofs for a 400-billion-parameter mixture-of-experts integer model on public weights, on one H100 at a steady rate of about 110 prefill tokens per second, in batches of 24 prompts, with the masking of the weights on; each proof has 18.74 to 18.81 MB, checks one output token, and has been checked by the C++ verifier alone. The theorems concern the protocol and its prover in Lean, not these programs; that the programs likely hide the weights, at the default configuration, in honest runs, under conditions that the release's documentation lists, is our judgement, not a proof, and does not cover a real model's weights.

Contents

1 Introduction **2**
2 Preliminaries and definitions **8**
3 The relation: model, policies, configuration **11**
4 The protocol **14**
5 Soundness **20**
6 Zero knowledge and completeness **26**
7 Evaluation **40**
8 Limitations and open problems **46**
9 Related work **50**
Who wrote and checked this **51**
A Proof of the soundness theorem **56**
B Further statements and proofs for Section 5 **66**
C Proofs on zero knowledge **70**
D Sub-protocols, parameters and round errors **77**
E Sources and credit **82**
F Notation and Lean names **87**
G Further limitations **94**
H The artifact and the evaluation in detail **98**

1 Introduction

1.1 The problem

A client of a hosted language model has difficulty checking which model produced its text [CSZS25]: the weights are secret.

In FlashProve, a hash-based commit-and-prove argument [CFQ19], the provider publishes a commitment to its weights M and proves, for a prompt x and an output y, that y is the greedy output of committed weights on x (at exact match; Section 1.3 says which weights). The task resembles that of a function-hiding functional commitment [BNO21]. Since floating-point inference differs between machines [DiFR], the proofs concern an integer model with the shape of Llama 4 Maverick (Definition 3.1). *Exact match* requires the greedy token at every token position; a *clipped-gap policy*, which builds on the logit margin of Token-DiFR [DiFR], bounds the clipped logit gaps on average (Definition 3.4).

The goals are soundness and zero knowledge, formally *index privacy* (Definition 2.14); what is proved, after compilation, is zero knowledge with request leakage (Definition 6.34; Remark 6.40). By design, every published value that depends on the weights is masked or enters only through a salted hash, apart from a declared list (the release's `docs/what-is-not-hidden.md`; Definition H.1) and from what a client learns by design; in Lean zero knowledge is proved for the protocol with one request per proof, under exact match (Section 6.5, with its bounds), and nothing for the programs: that the programs reveal nothing more is the design's aim, not an established property.

1.2 Overview of the protocol

Informally, the protocol (Section 4) is as follows.

- *Masked encoding.* The weights, written as a matrix of digits, are encoded row by row, over a 64-bit prime field, by a Reed–Solomon code with t random lowest coefficients ([CFWpc]; [CY24, Section 32.5]; [CFW26]). Any t columns of this *committed table* are independent of the weights; t is the *opening budget* (Definition 4.2).
- *Copies.* Every proof opens columns, hence the provider makes *copies* with fresh randomness ([CFWpc]; [CFW26]). An *offline proof* shows that the first table decodes to a matrix in range; a *link proof*, that a copy decodes to the same matrix; an *online proof*, that the output meets the policy at that matrix (Definition 4.6 and Definition 4.7).
- *Out-of-domain samples.* The proximity tests [CFW26] run beyond unique decoding, strictly below the Johnson radius, where fewer opened columns suffice but a word can be close to several codewords. Masked linear forms in each table's rows, published at a point derived from its root, determine one of these up to a small error (Definition 4.5; [CFWpc]; the masks: [CFW26, version of February 25, 2026, Section 9.2]; the technique: [BGKS20], as in [ACFY24b, Section 4.4]).
- *Compilation.* Rounds that sample positions have proof of work (Construction 4.13; [CFWpc], with published uses in [Sta21, Section 3.11.3], [ACFY24a, Section 6.2] and [ACFY24b, Section 6.2]). Merkle trees and the Fiat–Shamir transformation make the proofs non-interactive [BCS16, COS20].
- *Cost.* The weights, not the tokens, dominate the cost [LY26]: the table is committed once, and a batch opens one copy once for all its requests, in at most ν_{on} columns (Table 2), as in Brakedown [GLSTW23] and TensorSwitch [BFRW25] (see also [CFW26]). The model's run is proved segment by segment, so that one GPU's memory can suffice (Appendix E.2).

1.3 Results

Statements here are informal; Table 1 lists the theorems by object, and Section 1.4 gives their chief limits. The theorems, machine-checked in Lean, concern the protocol as Lean states it; the C++ and CUDA programs are tested and measured, not proved. Under human direction, AI systems wrote this paper and almost all of the programs and the Lean library, and made every test, calculation, reading and review that it reports; as far as we know, no person has checked these tests, calculations, readings and reviews (Who wrote and checked this).

The protocol. $\mathsf{FlashProve}_Z$ consists of three interactive oracle proofs (IOPs; Construction 4.13), with honest provers written in Lean as definitions, the *Lean provers* (Definition 6.14); compilation turns them into the compiled Lean argument Π_Z^{lp}: a committer, a prover and the verifier V_Z, which is also the verifier of Π_Z (Construction 4.15 and Definition 6.15). Π_Z is the argument whose committer forms several copies and whose prover serves batches; in the Lean statements of Section 5 its committer and prover are constant functions (Remark 6.16).

The *configuration* Z, three byte strings called its files, states all parameters and the policy; an *admissible configuration* equals the default one, Z_0, up to its scales and eight numbers, within stated bounds (Definition 3.8). Its files are *in use* if, moreover, the policy is exact match, which fixes three of the eight numbers, and the *parameter checks* pass, without which V_Z rejects (Definition 6.11); those of Z_0, as written out in Lean, are in use (Proposition 6.17).

Three properties are machine-checked, in the random-oracle model: soundness, a property of the verifier alone, and completeness and zero knowledge, for the protocol whose verifier is proved sound: their provers are those of the IOPs from which V_Z is compiled, and V_Z itself judges the compiled prover, all of one compiled system, Π_Z^{lp} (Remark 6.46).

> **Main theorem** (informal; Theorem 5.1, machine-checked). *Let Z be an admissible configuration with exact match, fixed before the oracle is sampled. In the random-oracle model with a second hash function H' fixed (Assumption 4.11), there is an extractor that reads weights M^* off the commitment and the committer's hash queries, before any proof is made, such that for all committers and provers whose bounds on queries add up to at most q*
>
> $$\Pr\big[\, V_Z \text{ accepts } (x,y), \text{ and } y \text{ is not a greedy output of } M^* \text{ on } x \,\big] \;\le\; \varepsilon(q) \;\approx\; q/2^{124} + 2^{-101}.$$

The property is index-binding knowledge soundness (Definition 2.7; Remark 2.8 (Names of the properties)). The weights M^* need not be the committer's own, no witness is extracted, and the extractor is a function, not shown computable (Remark 5.3, Reading the theorem). For the committer compiled from the Lean provers (Definition 6.15), at files in use (Definition 6.11), within its budget of 2^{48} hash queries (Proposition 6.33), the extractor gives back the committed weights except with probability about 2^{-155} (Theorem 6.31).

> **Completeness** (informal; Theorem 6.29, machine-checked). *Let the files of Z be in use. At every* served *request, that is, a well-formed request whose claimed output is the model's result on it (Definition 6.12), V_Z rejects the Lean prover's proof with probability at most 2^{-40}, over the hash function and the committer's and the prover's coins.*

> **Zero knowledge** (informal; Theorem 6.35 and Theorem 6.37, machine-checked). *Let the files of Z be in use. Against any verifier with at most 2^{64} hash queries, none of them before the commitment, and up to 688,127 answered requests (replies) on one copy of the committed table: if every request is served, one simulator, fixed before the weights, that gets each request and the largest number of replies, and beside these nothing of the weights, makes a view within 2^{-154} of the real one; it may depend on the files of Z, and whether a request is served is not hidden. At all requests the same holds of a simulator that gets besides one bit for each request: served or not.*

Both are in the model of the main theorem, with the files of Z and the weights fixed before the hash function is sampled, and are stated for one request per proof, one commitment, one copy of the committed table and fresh random masks (Remark 6.42). The bit says whether the claimed output is the model's result (Remark 6.44). The property is *zero knowledge with request leakage* (Definition 6.34); neither it nor the form before compilation is stated to be index privacy (Remark 6.40). On served requests completeness and zero knowledge are stated through the texts of two definitions shaped with a person, each with one condition more, that the request can be served (Definition 3.11); with the condition that the output is the model's, these are the served requests; the forms at every request are the Lean library's own (Remark 6.47 and Limitation G.7).

Further machine-checked statements, and the check. Table 1 gives the properties before compilation and on the commitment; Section 5.5 and Section 6 have results on other objects, each with its limits.

Table 1: What is proved, of which object: statement and bound, simplified. Every entry that does not say otherwise is *machine-checked*: Lean's kernel accepts a proof of the Lean statement that Appendix F names; where a statement is an instance of named machine-checked theorems by a substitution that no single Lean theorem states, its remark says so (Remark 6.32). Rows with hash queries are statements in the random-oracle model. A property that is not listed for an object is not stated of it.

Object; property	Statement and bound
V_Z, committer and prover compiled from the Lean provers (6.15)	
index-binding knowledge soundness	5.1, about $q/2^{124} + 2^{-101}$; stated of Π_Z (4.15), which has the same verifier (6.16)
files in use (6.11), one request per proof, one commitment, fresh random masks	
soundness with extractability of the commitment	6.31, about $q/2^{124} + 2^{-100}$ (q: the hash queries of whoever made the commitment and of the prover or the opener, together); no two openings, twice that
correctness of extraction	6.31, about 2^{-155}, at the honest committer's budget
completeness	6.29, on served requests, $\leq 2^{-40}$, on one copy of the committed table
zero knowledge with constant request leakage, on served requests	6.35, $\leq 2^{-154}$, at the counts of the next row; whether a request is served is not hidden
zero knowledge with request leakage (6.34), at every request, against any verifier	6.37, $< 2^{-154}$; one copy of the committed table, 688,127 replies, 2^{64} hash queries, none before the commitment; one bit per request is revealed
$\mathsf{FlashProve}_Z$ *with the Lean provers, before compilation (Section 6.5); one request to a run; an admissible configuration (3.8) with* $\mathsf{Chk}(Z) = 1$ *(4.14)*	
round-by-round soundness	6.20, 6.21: each round $\leq 2^{-128}$, proof-of-work bits included; one whole run, deterministic prover, $< 2^{-79}$ over the challenges that pass proof of work
completeness	6.23, on served requests, $\leq 2^{-41}$ over the prover's coins and the challenges that pass proof of work
honest-verifier zero knowledge	6.27, $\leq 2^{-135}$ over the life of one commitment; one bit per request is not hidden

- *Batches and policies.* A proof made in a batch, which V_Z verifies singly, is covered at $\varepsilon(q)$, and N proofs sent at once at $N\varepsilon(q)$ (Corollary 5.10). At every admissible Z fixed before the oracle is sampled, with the error $\varepsilon(q)$, an accepted output meets the policy of Z under M^* (Theorem 5.11 and Remark 5.12). For a clipped-gap policy (Definition 3.4), under which a stock engine's replies would be checked, this says only that an accepted output's token costs meet the policy's limits at the extracted weights; it does not prove that such an output is the model's own.
- *A verifier that runs.* An *executable form*, a Lean term that can be evaluated, equals V_Z at every hash function with 32-byte values (Theorem 5.5), so the bound $\varepsilon(q)$ holds for its own decision when its hash function is a random oracle (Corollary B.11). It runs as the command `lean-verify`; its file parser, and Lean's evaluator, compiler and runtime that execute it, are trusted, not proved (Remark 5.9 and Remark B.10).
- *The main theorem's proof.* Steps 4 to 9 of V_Z are the output of the Lean library's compiler at three interactive oracle proofs (Remark B.7, Which compiler); the arithmetization of the whole integer model, the lemma on that compiler (Lemma A.15) and the coding theory below the Johnson radius (Lemma 5.13 and Lemma A.23; the first through files adapted from ArkLib and Mathlib, Appendix E.4) all lie inside the proof. As printed, the theorem joins two Lean theorems (Section 5, head).
- *The check.* Lean's kernel accepts the proofs, with the provisos of Section 1.4 and Remark B.17, and Lean prints no axiom beyond its three standard ones for the theorems that `lean/CHECK-RESULTS.md` names. An independent proof checker has accepted the soundness theorem on `verify`, `sound_of_verify_at` (the form of Theorem 5.1 that leaves the extractor open: Section 5, head), with every declaration beneath it, and nothing that does not lie beneath it, in a pass on that theorem's export alone: the pass was made over a copy of the compiled files of the released Lean files' build (that the compiled files it read were that build's is known from a report alone) (Remark B.17, Checkers).

The programs. The *released programs* are `cp_commit`, `cp_prove` and `cp_verify` (the committer, the prover and the C++ verifier) as the release's sources build them; the *timed runs* are among the runs that are told as runs on them: the programs that made these runs were built from sources that differ from the released ones in comments, in the names of identifiers and files, in quoted texts that hold such a name and in the

wording of messages, and in nothing else (Appendix H.1, "Which programs ran"). A dagger (†) marks a result not on the released programs, of any other programs, which is not shown to hold for the released programs (Section 7.1). Most of these other programs are built from *other sources*: source text of this project's programs that differs from the released text in more than the ways that Appendix H.1, "Which programs ran", lists. In the timed runs (Appendix H.1) the C++ verifier (Section 7.1) accepted every proof (Section 7.2). In no reported test did the C++ verifier program accept an output that violates the run's own policy (Table 7 and Table 4); most of these tests ran not on the released programs (Section 7.1).

- *Honest proofs.* The executable form accepted (by `lean-verify`; four by a separate build of the form alone) every honest proof of Section 7.3, at the size of Z_0, on synthetic weights: 41 of the timed runs' own as reported, at exact match; and, by other provers†, 363 and 18 at exact match and, where an acceptance says less, 1 and 4 under a clipped-gap policy: the first of each pair by one prover built from other sources, the second by other provers than that one (Section 7.3). One proof made at the default configuration's size, on synthetic weights, by the programs that the release's sources build, as reported, was accepted by `cp_verify` and, through `lean-verify`, by the executable form (Appendix H.1).
- *False claims.* The released verifier rejected all 57 proof files that malicious provers made at the default configuration's size, at exact match and under one commitment, on synthetic weights, and that were put to it, 24 of them claiming a false output (Section 7.6); in the timed runs, on the public weights, the programs rejected 240 of 240 altered copies of honest files and 9 of 9 altered calls. `lean-verify` and a separate build of the executable form together rejected all 26 proofs† of claims false by the run's own policy, all from one test program, each rejected where the proof file is read (Section 7.3).
- *Weight hiding*, the programs' goal (Definition H.1). We judge that weight hiding likely holds at the default configuration, in honest runs, if the provider keeps its operating conditions and the judgement's own conditions (listed in `docs/limits.md`, not in this paper) hold. That judgement rests on the design, what is proved in Lean for the protocol, code review and tests; it is not a proof, and it does not cover a real model's weights (Section 7.5).

Measurements. The figures are of those timed runs (Appendix H.1) that used a third party's public, mostly 4-bit weights: on one H100, at the default configuration as fitted to the weights (Section 3.3), at exact match, on windows of 1,024 tokens, with masking on (read off the runs' own summaries, not tested: Appendix H.2); each proof checks one output token. Rates count prefill tokens; ± separates the mean and the sample standard deviation (Section 7.2).

- In batches of 24 requests the steady rate is 111.23 ± 0.92 tokens/second (n = 3), from 110.20 to 111.99; counted from the program's start, over five batches, it is 103.60 ± 0.84. The ± 0.92 is the standard deviation of three runs made side by side: three runs of the same kind on other programs, on another day on other machines, had a standard deviation of 3.9 (Limitation G.8). With one request in a batch the steady rate is 17.41 ± 0.15 tokens/second (n = 3). These rates are counted up to the moment at which the prover hands out its proofs; it runs no verifier, and a serving script's check comes on top (Section 7.2).
- A proof file has 18.74 to 18.81 MB; a client also downloads, once, the key file and the one copy's file, 6.24 MB together; the C++ verifier checks a proof in 21.14 ± 0.03 seconds called with its option for one thread, on the prover's machine (n = 3).
- Per GPU, proving alone takes 93.0 times (92.2 to 94.1 over all pairs of runs) the GPU time of stock vLLM for prefill at the highest load tried ($n = 3$ prover runs, 8 engine runs; calculated; FP8 checkpoint, eight H100s). The multiple omits committing and serving and could be higher (Section 7.8).

For its own integer model of Llama 4 Maverick, on other hardware and for a different statement, VerInf's README [VerInf] reports 14.3 h for one proof on 1,000 tokens and a proof of 93.6 GB (Table 8); each proof of the timed runs on the public weights checks one output token.

Credit. The definition of the primitive, the plan of the protocol (an offline phase, a phase for each batch, and link proofs between committed tables for the same weights), its zero-knowledge proximity tests [CFW26], and the step beyond unique decoding with out-of-domain samples, including the sample taken when a table is

committed, are due to Alessandro Chiesa, Giacomo Fenzi and Guy Weissenberg (personal communication). Table 11 lists further parts that draw on their work.

1.4 Limits and open problems

The release lists the gaps that we know of, and no attack on the protocol is known to us at any gap while its conditions are kept, the opening budget among them (Limitation 8.4, Opening budget). Almost no attack was sought: that none is described is not a finding of safety.

At least four things stand between the theorems and the measured runs. Exact match requires that the integer model itself be served: of the 96 honest replies scored from each of two stock engines none would pass it† (Section 7.4). The runs prove batches, with masks derived from a seed, which completeness and zero knowledge do not cover (Remark 6.42 and Remark G.15). Of the timed runs' proofs only 41, all from the runs on synthetic weights, have been put to the Lean verifier, over the released Lean files; it accepted each (Section 7.3). That the programs run Lean's protocol is tested, not proved, and differences are known (Limitation 8.2).

What the theorems do not say.

- *The programs.* The theorems concern Lean terms: V_Z, its extractor, and the Lean provers, which are definitions, not programs that run (Remark 6.43); the paper does not print the provers, and no statement compares two of the three provers with the honest provers that it describes (Definition 6.14 and Remark 6.16). Nothing is proved about the released programs, and no security level is claimed for them (Section 8.1).
- *The model.* That a second hash function is fixed, and the oracle is taken as independent of it, is an assumption that is neither tested nor proved (Assumption 4.11 and Remark 5.3), nothing is claimed against a quantum adversary (Limitation G.6), and Fiat–Shamir attacks on protocols of this kind have not been assessed (Limitation G.5).
- *The configuration.* Exact match is a hypothesis that V_Z does not test, and the provider's scale file, with five numbers of the parameter file (Definition 3.7), fixes which integer model is meant, and the scale file can disable any chosen blocks of it: the client must check these, and that y is as long as expected, since any prefix of the model's output is a greedy output (Remark 5.3, Limitation 8.6, and Limitation G.4). The provider, not the verifier, chooses the configuration, and a commitment does not say whose weights it holds (Limitation 8.5 and Limitation 8.7). A choice of the configuration after querying the oracle, which is open to every provider, is not covered (Remark 5.3 and Limitation 8.5). For proofs sent one after another no bound is stated (Remark B.13).
- *Trained weights.* No theorem says that a run on a trained model's weights meets its hypotheses, and no proof for such weights has been put to the executable form. The measured runs, on the public weights, are at a configuration fitted to those weights (Section 3.3); the parameter checks were not evaluated at it (Remark D.5). Nor are such weights shown to meet the model's range conditions, which limits what can be proved of such a model, not soundness (Remark 3.3 and Remark D.5).
- *The integer model.* How far the integer model and the floating-point model differ is stated by no theorem: on the public weights its greedy token differs from that of the 16-bit weights, evaluated in float32, at 6.81 % of token positions on 216 texts and at 9.41 % on the 24 natural passages among them (a Python implementation†, no proofs), and its task quality has not been measured (Section 7.7 and Limitation G.9). All texts are built on WikiText-2 [MXBS17], on which the scales were fitted.
- *Completeness.* It covers served requests alone, not batches, and that some request is served is shown at constructed weights. Where the model has no result, or another result than the claimed output, nothing is stated: the Lean prover returns the empty proof, which proves nothing and which a prover can also return at a request that it could serve (Remark 6.30; Limitation 8.9). A gap in completeness costs a rejection, with the provisos of Limitation 8.9.
- *Zero knowledge.* Batches, a further commitment, after compilation a second copy of the committed table, clipped-gap policies, masks derived from a seed, a verifier's hash queries before the commitment and

the C++ programs are not covered, and one bit per request, which depends on the weights, is revealed (Section 6.5). Over many requests the bits tell what the model answers and where it has no result, and no statement bounds what many bits reveal together (Remark 6.44). The simulators' running time is not modelled, and nothing is said of timing, or of what the files of Z tell of the weights (Remark 6.45), or of what the requests tell: each served request holds the model's own output (Remark 6.36); the released tools compute the scale file and lines of the parameter file from the weights, and no statement bounds what they reveal (Remark 6.1).

- *Clipped-gap policies.* A clipped-gap policy bounds a sum and a count, not each token (Limitation 8.8). As in every commit-and-prove argument, the committer chooses the weights. By itself, Theorem 5.11 therefore excludes nothing against a committer who commits to weights under which every output meets the policy (Remark 5.12). Some policies of admissible configurations exclude no token (Proposition B.1). V_Z does not reject a configuration with such a policy, and the C++ verifier rejects some such policies but not every one: no verifier sees the model's logits (Remark B.2).
- *Paper and Lean.* This paper gives V_Z in outline, and that its words match the Lean terms is checked by AI reading, not by Lean, except for the policy's three numbers (Remark D.10 and Remark 5.3).
- *How the proofs are checked.* Over a build of the released Lean files the proofs of completeness, zero knowledge and the theorem on the commitment have been checked by Lean's kernel alone (Remark 6.48); a whole run of the library's full check over all but one of the released Lean files (Remark B.17) passed at every step; its last line says that no theorem got a verdict on binders or on `Classical.choice` (the whole line, and what the check does not cover: `lean/CHECK-RESULTS.md`; Remark B.17). The pinned version of Lean, 4.29.1, is older than the published fixes (from version 4.32.1 on) of soundness bugs in Lean's kernel and runtime and in the big-number library (GMP) that its Linux builds link (Remark B.17). A re-checker, `leanchecker`, which is not the independent checker, stopped on 23 modules of a separate Lean library (counted over Lean files that differ from the released ones), which Lean's kernel accepts at a thread stack larger than the default; by import lines at least one lies beneath each theorem of Section 6.5 on completeness, zero knowledge and the commitment and beneath Proposition 6.17, part 1, none beneath the soundness theorems of Section 5 (Remark 5.7; the finer finding, by a walk over the constants: Remark 6.48).

What is tested, not proved.

- *That the C++ programs run Lean's protocol* (Limitation 8.2). In tests the two verifiers, from whichever programs (Table 4 says which), returned the same decision on every proof file that both judged. Of four matched alterations of one honest proof†, each of which left its claim true, they agreed on one and disagreed on three (these four are not common files: the C++ verifier read the altered raw proof, the form the parsed file altered at the matching place), where the C++ verifier was the stricter. V_Z has sixteen tests that the C++ verifier, which clients run by default, lacks (`lean/GAPS.md`, gap 25), and the one comparison of hash calls matched 881,280 of 1,166,983, a gap that the release rates as heavy.
- *That V_Z accepts the programs' proofs.* None of the 363, 18, 1 and 4 is shown to be the released programs' own, and the 363 are of little variety; the 41 are the timed runs' own, all on synthetic weights. All but those put to it in the reference run and the 41 of the timed runs (Appendix H.3) were judged over Lean files that differ from the released ones or are not shown to be these (`lean/GAPS.md`, gap 26).
- *That the C++ verifier is sound.* The tests are few. Test programs that bypass the verifier program and call the library beneath it directly, against the library's stated contract, did have false claims accepted, by at least two ways; that the verifier program itself never calls it so rests on AI readings of source text that differs in places from the released one, not on a test (Section 7.6). None of these ways went through the verifier program, so a client should check proofs with that program, not through the library.
- *That the programs hide the weights.* Of the judgement's five tests two have run on the released programs. The tests ran at the reduced configuration (Section 3.3), on synthetic weights; only one of the five has also run on a real model's weights, once on the released programs and three times on other programs†. None searches for a leak: they test that the masks are there and that a loss is noticed (Table 6).

Of the injected losses[†] (Table 6), 8 of 8 (none a loss of a single pad) stopped the committer, 130 of 130 in GPU memory stopped, and 263 of 263 riskier losses stopped. Losses were unnoticed in three places: 30 of 165 injected losses of random values in CPU memory; 9 of 9 at samples that no self-check covers; and, with one self-check disabled, at a parameter file that counted no proof of work, a commitment without masks, which passed every other check. Neither the programs nor the judgement has had an independent review.

What is not covered. Openings beyond the budget, or a lost mask, give away exact linear equations, unnoticed (Limitation 8.4, Limitation G.13, and Limitation G.14). Weight hiding rests on operating conditions of the provider, most of which no program can check, and the shipped vLLM integration breaks at least one (Limitation G.16). Section 8.1 and Appendix G state further limits, Section 8.2 the open problems.

2 Preliminaries and definitions

2.1 The random-oracle model

Definition 2.1 (Random oracle, query bound, trace). A *hash function* is a function $H\colon \{0,1\}^* \to \{0,1\}^\lambda$, with $\lambda = 256$. The *random oracle* is the hash function with independent uniform values at distinct inputs.

An *oracle algorithm* is a finite rooted tree whose leaves are labelled by outputs. An inner node is a *query node*, labelled by a string $\mathsf{bs} \in \{0,1\}^*$, with one child for each element of $\{0,1\}^\lambda$, or a *sampling node*. A run under H descends from the root, at a query node to the child $H(\mathsf{bs})$ and at a sampling node to a uniformly random child, and outputs the leaf's label. Its *query–answer trace* tr lists the pairs $(\mathsf{bs}, H(\mathsf{bs}))$ of its query nodes, in order.

A tree is *deterministic* if it has no sampling node, and a *q-query algorithm* if no path has more than q query nodes. An algorithm with inputs is a family of trees, and has either property if every tree has.

2.2 Codes and mutual correlated agreement

Definition 2.2 (Mutual correlated agreement). Let $\mathbb{K}$ be a finite field, D a finite set of *positions*, $\mathcal{C} \subseteq \mathbb{K}^D$ a linear code, $k \geq 2$ an integer, and $\mathcal{G}$ a distribution on $\mathbb{K}^k$, the *generator*. A word $g \in \mathbb{K}^D$ *agrees with* $\mathcal{C}$ *on* a set $S \subseteq D$ if g coincides on S with some codeword. For $\delta \in [0,1)$, the *mutual correlated agreement (MCA) error* $\varepsilon_{\mathrm{mca}}(\mathcal{C}, \mathcal{G}, \delta)$ is the maximum, over all words $g_1, \ldots, g_k \in \mathbb{K}^D$, of

$$\Pr_{(\mathsf{co}_1,\ldots,\mathsf{co}_k)\leftarrow\mathcal{G}}\Big[\exists\, S \subseteq D,\ |S| \geq (1-\delta)\,|D| \ :\ \textstyle\sum_j \mathsf{co}_j\, g_j \text{ agrees with } \mathcal{C} \text{ on } S \text{ and some } g_j \text{ does not}\Big].$$

The code $\mathcal{C}$ has *mutual correlated agreement* at the proximity parameter δ with error ε, under $\mathcal{G}$, if $\varepsilon_{\mathrm{mca}}(\mathcal{C}, \mathcal{G}, \delta) \leq \varepsilon$. The *Johnson radius* of a code of relative minimum distance Δ is $1 - \sqrt{1-\Delta}$.

Mutual correlated agreement was introduced in [ACFY24b, Definition 4.9], for proximity generators; it strengthens the correlated agreement of [BCIKS20, Section 1.3]. Definition 2.2 follows [CFW26, version of February 25, 2026, Definition 3.14], which defines the MCA error as a maximum over the words. In [BCGM25, Definitions 3.10 and 3.14] a generator is a function of a random seed; here it is the distribution of the coefficients. The Johnson radius is as in [ABF26, Section 1].

2.3 Interactive oracle proofs, compilation, proof of work

Definition 2.3 (Interactive oracle proof, batch). An *interactive oracle proof (IOP) with external oracle strings* has an input, the *IOP instance*, and a fixed number of rounds. Its *external oracle strings* are arbitrary, fixed before the first round, and not sent. In each round the prover sends an *oracle part* (a string) and a *plain part*; the verifier answers with a uniform *challenge* from a finite set. Finally the verifier accepts or rejects; it reads the IOP instance, the plain parts, the challenges, and the positions of the oracle strings that these determine.

A *batch IOP* for at most R IOP instances marks each round as *shared* or *own*: a shared round has one challenge for all of them, an own round one for each. Each prover message has a shared component and one

own component per instance. The verifier decides on each from its *local transcript*: the shared components, its own components, and its challenges.

Interactive oracle proofs were introduced in [BCS16, Section 4], which states that [RRR16] independently introduces the same notion, as probabilistically checkable interactive proofs.

Definition 2.4 (Compilation (BCS transformation)). The *compilation* $\mathsf{Compile}(Q, H)$ of an IOP Q (Definition 2.3) under a hash function H is the following non-interactive argument; its queries to H are of three kinds (leaf, inner node, Fiat–Shamir), coded injectively with disjoint ranges.

1. Each oracle string is committed by the root of a Merkle tree whose leaves hash a symbol and a random salt. The external roots are an input of the verifier.
2. The first Fiat–Shamir query holds the external roots, a salt and the IOP instance. The query of round m holds a chaining value from the preceding answer and the round's roots and plain part; its answer gives the round's challenge and the next chaining value.
3. The proof string consists of that salt, the other roots, the plain parts and, for each position read, symbol, salt and authentication path.
4. The verifier recomputes the challenges, checks the paths, and decides as the verifier of Q does.

Definition 2.5 (Proof of work (grinding)). Round m of an IOP (Definition 2.3) has κ_m bits of *proof of work* if its plain part contains a *nonce*, a component that the verifier does not read, and the verifier rejects unless κ_m designated bits of its challenge are 0. The compiled prover (Definition 2.4) tries values of the nonce, up to a fixed number, until these bits are 0.

2.4 Commit-and-prove arguments and their properties

Definition 2.6 (Indexed relation, commit-and-prove argument). A *token* is a natural number. An *instance* is a pair (x, y) of finite lists of tokens, the *prompt* and the *output*. An *indexed relation* consists of a set $\mathcal{I}$ of *indices*, a set Rel of triples (M, x, y) with $M \in \mathcal{I}$ (there is no witness), and a function pp on $\mathcal{I}$, the *public parameters of the index*. The *language* of M is $\mathsf{Lang}(M) = \{(x, y) : (M, x, y) \in \mathsf{Rel}\}$, and $\mathsf{Lang}(\perp) = \emptyset$.

A *commit-and-prove argument* for this relation is a tuple $\Pi = (\mathsf{Rnd}, \mathsf{Commit}, P, V)$ of oracle algorithms (Definition 2.1):

1. Rnd^H outputs commitment randomness r;
2. $\mathsf{Commit}^H(M, \mathsf{r})$ is deterministic and outputs a *commitment* $c \in \{0,1\}^\lambda$;
3. $P^H(M, \mathsf{r}, x, y)$ outputs a *proof* π;
4. $V^H(c, x, y, \pi)$ is deterministic and outputs 1 (accept) or 0 (reject).

The *honest committer* on M samples $\mathsf{r} \leftarrow \mathsf{Rnd}^H$, publishes $c = \mathsf{Commit}^H(M, \mathsf{r})$ and keeps (M, r) for the *honest prover* P. In the *general form*, instances and commitments range over arbitrary sets, and P takes a list of instances.

Indexed relations are as in [CHMMVW19, Section 3.1], where a triple holds an index, an instance and a witness.

Definition 2.7 (Index-binding knowledge soundness). Let Π be a commit-and-prove argument with verifier V for an indexed relation with index set $\mathcal{I}$ and languages $\mathsf{Lang}(\cdot)$ (Definition 2.6), and let $\varepsilon\colon \mathbb{N} \to [0, \infty]$. An *extractor* is a function Ext from pairs of a commitment and a query–answer trace to $\mathcal{I} \cup \{\perp\}$. For oracle algorithms $\mathcal{A}_C$, the *adversarial committer*, and $\mathcal{A}_P$, the *adversarial prover*, the experiment is:

1. sample the random oracle H;
2. run $(c, \mathsf{st}) \leftarrow \mathcal{A}_C^H$, with query–answer trace tr_C;
3. set $M^* = \mathsf{Ext}(c, \mathsf{tr}_C)$;
4. run $(x, y, \pi) \leftarrow \mathcal{A}_P^H(\mathsf{st})$;

5. set $\mathsf{acc} = V^H(c, x, y, \pi)$.

The argument Π is *index-binding knowledge sound* for the relation, with error ε, if there is an extractor Ext such that, for all $q_C, q_P \in \mathbb{N}$, every q_C-query committer $\mathcal{A}_C$ and every q_P-query prover $\mathcal{A}_P$,

$$\Pr\big[\,\mathsf{acc} = 1 \text{ and } (x, y) \notin \mathsf{Lang}(M^*)\,\big] \;\le\; \varepsilon(q_C + q_P).$$

Remark 2.8 (Names of the properties). The terms index-binding knowledge soundness, correctness of extraction and index privacy are due to [CFWpc], and Definition 2.6, Definition 2.7, Definition 2.12, Definition 2.13, and Definition 2.14 are adapted from that work; the name and the requirement of Definition 2.9 are taken from it too, and its experiment and error are this paper's. Here the word *knowledge* refers to the index: the relation has no witness, and the extractor returns an index or $\perp$. The names of the roles, index committer, index verification key and index extractor, are from the same work. Table 12 lists uses of the same or like names in the literature, of other objects. From here on an index committer is called a committer, and an index extractor an extractor. The commitment is the 256-bit value that has the role of the index verification key. Such a key may be all that a committer publishes; the commitment is narrower: in the theorem it is these 256 bits alone.

Definition 2.9 (Correctness of extraction). Let $\Pi = (\mathsf{Rnd}, \mathsf{Commit}, P, V)$ be a commit-and-prove argument with index set $\mathcal{I}$ (Definition 2.6), Ext an extractor (Definition 2.7), $\varepsilon_{\mathrm{ext}} \in [0, 1]$, and H the random oracle (Definition 2.1). Π has *correctness of extraction* for Ext with error $\varepsilon_{\mathrm{ext}}$ if, for every $M \in \mathcal{I}$ fixed before H is sampled,

$$\Pr\big[\,\mathsf{Ext}(c, \mathsf{tr}) \neq M \;:\; \mathsf{r} \leftarrow \mathsf{Rnd}^H,\; c = \mathsf{Commit}^H(M, \mathsf{r})\,\big] \;\le\; \varepsilon_{\mathrm{ext}},$$

where tr is the query–answer trace of this run of Rnd and Commit, and the probability is over H and the random choices of Rnd.

Definition 2.10 (Extractability of the commitment, no two openings). Let Π, Ext and H be as in Definition 2.9 and $\varepsilon_{\mathrm{op}} \colon \mathbb{N} \to [0, \infty]$. A pair (M, r) *opens* a commitment c if $\mathsf{Commit}^H(M, \mathsf{r}) = c$; the queries of this test are not counted in q_C, q_O or q. Probabilities are over H and the algorithms' random choices.

1. Π has *extractability of the commitment* (also called opening-form binding) for Ext with error $\varepsilon_{\mathrm{op}}$ if, for every q_C-query algorithm $\mathcal{A}_C$ that outputs a commitment c and a state, with trace tr_C, and every q_O-query algorithm that outputs a pair (M, r) given this state,

$$\Pr\big[\,(M, \mathsf{r}) \text{ opens } c \text{ and } \mathsf{Ext}(c, \mathsf{tr}_C) \neq M\,\big] \;\le\; \varepsilon_{\mathrm{op}}(q_C + q_O).$$

2. Π has *no two openings* with error $\varepsilon_{\mathrm{op}}$ if, for every q-query algorithm that outputs c, (M_1, r_1) and (M_2, r_2),

$$\Pr\big[\,\text{both pairs open } c \text{ and } M_1 \neq M_2\,\big] \;\le\; \varepsilon_{\mathrm{op}}(q).$$

Remark 2.11 (Standard names). No two openings is the standard binding property of a commitment scheme [CFQ19, Definition 2.1], for $(\mathsf{Rnd}, \mathsf{Commit})$. Extractability of the commitment is the property that [BCS16, Section 3.1] calls extractability; correctness of extraction is its form for the honest committer's own input.

Definition 2.12 (Completeness on a set of triples). Let $\Pi = (\mathsf{Rnd}, \mathsf{Commit}, P, V)$ be a commit-and-prove argument (Definition 2.6) for the indexed relation Rel, S a set of triples (M, x, y), and $\alpha \in [0, 1]$. The argument Π is *complete on* S with error α if, for every $(M, x, y) \in S \cap \mathsf{Rel}$,

$$\Pr\left[\, V^H(c, x, y, \pi) = 0 \;:\; \begin{array}{l} H \text{ sampled as the random oracle}, \quad \mathsf{r} \leftarrow \mathsf{Rnd}^H, \\ c = \mathsf{Commit}^H(M, \mathsf{r}), \quad \pi \leftarrow P^H(M, \mathsf{r}, x, y) \end{array} \,\right] \;\le\; \alpha.$$

It is *complete* with error α if it is complete on the set of all triples.

In what follows SD is the statistical distance.

Definition 2.13 (Real and ideal experiment). Let Π be an argument in the general form (Definition 2.6), with arbitrary indices and one or several independent random oracles H. A *batch* is a list of instances, each marked *revealed* or *hidden*; P maps it to a proof or $\perp$ for each instance. For an oracle algorithm V^*, a function Env, the *environment*, and $N \in \mathbb{N}$, the *real experiment* $\mathsf{Real}_\Pi(M, \mathsf{Env}, V^*, N)$ is:

1. sample H; the honest committer, on M, outputs c and keeps $\mathsf{sk} = (M, \mathsf{r})$ (Definition 2.6);
2. run $V^{*\,H}(c)$, which asks for a batch at most N times;
3. each time, set $\mathsf{bt} = \mathsf{Env}(\mathsf{view})$; if $\mathsf{bt} = \bot$, stop;
4. run $P^{H}(\mathsf{sk}, \mathsf{bt})$ and give V^* the revealed instances with their proofs;
5. output view: c, what V^* was given, its query–answer trace and its coins.

In the *ideal experiment* $\mathsf{Ideal}_{\mathsf{Sim}}(M, \mathsf{Env}, V^*, N)$ a stateful Sim replaces H, committer and prover (so the random oracle is programmable). Besides the queries of V^*, which it answers, it is given only the *leakage*: $\mathsf{pp}(M)$, N, and each batch's revealed instances.

Definition 2.14 (Index privacy). Let $\mathcal{S}$ assign to each index M of a set $\mathcal{J}$ a set $\mathcal{S}_M$ of instances, let $R \geq 1$ and $\zeta\colon \mathbb{N}^2 \to [0, \infty]$. An environment is *valid* at M if each of its batches has at most R instances, all in $\mathcal{S}_M$. An argument Π has *index privacy on $\mathcal{S}$ for batches of at most R, with error ζ*, if a simulator Sim exists such that, for all $M \in \mathcal{J}$, all environments Env valid at M, all $q, N \in \mathbb{N}$ and every q-query V^*,

$$\mathsf{SD}\big(\mathsf{Real}_{\Pi}(M, \mathsf{Env}, V^*, N),\ \mathsf{Ideal}_{\mathsf{Sim}}(M, \mathsf{Env}, V^*, N)\big)\ \leq\ \zeta(q, N)$$

(Definition 2.13), and in the ideal experiment equal queries receive equal answers.

What Section 6.5 proves of zero knowledge after compilation is stated for another notion, zero knowledge with request leakage (Definition 6.34), and there in two statements: on the served instances with a constant leakage (Theorem 6.35), and on all instances, those that are not served among them (Definition 6.12): at such an instance the prover returns the empty proof, and the simulator is told, for each instance, whether it is served, which depends on the weights (Theorem 6.37, with its bounds; Remark 6.40). Completeness says nothing at such an instance (Remark 6.30).

3 The relation: model, policies, configuration

The complete definitions of the integer model, the placement rule, the arithmetization and the parameter checks are terms of the Lean development, which Appendix F names. This paper outlines them (Construction D.3 and Construction D.9, Definition 4.1 and Definition 4.14); Remark D.10 (Given in outline) lists what it does not print. Every admissible configuration is Z_0 with its scale file, five of its model parameters and the three numbers of its policy replaced (Definition 3.7 and Definition 3.8); the layout file and all sizes of Table 2 are those of Z_0.

3.1 The integer model

The integer model is the function that one term of the Lean development computes. The release's `docs/model.md` describes it step by step, and Construction D.3 outlines it, without defining it.

Definition 3.1 (Integer model). The *integer model* (Construction D.3) is a transformer in integer arithmetic with *window length* $L = 2^{10}$ and *vocabulary* $\mathcal{V} = \{0, \dots, 202047\}$. Its *weights* M are integer matrices of fixed shapes whose entries lie in fixed intervals that contain 0; their set is $\mathcal{I}$. It rescales by its model parameters μ (Definition 3.6). A *context* is a list $\bar{x}$ of at least one and at most L tokens of $\mathcal{V}$. At valid μ and $M \in \mathcal{I}$, the model is *defined* on $\bar{x}$ if no intermediate value leaves its prescribed interval at any token position (the *range conditions*); it then maps $\bar{x}$ to a vector of *logits* $\mathrm{logit}_M(\bar{x}) \in \mathbb{Z}^{\mathcal{V}}$.

That Lean's model function has a value exactly on such contexts is argued by hand; no Lean theorem states it.

Definition 3.2 (Greedy token, greedy output). Let μ be valid model parameters and $M \in \mathcal{I}$ (Definition 3.1). The *greedy token* after a context $\bar{x}$ on which the model is defined is the least token of largest logit,

$$\mathrm{greedy}_M(\bar{x})\ =\ \min\big\{\, \mathsf{v} \in \mathcal{V}\ :\ \mathrm{logit}_M(\bar{x})[\mathsf{v}] = \max_{\mathsf{v}' \in \mathcal{V}} \mathrm{logit}_M(\bar{x})[\mathsf{v}'] \,\big\}.$$

An output $y = (y_0, y_1, \dots)$ is a *greedy output* of M on a prompt x if, for every $i < |y|$, the list $x \,\|\, (y_0, \dots, y_{i-1})$ is a context on which the model is defined, and y_i is the greedy token after it. The *relation of greedy outputs* at

μ is the indexed relation (Definition 2.6) that consists of these triples (M, x, y), with index set $\mathcal{I}$ and constant pp.

Remark 3.3 (Trained weights). The theorems' weights are arbitrary elements of $\mathcal{I}$, all-zero matrices among them, and the theorems name no public model. No trained model's weights are shown to meet the range conditions, which limits what can be proved of such a model, not soundness. Remark D.5 (No real weights shown) states what was counted and run.

3.2 Acceptance policies

The clipped-gap policies are modelled on Token-DiFR [DiFR], at temperature 0 (Appendix E.2).

Definition 3.4 (Logit gap, acceptance policy, exact match, clipped gap). Let μ, M, logit_M and greedy_M be as in Definition 3.2. The *logit gap* of a token $\mathsf{v} \in \mathcal{V}$ after a context $\bar{x}$ on which the model is defined is

$$\mathrm{gap}_M(\bar{x}, \mathsf{v}) \;=\; \max_{\mathsf{v}' \in \mathcal{V}} \big(\mathrm{logit}_M(\bar{x})[\mathsf{v}'] - \mathrm{logit}_M(\bar{x})[\mathsf{v}] + [\mathsf{v}' < \mathsf{v}]\big).$$

An *acceptance policy* is a triple $\mathsf{pol} = (C, \theta, \mathit{cap})$ of natural numbers: *clip*, *threshold*, *cap*.

Let x and $y = (y_0, y_1, \dots)$ be non-empty lists of tokens of $\mathcal{V}$ with $|x|+|y| \le L+1$. Put $\bar{x}_i = x \,\|\, (y_0, \dots, y_{i-1})$ and $n_{\mathrm{ct}} = \min(|y|, L - |x|)$; the *counted* tokens are $y_0, \dots, y_{n_{\mathrm{ct}}-1}$. The output y is *eligible* on x under M if the model is defined on every $\bar{x}_i$ and, in case $n_{\mathrm{ct}} < |y|$, $y_i = \mathrm{greedy}_M(\bar{x}_i)$ at $i = |y| - 1$. Put $\mathrm{gap}_i = \mathrm{gap}_M(\bar{x}_i, y_i)$; the *cost* of a counted token y_i is $\mathrm{cost}_i = \mathrm{gap}_i + 2^8$. The output *meets* pol if it is eligible,

$$\sum_{i < n_{\mathrm{ct}}} \min(\mathrm{cost}_i,\, C) \;\le\; \theta\, n_{\mathrm{ct}}, \qquad \text{and} \qquad \#\{\, i < n_{\mathrm{ct}} : \mathrm{cost}_i > C \,\} \;\le\; \mathit{cap}.$$

No other pair (x, y) meets a policy. *Exact match* is the policy $(2^8, 2^8, 0)$; every other policy is a *clipped-gap policy*.

Exact match forces every logit gap to 0, and the token after a full window has no cost.

Proposition 3.5 (Exact match and greedy outputs). [machine-checked] *Let μ be valid model parameters, $M \in \mathcal{I}$, and x, y lists of tokens. The output y meets exact match on x under M (Definition 3.4) if and only if y is non-empty and a greedy output of M on x (Definition 3.2). In that case y meets every acceptance policy with $\theta \ge 2^8$ and $C \ge 2^8$.*

Each of the two sentences is a single Lean theorem.

Outline. A logit gap is at least 0, and it is 0 exactly at the greedy token. At exact match, $\mathit{cap} = 0$ forces $\mathrm{cost}_i \le 2^8$, hence $\mathrm{gap}_i = 0$, at every counted token, and an eligible output's token after a full window is greedy. Conversely, costs of 2^8 satisfy both bounds whenever $\theta, C \ge 2^8$.

3.3 Configurations

Definition 3.6 (Model parameters). The integer model has 48 layers, numbered from 0. Its *model parameters* μ are natural numbers: in each layer eight shifts and the triples $(\mathsf{M}_1, \mathsf{K}_1, \epsilon_1)$ and $(\mathsf{M}_2, \mathsf{K}_2, \epsilon_2)$ of its two normalizations; in each odd layer three further shifts; and the triple $(\mathsf{M}_F, \mathsf{K}_F, \epsilon_F)$ of the final normalization. They are *valid* if each lies in its range:

Number	Meaning	Range
k_q, k_k	shifts of queries, keys	$5, \dots, 32$; equal, or both at least 16
k_v, k_g, k_u	shifts of values, gate, up	$1, \dots, 24$
k_o	shift of attention's output	$1, \dots, 19$
k_d	shift of down projection	$10, \dots, 19$
k_s	shift of softmax keys	$4, \dots, 12$; equal in the layers $4j, \dots, 4j+3$
k_r	shift of router (odd layers)	$8, \dots, 24$
k'_g, k'_u	routed expert's gate, up	$1, \dots, 24$
M, K	exponents of a normalization	$24 \le \mathsf{M} \le 47$, $17 \le \mathsf{K} \le 32$, $\mathsf{K} \le \mathsf{M} \le \mathsf{K} + 14$
ϵ	its regularizer	$1 \le \epsilon < 2^{48}$

Definition 3.7 (Configuration, default configuration). A *configuration* is a triple Z of byte strings: the *parameter file*, the *layout file* and the *scale file* (their encoding: Definition D.1). The *default configuration* Z_0 is one fixed configuration (Definition D.1). Its parameter file holds its policy (C, θ, cap), which is exact match (Definition 3.4), and ϵ_1, ϵ_2, ϵ_F, M_F, K_F of its model parameters μ_0 (Definition 3.6); its scale file holds the rest of μ_0; its layout file says where each cell (Definition 4.8) lies; a parameter check tests that each lies where a Lookup (Construction D.6) reads it (Definition 4.14, item 3). For model parameters μ with the same ϵ_1 and ϵ_2 in every layer and an acceptance policy pol, $Z_0[\mu, \mathsf{pol}]$ denotes Z_0 with these eight numbers and the scale file replaced by those of μ and pol. Put $Z_1 = Z_0[\mu_0, (25856, 512, 1)]$.

The release's pages call the three byte strings of a configuration its public files.

Definition 3.8 (Admissible configuration). A configuration Z is an *admissible configuration*, written $\mathsf{Adm}(Z) = 1$, if $Z = Z_0[\mu, \mathsf{pol}]$ (Definition 3.7) for valid model parameters μ (Definition 3.6) with one ϵ_1 and one ϵ_2 for all layers and an acceptance policy $\mathsf{pol} = (C, \theta, cap)$ with

$$256 \le \theta \le C < 2^{32} \qquad \text{and} \qquad cap \le 1{,}024.$$

Then $\mu_Z = \mu$ are the *model parameters of* Z, and $\mathsf{pol}_Z = \mathsf{pol}$ is the *policy of* Z.

The configuration Z_1 is admissible, and Proposition B.3 describes its policy. No theorem ties the shipped files of Z_1 to the Lean term (Remark D.2); five of the honest proofs of Section 7.3 are at these files. Section 7 uses two further kinds of configuration.

- A configuration is *fitted* to weights if its scale file and lines of its parameter file are computed from them (Remark D.5, No real weights shown).
- A *reduced configuration* is one for a synthetic model far smaller than the integer model, for instance with 8 layers and a window of 64 tokens; its parameter file states other sizes than that of Z_0, so it is not an admissible configuration.

3.4 Policies that exclude nothing

Proposition 3.9 (All-zero weights). [machine-checked] *Let μ be valid model parameters (Definition 3.6) and $\mathsf{pol} = (C, \theta, cap)$ an acceptance policy (Definition 3.4) with $\theta, C \ge 257$. Under the all-zero weights, every non-empty output y meets pol on every non-empty prompt x, provided that all tokens lie in $\mathcal{V}$, that $|x| + |y| \le L + 1$, and that the last token of y is 0 if $|x| + |y| = L + 1$.*

Outline. The model is defined on every context. With an all-zero output matrix all logits are 0, so token 0 is the greedy token, every logit gap is 0 or 1, and every cost is at most 257.

Remark 3.10 (What a policy bounds).

1. A clipped-gap policy bounds a sum and a count over the counted tokens of one output, which in a soundness statement the prover chooses; each token is scored after the output's own earlier tokens.
2. A logit gap is measured in the unit of the logits of the weights at hand, so what a policy excludes depends on the weights (Proposition 3.9). The parameter file states that unit, as 2^{-15} nats at every admissible configuration; the model does not read this number, and nothing in the proof ties it to the weights.
3. The bound on the sum has the form of the test of Token-DiFR [DiFR], whose checker runs a model that it knows, pools many outputs' tokens and fits the clip to honest runs. Here each output is bounded alone, and the provider writes the clip, so an output that meets the policy has not thereby passed that test.

3.5 Instances that can be served

Definition 3.11 (Instances that can be served). Let μ be valid model parameters and $M \in \mathcal{I}$ (Definition 3.1). An instance (x, y) with y non-empty and all tokens in $\mathcal{V}$ *can be served* at M if the integer model at μ is defined, for M, on the first $L = 1{,}024$ tokens of $x \,\|\, y \,\|\, 0\,0\cdots$. The set Serv_μ consists of all (M, x, y) such that (x, y) can be served at M and y is a greedy output of M on x (Definition 3.2).

Table 2: Parameters, with their values at Z_0 (Definition 3.7).

Symbol	Parameter	Value
$\mathbb{F}$, p	prime field, its order	$p = 2^{64} - 2^{32} + 1$
$\mathbb{E}$	extension field, for V's samples	$\mathbb{F}[\vartheta]$, $\vartheta^3 = 2$
L; G	length of the window; segments of the model's run (Definition 4.8)	1,024; 25
d; $n = n_0 + v$	A has d matrix rows, 2^n matrix columns; a table 2^v rows	3,076,096; $18 = 7 + 11$
D	positions, $D \subseteq \mathbb{F}$	$\lvert D\rvert = 2^{32}$
ℓ; $\ell_0 = \ell - 3s$	entries of a row's message; plain entries	402,653,184; 402,653,181
t	random coefficients of a row: opening budget	44,040,191
s	OOD answers for each row	1
ρ	rate $(\ell + t)/\lvert D\rvert$	0.1041
e; $\delta = e/\lvert D\rvert$	decoder's radius; proximity parameter	2,852,142,980; 0.6640
$\nu_{\text{off}}, \nu_{\text{first}}, \nu_{\text{copy}}, \nu_{\text{on}}$	maximum number of opened columns (Construction 4.13)	$62, 62, 62, 64$
δ_{seg}	proximity parameter of a segment's first proximity test (Construction D.9); its positions	0.2901; 219
$\kappa_{\text{req}}, \kappa_{\text{bat}}, \kappa_{\text{off}}$	bits of proof of work: a request's rounds; shared rounds; offline and link proofs	$20, 28, 32$
	Range test (Construction D.6): columns of its matrix Θ; range of an entry of the product, sent as two digits	128; $[-2^{17}, 2^{17})$
R	largest batch	64
K	copies: at most, by $\nu_{\text{off}} + K\nu_{\text{first}} \le t$	710,324
N_j	batches on copy j: at most, by $\nu_{\text{copy}} + N_j\nu_{\text{on}} \le t$	688,127

The parameter file states the sizes of the tables and the bits of proof of work; every other value is calculated from the parameter file and taken from no run; the rate ρ is rounded up, proximity parameters down (Remark D.14, How the parameters are obtained).

Proposition 3.12 (Non-vacuity of the instances that can be served). [machine-checked]

(a) Valid model parameters exist, and for all valid μ the set Serv_μ (Definition 3.11) is not empty.

(b) There are valid model parameters μ and weights $M \in \mathcal{I}$ at which no instance can be served.

Remark 3.13 (Which instances can be served). Serv_μ leaves out the empty output, which is always a greedy output; whether each of the other conditions of Definition 3.11 is needed for the honest proof to be accepted is not known. At some valid model parameters and some weights no instance can be served (Proposition 3.12(b)). Instances are proved to lie in Serv_μ at constructed weights and, at the model parameters of Z_0, at all valid weights whose layers' matrices have every entry in $-800\,..\,800$: there an instance can be served exactly if its tokens are in the vocabulary and its output is not empty. No statement says whether a real model's weights lie within that bound, and no instance is proved to lie in Serv_μ at a real model's weights.

4 The protocol

Section 1.2 gives an overview of the protocol. This section gives its form, and Appendix D its rounds; what of V_Z the paper does not print is listed in Remark D.10. Sources are named where each idea is introduced; Table 11 lists the parts of the protocol and their sources.

4.1 Parameters

Table 2 introduces the symbols for the sizes of the protocol. The parameter file states or determines them; every admissible configuration has the values of Z_0.

4.2 Encoding and commitment

The weights are encoded once, with a masked code, a zero-knowledge Reed–Solomon encoding ([CFWpc]; [CY24, Section 32.5]; [CFW26]). In the literature's terms, a committed table (Definition 4.2) is a codeword of a

2^v-fold interleaved Reed–Solomon code, with the columns as its symbols. The *masks* are the pads and random coefficients of the tables and the mask rows of the provers' oracle messages (Construction D.6); Section 7.1 defines the mode *masking on* of a run. Any t columns of a table sampled with fresh randomness, together with its OOD answers at an OOD seed chosen before the table is sampled, are distributed independently of the matrix (Theorem 6.3; Remark 6.4); $t+1$ columns are not (Proposition 6.5). This is the reason for the opening budget t (Definition 4.2), a limit on the number of columns of one table that all proofs on it open in total (Definition C.6; Remark C.7).

Definition 4.1 (Digit matrix). Each weight of M (Definition 3.1) is written in signed digits to the base 2^8: four digits for an entry of the embedding matrix, two for every other weight. A public placement rule (Remark D.10, Given in outline), fixed by the parameter file of Z, assigns to each digit of each weight one entry of a matrix with d *matrix rows* and 2^n *matrix columns*. An integer matrix A of this shape is *in range* if every entry lies in $[-2^7, 2^7)$ and

$$b_0 + 2^8\, b_1 \in [-2^{12}, 2^{12})$$

for the entries b_0, b_1 in each matrix column of each pair of matrix rows that the rule assigns to the low and the high digits of two-digit weights. For A in range, $\mathsf{wt}(A) \in \mathcal{I}$ denotes the weights that the rule reads off A. A *digit matrix* of M is a matrix A in range with $\mathsf{wt}(A) = M$; it need not be unique.

Definition 4.2 (Masked encoding, committed table, opening budget). Let $D \subseteq \mathbb{F}$ with $|D| \geq \ell + t$. The *masked encoding* of a message $f \in \mathbb{F}^\ell$ with random coefficients $\tau \in \mathbb{F}^t$ is the Reed–Solomon codeword

$$\mathsf{Enc}(f;\tau) \;=\; \Big(\sum_{j<t} \tau_j\, \phi^j \;+\; \sum_{j<\ell} f_j\, \phi^{t+j}\Big)_{\phi \in D}.$$

Let A be a matrix over $\mathbb{F}$, or an integer matrix, of the shape of Definition 4.1. Let $n = n_0 + v$, let s be the number of OOD answers for each row (Definition 4.3), and let $d\,2^{n_0} \leq \ell_0 = \ell - 3s$. The *message of row* $r < 2^v$ is $f_r \in \mathbb{F}^\ell$ with

$$f_r[\,j + 2^{n_0}\iota\,] \;=\; A[\iota][\,j + 2^{n_0} r\,] \bmod p \qquad (j < 2^{n_0},\ \iota < d),$$

with 0 at its other entries below ℓ_0, and with uniform elements of $\mathbb{F}$, the *pads*, at its last $3s$ entries. The entries below ℓ_0 are *plain*. A *committed table* for A is $\mathbf{T} = (\mathsf{Enc}(f_r;\tau_r))_{r<2^v}$ with independent uniform τ_r. Its *column* at the *position* $\phi \in D$ is $\mathbf{T}[\phi] \in \mathbb{F}^{2^v}$. Its *opening budget* is t.

Definition 4.3 (Out-of-domain seed, answers, sample). Let $\mathbb{E} = \mathbb{F}[\vartheta]$ be of degree 3 over $\mathbb{F}$, let $\mathrm{eq}(z,j) = \prod_k \big(z_k j_k + (1-z_k)(1-j_k)\big)$, where j_k is bit k of j, the lowest first, and let $(f_r)_{r<2^v}$ be row messages of length ℓ, over $\mathbb{F}$ or over $\mathbb{E}$ (Definition 4.2). An *out-of-domain (OOD) seed* is a list $\varsigma = (z_i, \beta_i)_{i<s}$ of points $z_i \in \mathbb{E}^{\lceil \log_2 \ell \rceil}$ and factors $\beta_i \in \mathbb{E} \setminus \{0\}$. The *OOD answers* of $(f_r)_r$ at ς are $\mathbf{a} = (a_{r,i})_{r<2^v,\, i<s}$ with

$$a_{r,i} \;=\; \sum_{j<\ell_0} \mathrm{eq}(z_i, j)\, f_r[j] \;+\; \beta_i\, \mathsf{pad}_{r,i}, \qquad \mathsf{pad}_{r,i} \;=\; \sum_{k<3} \vartheta^k f_r[\ell_0 + 3i + k].$$

The OOD answers of a committed table are those of its row messages. The pair $\sigma = (\varsigma, \mathbf{a})$ is the table's *OOD sample* at ς.

Remark 4.4 (Standard terms). The masked encoding is the zero-knowledge encoding of [CY24, Section 32.5] and [CFW26]. The opening budget is the *query bound* of bounded-query zero knowledge [BCRSVW18, Definition 4.4]; this paper keeps the word query for queries to H. In [ACFY24b] and [CFW26] an out-of-domain sample is the verifier's point alone; here it is the OOD seed with the committer's answers. Each answer is a masked sum of the weights.

Definition 4.5 (List, collision, decoder). Let $\mathbf{T}^* \in (\mathbb{F}^{2^v})^D$ be a word and e the *radius*, a number of positions. A *table of codewords* is $(\mathsf{Enc}(f_r;\tau_r))_{r<2^v}$ for any $f_r \in \mathbb{E}^\ell$ and $\tau_r \in \mathbb{E}^t$. The *list* $\Lambda(\mathbf{T}^*)$ consists of the tables of codewords that differ from $\mathbf{T}^*$ in at most e columns. The word has a *collision* at an OOD seed ς, written $\mathsf{Coll}(\mathbf{T}^*, \varsigma)$, if two members of $\Lambda(\mathbf{T}^*)$ with different messages $(f_r)_r$ have the same OOD answers at ς (Definition 4.3). The *decoder* maps $\mathbf{T}^*$ and $\sigma = (\varsigma, \mathbf{a})$ to a matrix or to $\perp$.

1. If $\mathsf{Coll}(\mathbf{T}^*, \varsigma)$, then $\mathsf{Dec}(\mathbf{T}^*, \sigma) = \perp$.

2. Otherwise let $\mathbf{T}'$ be nearest to $\mathbf{T}^*$ among all tables of codewords with OOD answers $\mathbf{a}$ at ς, ties broken by a fixed order. If $\mathbf{T}' \notin \Lambda(\mathbf{T}^*)$, the value is $\perp$.

3. Otherwise the value is the matrix A in range (Definition 4.1) whose residues modulo p the messages of $\mathbf{T}'$ hold at the places of Definition 4.2, or $\perp$ if there is none.

4.3 Copies and the link proof

Every proof opens columns, hence the provider makes copies of the table with fresh masks, each with a link proof ([CFWpc]; [CFW26]).

Each committed table has an OOD sample ([CFWpc]; the technique: [BGKS20], as in [ACFY24b, Section 4.4]). The mask of an answer, a pad times a sampled nonzero factor (Definition 4.3), follows the private zero-evaders of [CFW26, version of February 25, 2026, Section 9.2]; that only list members with different messages count as a collision (Definition 4.5) follows its Section 9.1.

The OOD samples allow a radius e beyond half the code's minimum distance, where the list of a word can have several members; the sample selects one, up to a small error (Lemma A.25). Definition 4.5 takes list members over $\mathbb{E}$; those of a word over $\mathbb{F}$ are over $\mathbb{F}$, since e is below the minimum distance (proof of Lemma A.25). A word at relative distance δ passes ν sampled positions with probability $(1-\delta)^\nu$, hence a larger δ requires fewer opened columns, which gives shorter proofs and allows more proofs within one opening budget.

Definition 4.6 (First table, copy, link proof). Let A be in range. The *first table* $\mathbf{T}_0$ and the *copies* $\mathbf{T}_1, \dots, \mathbf{T}_K$ are committed tables for A (Definition 4.2) whose pads and random coefficients are mutually independent; σ_j is an OOD sample of $\mathbf{T}_j$. The *relation of the link proof* is the set of all $(\mathbf{T}_0^*, \sigma_0, \mathbf{T}_j^*, \sigma_j)$, two words with a sample each, such that

$$\mathsf{Dec}(\mathbf{T}_0^*, \sigma_0) = \perp \quad \text{or} \quad \mathsf{Coll}(\mathbf{T}_j^*, \varsigma_j) \quad \text{or} \quad \mathsf{Dec}(\mathbf{T}_j^*, \sigma_j) = \mathsf{Dec}(\mathbf{T}_0^*, \sigma_0),$$

with Dec and Coll as in Definition 4.5. A *link proof* for the copy $\mathbf{T}_j$ is a proof string (Construction D.11) for $\mathsf{IOP}_Z^{\mathrm{link}}$ (Construction 4.13), whose IOP instance is (σ_0, σ_j) and whose external oracle strings are $\mathbf{T}_0$ and $\mathbf{T}_j$.

4.4 Requests and proofs

Definition 4.7 (Encoded request, offline and online proof, files). An *encoded request* is $\xi = (\mathsf{win}, i_0, i_1, y^+)$: a window $\mathsf{win} \in \mathcal{V}^L$ (Definition 3.1), token positions $i_0 < i_1 \le L$, and a token $y^+ \in \mathcal{V}$ with $y^+ = 0$ if $i_1 < L$. For lists x, y, fill of tokens of $\mathcal{V}$, with x and y non-empty and $|x| + |y| \le L + 1$, $\mathsf{build}(x, y, \mathsf{fill})$ is the encoded request with

$$\mathsf{win} = \text{the first } L \text{ tokens of } x \,\|\, y \,\|\, \mathsf{fill} \,\|\, 0\,0\cdots, \qquad i_0 = |x| - 1, \qquad i_1 = i_0 + |y|,$$

where y^+ is the last token of y if $|x| + |y| = L + 1$, and 0 otherwise; on all other inputs build is undefined. An *offline proof* π_{key} is a proof string (Construction D.11) for $\mathsf{IOP}_Z^{\mathrm{key}}$, an *online proof* π_{req} one for $\mathsf{IOP}_Z^{\mathrm{req}}$. With rt_j the Merkle root of table j (Construction D.11) and π_{link} the link proof of copy j (Definition 4.6), the *key file* holds $(\mathsf{rt}_0, \mathbf{a}_0, \pi_{\mathrm{key}})$, the *file of copy* j holds $(\mathsf{rt}_j, \mathbf{a}_j, \pi_{\mathrm{link}})$, and a *proof file* holds (j, π_{req}). A proof π of Π_Z (Construction 4.15) consists of fill, the contents of these three files and the two OOD seeds.

In running text a *request* is an instance (x, y) that is put to the prover; build turns each into one encoded request. The request relation (Definition 4.8) and true requests (Definition 4.17) concern encoded requests.

Definition 4.8 (Segments, cells, request relation). The arithmetization (Remark D.10, Given in outline) of the integer model at μ_Z (Definition 3.1) with the policy of Z (Definition 3.4) divides the model's run on a window into G *segments*, the last of which is the *head*. Segment g has

- *cells* Y_g, integer matrices of intermediate values;
- *identities*, polynomials with integer coefficients in the entries of A, the cells $Y_0, \dots, Y_g$ and the components of ξ;

- *lookups*, each a polynomial of the same kind, its *lookup key*, with a public table.

The *request relation* Req_Z is the set of pairs (A, ξ), with A in range and ξ an encoded request, for which there are cells of absolute value below the *relaxed bound* 2^{18} such that, modulo p, every identity is 0 and every lookup key is in its table.

Lemma 4.9 (Meaning of the request relation). [machine-checked] *Let μ be valid model parameters (Definition 3.6) and (C, θ, cap) a policy with $C, \theta < 2^{40}$ and $cap < 2^{16}$, and form the request relation of Definition 4.8 at these. Let A be in range, $M = \mathsf{wt}(A)$, and $\xi = (\mathsf{win}, i_0, i_1, y^+)$ an encoded request. Put $\bar{x}_i = (\mathsf{win}[0], \ldots, \mathsf{win}[i])$ and $I = \{\, i : i_0 \le i < \min(i_1, L-1) \,\}$. Then (A, ξ) is in the relation if and only if*

1. *the model is defined on* win *(Definition 3.1);*
2. *with* $\mathrm{gap}_i = \mathrm{gap}_M(\bar{x}_i, \mathsf{win}[i+1])$ *(Definition 3.4),*

$$\sum_{i \in I} \min(\mathrm{gap}_i + 2^8, C) \le \theta\,|I| \qquad \text{and} \qquad \#\{\, i \in I : \mathrm{gap}_i + 2^8 > C \,\} \le cap;$$

3. *if $i_1 = L$, then $\mathrm{gap}_M(\bar{x}_{L-1}, y^+) = 0$.*

One Lean theorem proves the first direction for all cells within the relaxed bound, and for the second that the honest cells satisfy the constraints within the honest bound; a second Lean theorem passes from the honest bound to the relaxed one; the two are joined on paper, and items 1 to 3 put a Lean definition in words (Appendix F names all three).

Beyond that join, no proof of the lemma is printed. Lemma A.33 is the form of its direction from the relation to conditions 1 to 3 that the proof of Theorem 5.1 uses: at an admissible configuration and at $\xi = \mathsf{build}(x, y, \mathsf{fill})$, membership in Req_Z gives that y meets pol_Z on x.

4.5 The interactive protocol and the argument

A batch opens its copy once for all its requests, as in [GLSTW23] and [BFRW25] (see also [CFW26]); the requests of a batch are the leaves of a Merkle tree, the *batch tree* (Construction D.11). The proximity tests are the zero-knowledge tests of [CFW26], with changes, of which this paper describes one (Table 11). A claim on the table (Construction 4.13, item 2), one linear claim with a linear form in committed masks, has the form of its committed sumcheck relation [CFW26, version of February 25, 2026, Section 2.4 and Definition 5.8]; the statements that are reduced to such a claim are this protocol's, and the sources of the reductions are in Table 11. The rounds that sample positions have proof of work ([CFWpc], with published uses in [Sta21, Section 3.11.3], [ACFY24a, Section 6.2] and [ACFY24b, Section 6.2]). Compilation follows the BCS transformation [BCS16]; the committed tables are compiled as external oracle strings, as in [COS20, Section 10] and [CY24, Section 32.7], following [CFWpc].

Assumption 4.10 (Random oracle). The hash function H that the algorithms of Π_Z (Construction 4.15) and all adversaries query is a random oracle (Definition 2.1) with outputs of $\lambda = 256$ bits. It models the *format's hash function* O^*, the function from byte strings to 32-byte strings, built on BLAKE2b, by which the release instantiates H. Every probability in a statement that cites this assumption is taken over the choice of H and over the random choices of the algorithms.

That the Lean definition of BLAKE2b, beneath the Lean term for O^*, is BLAKE2b [RFC7693] rests on known answers alone.

Assumption 4.11 (Separate second hash). The second hash function H' that the algorithms of Π_Z (Construction 4.15) evaluate is one particular function, the second hash function of the release's format. It is not sampled, and its values do not depend on H. It is assumed that O^* (Assumption 4.10) can be modelled by an oracle sampled independently of H'.

Remark 4.12 (On the second hash function). Like O^*, H' is built on the compression function of BLAKE2b. The statements that cite Assumption 4.11 use no further property of H'. They are proved for this one function, on which $\mathsf{Chk}(Z)$ depends, and are not stated for another function in its place. The programs, which check the

lookup tables against their four digests under H' in the parameter file, also assume that H' resists collisions. Remark D.13 (How the two hash functions differ) says how the two hash functions differ, and why a collision of H' merges no two of V_Z's queries.

Construction 4.13 (The interactive protocol $\mathsf{FlashProve}_Z$). $\mathsf{FlashProve}_Z$ consists of the encoder, which maps a matrix A, pads and random coefficients to a committed table (Definition 4.2); the map from a table and an OOD seed to its OOD answers (Definition 4.3); Coll and Dec (Definition 4.5); the largest size R of a batch; and three IOPs with external oracle strings (Definition 2.3). After a table is sent, V samples its OOD seed and P answers. The OOD sample belongs to no round of the three IOPs; it is part of their IOP instances.

IOP	IOP instance	Oracle strings	Set $\mathcal{L}$
$\mathsf{IOP}_Z^{\mathrm{key}}$	σ_0	$\mathbf{T}_0^*$	$\mathsf{Coll}(\mathbf{T}_0^*, \varsigma_0)$ or $\mathsf{Dec}(\mathbf{T}_0^*, \sigma_0) \neq \perp$
$\mathsf{IOP}_Z^{\mathrm{link}}$	(σ_0, σ_j)	$\mathbf{T}_0^*, \mathbf{T}_j^*$	the relation of the link proof (Definition 4.6)
$\mathsf{IOP}_Z^{\mathrm{req}}$	(ξ, σ_j)	$\mathbf{T}_j^*$	$\mathsf{Dec}(\mathbf{T}_j^*, \sigma_j) = \perp$ or $(\mathsf{Dec}(\mathbf{T}_j^*, \sigma_j), \xi) \in \mathsf{Req}_Z$

$\mathcal{L}$ is the set for which Proposition 5.14 states round-by-round soundness; Req_Z is the request relation (Definition 4.8). The rounds are those of Construction D.9, with this structure.

1. $\mathsf{IOP}_Z^{\mathrm{key}}$ and $\mathsf{IOP}_Z^{\mathrm{req}}$ have a *first part*, a *shared part* and a *last part*; $\mathsf{IOP}_Z^{\mathrm{link}}$ has one part. In a batch of at most R requests on one copy, the shared part is run once, the other two once for each request.
2. The first part reduces its statement to one *claim on the table*, $\langle h \otimes \mathrm{eq}(\psi, \cdot), A\rangle + \eta = b$, with h, ψ, b known to V and η a linear form in committed masks. In $\mathsf{IOP}_Z^{\mathrm{key}}$ the statement is that A is in range; in $\mathsf{IOP}_Z^{\mathrm{req}}$ it is $(A, \xi) \in \mathsf{Req}_Z$.
3. The shared part folds the table's rows at a sampled point, opens at most ν_{off} columns of $\mathbf{T}_0^*$, or at most ν_{on} of $\mathbf{T}_j^*$, and compares the fold with them and with the OOD answers. The last part relates the claim to the fold.
4. $\mathsf{IOP}_Z^{\mathrm{link}}$ folds both tables at one sampled point, opens at most ν_{first} columns of $\mathbf{T}_0^*$ and ν_{copy} of $\mathbf{T}_j^*$, and compares the folds on the plain entries at a second sampled point.
5. Exactly the rounds m in which V samples positions have proof of work, of κ_m bits (Definition 2.5).
6. Each of the three verifiers rejects if $\mathsf{Chk}(Z) = 0$ (Definition 4.14).

Definition 4.14 (Parameter checks). $\mathsf{Chk}(Z) \in \{0, 1\}$ is a function of Z and H' alone. Chk is a conjunction of tests, in four groups; oracle messages, Lookups and blocks are those of Construction D.6.

1. *Errors.* A list of the IOPs' rounds, each with an error bound, is derived from Z; every bound is at most the parameter file's target.
2. *Rounds.* The derived rounds, sizes and oracle messages are constants of V_Z.
3. *Layout.* An oracle message is opened only after it is sent; the layout file places each cell where a Lookup reads it.
4. *Sizes.* A challenge's blocks, the bits of proof of work, the Merkle trees' depths and the compiled verifiers' queries are within fixed bounds.

The closed list is the Lean term that Appendix F names for this definition (Remark D.10, Given in outline).

In Construction 4.15 the committer and prover of Π_Z are those of Construction D.12. In a machine-checked statement Π_Z is the Lean term with the same verifier and extractor, whose committer and prover are constant functions (Remark 4.16). The statements of Section 5 on Π_Z depend on neither: Definition 2.7 names the verifier and an extractor alone.

Construction 4.15 (The argument Π_Z). Fix Z; H' is the function of Assumption 4.11. H is an oracle of all algorithms; in every experiment it is sampled after Z is fixed. $\Pi_Z = (\mathsf{Rnd}_Z, \mathsf{Commit}_Z, P_Z, V_Z)$ is a commit-and-prove argument (Definition 2.6) for the relation of greedy outputs at the model parameters μ_Z (Definition 3.2 and Definition 3.8). Construction D.11 defines Merkle roots rt, the map Seed, which derives an OOD seed from a root by two queries, and the compiled verifier of an IOP. code denotes fixed injective maps

to bit strings, one on configurations and one on pairs of a key and a digest; $\mathsf{dg}_Z = H(\mathsf{code}(Z))$ is the *digest* of Z. A proof π is a list of tokens and five byte strings.

Verifier. $V_Z^H(c, x, y, \pi)$ is deterministic. It evaluates H' on public data alone: on parts of Z, and on x and y.

1. *Scope test.* Reject if $\mathsf{Adm}(Z) = 0$ (Definition 3.8).
2. *Unpacking.* Unpack π, at dg_Z and by one query to H (Construction D.11), into $(\mathsf{fill}, \mathsf{k}, j, \mathsf{rt}_j, \sigma_j, \pi_{\text{key}}, \pi_{\text{link}}, \pi_{\text{req}})$, where $\mathsf{k} = (\mathsf{rt}_0, \sigma_0)$ is the *key* and each of the last three may be $\perp$; if this fails, continue with the fixed tuple of Construction D.11, at which step 4 rejects.
3. *Public readings.* Compute the *public readings* $(\mathsf{dg}_Z, \mathsf{par})$ (Construction D.11); they enter every string that steps 7 and 8 hash.
4. Reject unless $\xi = \mathsf{build}(x, y, \mathsf{fill})$ is defined (Definition 4.7).
5. Reject unless $H(\mathsf{code}(\mathsf{k}, \mathsf{dg}_Z)) = c$.
6. Reject if one of π_{key}, π_{link}, π_{req} is $\perp$.
7. Reject unless $\varsigma_0 = \mathsf{Seed}(\mathsf{rt}_0)$ and $\varsigma_j = \mathsf{Seed}(\mathsf{rt}_j)$: the OOD seeds that V samples in Construction 4.13 are derived from the roots.
8. Run the three compiled verifiers, each on a list of digests (roots, and OOD answers written as blocks of λ bits), an IOP instance and a proof string:

$$\begin{aligned} \mathsf{IOP}_Z^{\text{key}}: &\quad (\mathsf{rt}_0, \mathbf{a}_0), \quad \sigma_0, \quad \pi_{\text{key}}; \\ \mathsf{IOP}_Z^{\text{link}}: &\quad (\mathsf{rt}_0, \mathsf{rt}_j, \mathbf{a}_0, \mathbf{a}_j), \quad (\sigma_0, \sigma_j), \quad \pi_{\text{link}}; \\ \mathsf{IOP}_Z^{\text{req}}: &\quad (\mathsf{rt}_j, \mathsf{rt}_j, \mathbf{a}_j), \quad (\xi, \sigma_j), \quad \pi_{\text{req}}. \end{aligned}$$

   The third list holds the copy's root twice. Each verifier includes the parameter checks $\mathsf{Chk}(Z)$ (Construction 4.13). Reject unless all three accept.
9. Reject unless the plain parts in π_{req} hold the number that π names for its copy. Otherwise accept.

Steps 4 to 9 are the *compiled steps* of V_Z.

Extractor. $\mathsf{Ext}_Z(c, \mathsf{tr})$, for a query–answer trace tr, makes no query; its value is $\perp$ if a step fails.

1. Find the first entry of tr whose answer is c and whose query is $\mathsf{code}(\mathsf{k}, \mathsf{dg})$ for a key $\mathsf{k} = (\mathsf{rt}_0, \sigma_0)$ and a digest dg.
2. Translate the queries of tr, at the digest dg, into queries for leaves and inner nodes of Merkle trees (Construction D.11).
3. From these, rebuild the word $\mathbf{T}_0^*$ under the root rt_0.
4. Return $\mathsf{wt}(\mathsf{Dec}(\mathbf{T}_0^*, \sigma_0))$.

Committer and prover are those of Construction D.12. P_Z serves up to R instances as one batch; V_Z checks each proof by itself.

Remark 4.16 (Honest parties). The machine-checked statements on Π_Z concern V_Z, Ext_Z and the model, which are written in Lean, as above, up to what Remark D.10 (Given in outline) lists. Index-binding knowledge soundness (Definition 2.7) depends on nothing else: a commitment is whatever $\lambda = 256$ bits the adversary presents.

In the Lean term for Π_Z, Rnd_Z, Commit_Z and P_Z are constants, and the provers of the three IOPs send fixed messages. Hence completeness, index privacy (and with it anything on hiding the weights), correctness of extraction, extractability of the commitment and no two openings cannot be stated with any content of that term; a second Lean term, with the same verifier and with the committer and prover that the Lean function compile yields (Remark B.7 (Which compiler); its output: Definition 6.15) from the Lean provers, is the object of Section 6.5. Honest provers of the three IOPs, and the committer and prover compiled from them, are written in Lean as definitions, for one request to a proof; completeness and zero knowledge are machine-checked for these, with V_Z and with their limits (Section 6.5). Of the committer compiled from

them, with V_Z and its extractor, soundness with extractability of the commitment, no two openings and correctness of extraction are proved, at the files in use, with their limits, in Section 6.5; these read no prover. Of the released programs none of these properties is proved. The results of Theorem 6.3, Corollary 6.7, and Theorem 6.9 concern other objects: the table experiment, the whole-key argument and $\mathsf{IOP}_Z^{\mathrm{link}}$. The extractor need not return the weights that a committer used. The results on the explicit committer (Definition 5.16) are statements on Π_Z^{Γ}, not on Π_Z.

The number that a proof names for its copy (step 9 of Construction 4.15) is tied to rt_j by no step, and an acceptance does not certify it.

4.6 True requests

Definition 4.17 (True request, constraints, honest bound). Let μ be valid model parameters (Definition 3.6), for a configuration Z those that Z states. Let A be in range (Definition 4.1) and $\xi = (\mathsf{win}, i_0, i_1, y^+)$ an encoded request (Definition 4.7), so that $y^+ = 0$ if $i_1 < L$. Under exact match, the encoded request is *true at* A if, for the integer model at μ and the weights $\mathsf{wt}(A)$ (Definition 3.1 and Definition 3.2),

1. the model is defined on win,
2. the token $\mathsf{win}[i+1]$ is the greedy token after $(\mathsf{win}[0], \ldots, \mathsf{win}[i])$ for all $i_0 \le i < \min(i_1, L-1)$, and
3. if $i_1 = L$, then y^+ is the greedy token after win.

The *constraints* of an arithmetization (Definition 4.8) are its identities and lookups. Cells *satisfy the constraints* modulo p, or over the integers, if every identity is 0 and every lookup key is an entry of its table, modulo p or as integers. Cells are *within the honest bound* if each has absolute value at most 2^7.

Lemma 4.18 (True requests and greedy outputs). [machine-checked] *Let μ be valid model parameters, A a digit matrix of $M \in \mathcal{I}$, and x, y, fill lists of tokens.*

(a) If $\mathsf{build}(x, y, \mathsf{fill})$ is defined and true at A (Definition 4.17), then y is a greedy output of M on x.

(b) If $(M, x, y) \in \mathsf{Serv}_\mu$ (Definition 3.11), then $\mathsf{build}(x, y, \mathsf{fill})$ with empty fill is defined and true at A.

5 Soundness

All theorems, propositions, lemmas and corollaries of this section are machine-checked, nine of them (Theorems 5.1 and 5.19 among them) by joining Lean theorems on paper, as the text beside each says. For each machine-checked statement, Appendix F names the Lean declarations that it renders. Lean proves Theorem 5.1 in two forms, at the same error: with the extractor left open, as in Definition 2.7, and with Ext_Z of Construction 4.15 named (Remark 5.2, What the theorem gives). The Lean form of Theorem 5.1 has one further hypothesis, that the three byte strings of Z decode without error; admissibility implies it by a further Lean theorem; Theorem 5.1 as printed joins the two, and so does every statement of this section whose Lean form carries that hypothesis, Corollary 5.10 and Theorem 5.11 for this reason alone.

5.1 The theorem

Theorem 5.1 (Soundness). [machine-checked] *Let Z be an admissible configuration (Definition 3.8) with the exact-match policy (Definition 3.4), fixed before H is sampled. Then the argument Π_Z (Construction 4.15) is index-binding knowledge sound (Definition 2.7) for the relation of greedy outputs at μ_Z (Definition 3.2), in the random-oracle model (Assumption 4.10 and Assumption 4.11), with error*

$$\varepsilon(q) \;=\; \frac{16\,T}{2^{128}} + \frac{(T+2)^2}{2^{256}}, \qquad T = q + 1 + 2^{23},$$

where q is the sum of a bound on the committer's queries to H and a bound on the prover's.

In T the 1 is the query of step 2 of V_Z (Construction 4.15 and Construction D.11), and the summand 2^{23} covers the verifier's queries in steps 3 and 6 to 9 (as the compiler counts them, not its calls to H) and one query that the proof adds; the query of step 5 is in neither, and the term $(T+2)^2/2^{256}$ accounts for it. Section 5.6 accounts for that term and for the factor 16.

Remark 5.2 (What the theorem gives). Except with probability $\varepsilon(q)$, an output that V_Z accepts is a greedy output of the weights that the straight-line extractor, Ext_Z of Construction 4.15, reads off the commitment and the committer's queries; these weights are determined before any proof is made. The theorem holds for every proof string: a proof that is made in a batch, which V_Z verifies singly, with its path in the batch tree, is covered at $\varepsilon(q)$, and N proofs sent at once are covered at $N \cdot \varepsilon(q)$, q being the committer's bound and the whole list's prover bound added up (Corollary 5.10).

Here $\varepsilon(q) \approx q/2^{124} + 2^{-101}$: 124 bits per query (Remark B.5, Bits per query). The bounds on Reed–Solomon codes strictly below the Johnson radius that the proof uses (Lemma 5.13 and Lemma A.23) are machine-checked, resting in part on files adapted from ArkLib and Mathlib (Remark B.18, Typed statement). On how the Lean proof is checked, see Remark B.17. Where $\mathsf{Chk}(Z) = 0$, V_Z rejects every input; $\mathsf{Chk}(Z_0) = 1$ (Corollary 5.8).

Remark 5.3 (Reading the theorem).

1. The configuration is admissible and fixed before the oracle. A choice of it after querying H, which is open to every provider, is not covered (Open problem 8.16; Limitation 8.5).

2. A second hash function is fixed (Assumption 4.11), an assumption that is neither tested nor proved and on which no run bears. The proofs hold for this function, not for every function in its place.

3. No witness is extracted: the relation has none, and no party is asserted to know the weights. The extractor need not be efficient: it is a function defined with the axiom of choice, not a program.

4. The extracted weights need not be the committer's: that they are is correctness of extraction (Section 5.5; Theorem 6.31). Ext_Z takes Z as an argument: no statement says that Ext_Z and $\mathsf{Ext}_{Z'}$ agree, or ties a commitment to one configuration.

5. The client checks three things. Exact match is a hypothesis that V_Z does not test: the client checks that $\mathsf{pol}_Z = (256, 256, 0)$; at other policies Theorem 5.11 applies. The conclusion holds whatever the other two are: the model parameters, which fix the integer model and lie in the scale file and, for ϵ_1, ϵ_2, ϵ_F, M_F and K_F, in the parameter file (Definition 3.7; Limitation 8.6, Scales can disable blocks), and the length of y (Limitation G.4, Prefixes count). The verifier is V_Z, not the C++ verifier (Limitation 8.1, Lean verifier only); on what a run trusts, see Remark B.10.

6. That the parameter checks pass, which is necessary for acceptance and not sufficient, is proved only at Z_0 and, for Theorem 5.11, at the configurations that differ from it only in the policy (Proposition 5.6). Besides the reduction of Remark 6.49, the one theorem on acceptance by V_Z is completeness with the Lean prover, on served requests and over a random oracle; that some request is served is stated by a lemma beside it, at constructed weights (Section 6.5). A verifier that rejects everything also meets Definition 2.7, and likewise round-by-round soundness (Proposition 5.14).

   At Z_0, and at the configurations that differ from it only in the policy, the parameter checks are ruled out as the cause, and in runs at Z_0 and Z_1 the executable form, which equals V_Z by Theorem 5.5, accepted honest proofs on the parsed proof files (Section 7.3); so there V_Z is seen not to reject everything, at the format's hash function and as far as a run is trusted (Remark B.10, Claim from files). These runs were made over Lean files that differ from the released ones, but for the calls of the reference run and of the second set of 41, which were made over the released Lean files (Appendix H.3). Elsewhere, and at any other hash function, the theorem may hold only because the verifier rejects everything (Limitation 8.9, Completeness on served requests only); over a random oracle, Section 6.5 excludes this where the files of Z are in use (defined there), and some request is served.

7. That Definition 3.8 and Construction 4.15 and the relation of greedy outputs are the Lean terms of Theorem 5.1 is checked by reading, not by Lean, except for the policy's three numbers; so is Construction 4.13, its item 5 by a count by programs.

8. A verifier that treated the programs' hash function specially would also meet the theorem, whose bound is over a random oracle (Limitation G.5, Fiat–Shamir instantiation).

Remark B.4 adds particulars to items 3 to 5.

5.2 The executable form and the default configuration

As a Lean term, V_Z cannot be evaluated, because parts of its definition use the axiom of choice. A second term, the executable form, can, and it computes the same decision.

Definition 5.4 (Executable form). A function O from byte strings to byte strings *induces* the hash function H_O, where $H_O(\mathsf{bs})$ consists of the first $\lambda = 256$ bits of $O(\mathsf{byt}(\mathsf{bs}))$; the injective coding byt of bit strings as byte strings and the order of bits are those of Construction B.8. The *executable form* of V_Z at O is the function E_Z^O that the algorithm of Construction B.8 computes. It maps a commitment c, a prompt x, an output y and a proof π (Construction 4.15) to 1 (accept), 0 (reject) or $\perp$.

Theorem 5.5 (The executable form computes the verifier). [machine-checked] *For all configurations Z, all functions O from byte strings to 32-byte strings, and all commitments c, prompts x, outputs y and proofs π (Definition 5.4),*

$$E_Z^O(c, x, y, \pi) \;=\; V_Z^{H_O}(c, x, y, \pi).$$

In particular, $E_Z^O(c, x, y, \pi) \neq \perp$.

Proposition 5.6 (The parameter checks pass at the default configuration). [machine-checked] *The default configuration Z_0 (Definition 3.7) is admissible (Definition 3.8), its policy is exact match, and $\mathsf{Chk}(Z_0) = 1$ (Definition 4.14). Every configuration Z that differs from Z_0 only in the values (C, θ, cap) of its policy, where $2^8 \le \theta \le C < 2^{32}$ and $cap \le 1024$, is admissible, and $\mathsf{Chk}(Z) = 1$.*

Remark 5.7 (How the proposition is checked). Proposition 5.6, a conjunction made on paper of Lean theorems, concerns Z_0 as written out in Lean, not the release's files (Remark D.2, Written copy). The proofs of its statements on Chk lie in a separate Lean library, which nothing beneath Theorem 5.1 or Proposition 5.14 imports, so neither of these rests on it. One constant, on which that library rests, equates the layout file as one string with its pieces. At the default thread stack Lean's kernel stops on it at too deep a recursion; at a larger one the kernel accepts it, both when the file is compiled and when the toolchain checks it again. The re-checker `leanchecker` stops on it in the same way at each stack limit that was tried, and no independent program has accepted it. It is not the only one. In passes over every other module of the main and the separate library, over Lean files that differ from the released ones, in code too, the re-checker stopped with the same message on 22 further modules of that library, and on no module of the main library. Replayed at the larger stack over the build of those files, by Lean's own replay, Lean's kernel accepted each of the 22. The check script replays all 1,438 modules of two libraries (`lean/AtFullSide/`, 914 modules; `lean/ShapeLines/`, 524), not counting the file added later (Remark B.17), by Lean's own replay at a larger stack and not by `leanchecker`; 23 of them are shown to need it, all of them modules of `lean/AtFullSide/`; on `lean/ShapeLines/` nobody has tried `leanchecker`. By import lines at least one of the 23 lies beneath the module of each theorem of Section 6.5 on completeness and zero knowledge, and beneath those of Theorem 6.31, of Proposition 6.17, part 1, and of Proposition 6.33, parts 1 and 4; and beneath none of the soundness theorems of Section 5 (Theorem 5.1; Theorem 5.17 and Theorem 5.19), nor beneath Theorem 5.5, Corollary B.11 or Theorem 6.20: no module of that library lies beneath any of them (`lean/GAPS.md`, gap 73; `lean/LIMITS.md`, L69).

Corollary 5.8 (At the default configuration). [machine-checked] *Let Z_0 be the default configuration (Definition 3.7). In the random-oracle model (Assumption 4.10 and Assumption 4.11), the argument Π_{Z_0} is index-binding knowledge sound (Definition 2.7) for the relation of greedy outputs at μ_{Z_0}, with the extractor Ext_{Z_0} and the error $\varepsilon(q)$ of Theorem 5.1. Moreover, $\mathsf{Chk}(Z_0) = 1$, and $E_{Z_0}^O = V_{Z_0}^{H_O}$ for every function O from byte strings to 32-byte strings.*

Lean has these three parts as separate theorems, joined here on paper; on the part $\mathsf{Chk}(Z_0) = 1$, see Remark 5.7.

Remark 5.9 (What a client gets). At Z_0, comparing three digests (Definition D.1; Remark D.2) settles the policy and the model parameters, and of the three checks of Remark 5.3 (item 5) the length of y remains. Corollary 5.8 is the conjunction of Theorem 5.1, Theorem 5.5, and Proposition 5.6 at Z_0. The bound $\varepsilon(q)$ also holds for the executable form's own decision (Corollary B.11), and the executable form has accepted, in calls that ran to their end, every honest proof at Z_0 counted in Section 7.3 (two of them by a separate build of the executable form alone, tied to V_Z by name and statement: Appendix H.3), in runs of Lean's evaluator

on synthetic weights; it has also accepted parsed proofs altered at entries that it does not read, and at three of them a C++ verifier of other programs† rejected the matching altered raw proof. These calls were made over Lean files that differ from the released ones, but for the calls of the reference run and of the second set of 41, which were made over the released Lean files (Appendix H.3).

The executable form is run inside Lean, by the command `lean-verify`, which evaluates $E_Z^{O^*}$ (Assumption 4.10). The statements concern the values of terms, not whether or when an evaluation ends; Remark B.10 lists what a run trusts beyond them.

Corollary 5.10 (Many proofs). [machine-checked] *Let Z and ε be as in Theorem 5.1. There is an extractor as in Definition 2.7, with output M^*, such that for all N, q_C, q_P, every q_C-query committer, and every q_P-query prover that receives the committer's state and outputs at once a list of at most N triples (x_i, y_i, π_i): with probability at most*

$$N \cdot \varepsilon(q_C + q_P),$$

V_Z accepts some (c, x_i, y_i, π_i) with $(x_i, y_i) \notin \mathsf{Lang}(M^*)$.

5.3 Other policies

Theorem 5.11 (Soundness under any policy). [machine-checked] *Let Z be an admissible configuration (Definition 3.8) with policy* pol*, fixed before H is sampled. Then, in the random-oracle model (Assumption 4.10 and Assumption 4.11), the argument Π_Z (Construction 4.15) is index-binding knowledge sound (Definition 2.7), with the error $\varepsilon(q)$ of Theorem 5.1, for the relation*

$$\{\, (M, (x, y)) \;:\; y \text{ meets } \mathsf{pol} \text{ on } x \text{ under the weights } M \,\}$$

of Definition 3.4 in the place of the relation of greedy outputs.

Remark 5.12 (Committer-chosen weights). As in every commit-and-prove argument, the committer chooses the weights. By itself, Theorem 5.11 therefore excludes nothing against a committer who commits to weights under which every output meets the policy. Such weights exist for many policies: under all-zero weights, every non-empty output that fits one window together with a non-empty prompt, all tokens in $\mathcal{V}$, meets every policy with $\theta, C \geq 257$ (Proposition 3.9). The theorem constrains a provider only together with correctness of extraction at the same extractor (Definition 2.9; for the explicit committer, Theorem 5.17) and with knowledge of who made the commitment (Limitation 8.7, Unattributed commitment). At exact match it also excludes an accepted empty output (Proposition 3.5).

5.4 Rounds and codes

Lemma 5.13 (Mutual correlated agreement strictly below the Johnson radius). [machine-checked] *Let $\mathbb{K}$ be a finite field, $\mathbb{K}_0$ a subfield and $D \subseteq \mathbb{K}_0$ a finite set. Let $\mathcal{C} \subseteq \mathbb{K}^D$ be the Reed–Solomon code of the polynomials over $\mathbb{K}$ of degree at most $a \geq 1$, evaluated on D, and put $\varrho = a/|D|$. Let $e \geq 1$ be an integer with $\delta = e/|D| < 1 - \sqrt{\varrho}$. Then $\mathcal{C}$ has mutual correlated agreement (Definition 2.2) at δ, for two words under the generator $(1, \mathsf{z})$ with z uniform in $\mathbb{K}$, with error $J/|\mathbb{K}|$, where*

$$J \;=\; \frac{2\,(\mathsf{mult} + \tfrac{1}{2})^5 + 3\,(\mathsf{mult} + \tfrac{1}{2})\,\delta\varrho}{3\,\varrho^{3/2}}\,|D| \;+\; \frac{\mathsf{mult} + \tfrac{1}{2}}{\sqrt{\varrho}}, \qquad \mathsf{mult} = \max\Big(\Big\lceil \frac{\sqrt{\varrho}}{1 - \sqrt{\varrho} - \delta} \Big\rceil,\; 3\Big).$$

The lemma is the statement as formalized in Lean, after [BCHKS25, Theorem 4.6]; its Lean proof runs through files adapted from ArkLib and Mathlib [ArkLib, Mathlib] (Appendix E.4). Remark B.18 (Typed statement) relates it to the published theorem, which Theorem 5.1 does not need.

Proposition 5.14 (Round errors). [machine-checked] *Let Z be an admissible configuration with $\mathsf{Chk}(Z) = 1$ (Definition 4.14). Then $\mathsf{IOP}_Z^{\mathrm{key}}$ and $\mathsf{IOP}_Z^{\mathrm{link}}$ are round-by-round sound, and $\mathsf{IOP}_Z^{\mathrm{req}}$, as a batch IOP for at most R encoded requests, is round-by-round knowledge sound (Definition A.1 and Definition A.2), each for its set $\mathcal{L}$ in Construction 4.13, with every round error, proof of work (Definition 2.5) included, at most*

$$\varepsilon_{\mathrm{rd}} = 2^{-128}.$$

For $\mathsf{IOP}_Z^{\mathrm{req}}$ *the witness is empty:* $\mathcal{L}^{\mathsf{wit}}$ *consists of the triples* $(\mathsf{in}, \mathsf{ext}, \star)$ *with* $(\mathsf{in}, \mathsf{ext}) \in \mathcal{L}$.

For the rounds of the Proximity tests, the source whose error terms the counts of bad challenges (Table 10) follow in form is named in Remark A.26.

Remark 5.15 (Single rounds).

1. These are errors of single rounds; the probability of a forgery is bounded by Theorem 5.1, at 124 bits per query. The proposition joins on paper three Lean theorems, one for each IOP. Where Z is admissible and $\mathsf{Chk}(Z) = 1$ (Section 6.5), Lean proves these round errors for the three IOPs together, as one statement, and bounds the error of the whole run before compilation by $67{,}391 \cdot 2^{-96} < 2^{-79}$; after compilation the bound stays that of Theorem 5.1.
2. In the rounds that sample positions, the bound holds only with the round's 20, 28 or 32 bits of proof of work; without them the errors are about 2^{-96} to 2^{-110} (a program's arithmetic, not machine-checked; where Lean joins the three, as in item 1, it is machine-checked that no round is worse than 2^{-96}). At Z_0 the 28 bits for the 64 positions of a table are needed: with 27 the bound exceeds 2^{-128} (Proposition D.8).
3. The figures that the C++ verifier prints follow another accounting of round errors. Under stricter counts than that accounting (all columns of the matrix of cells that is tested; a factor for the list size), some of its rounds would miss, by a planning calculation that is not machine-checked, its target of 2^{-128} (Remark G.2, Stricter counts). Proposition 5.14 does not depend on it: its bound is proved for the IOPs of Construction 4.13.

5.5 The commitment

A client needs soundness for the weights that were committed, not for weights that an extractor returns. Theorem 5.19 gives it, for an explicit committer: except with a small probability, an accepted output meets the policy under the committed weights themselves. It rests on Theorem 5.11 and on Theorem 5.17: where the parameter checks pass, Ext_Z returns the committed weights, except with probability about 2^{-89}. Theorem 5.17 states its errors in the form of the error of Theorem 5.1; Proposition 5.18 gives sharper ones for its parts 1 and 2, and Remark B.23 the form of part 1 of Theorem 5.19 with them.

Definition 5.16 (Explicit committer). Let Z be an admissible configuration (Definition 3.8). A *coin map* Γ maps a secret seed in $\{0,1\}^\lambda$ and two values of H to *coins*: random coefficients, pads and leaf salts for the first table $\mathbf{T}_0$ (Definition 4.6). The *explicit committer* is the pair $(\mathsf{Rnd}, \mathsf{Commit}_{Z,\Gamma})$:

1. Rnd outputs a uniform seed, without queries;
2. the randomness r of $\mathsf{Commit}^H_{Z,\Gamma}(M, \mathsf{r})$ is either coins or a seed sd; in the second case the coins are $\Gamma(\mathsf{sd}, \mathsf{dg}_Z, \mathsf{dg}')$, where dg' is the value of H at a code of dg_Z and the table's sizes;
3. from the coins and a digit matrix determined by M (Definition 4.1), it computes $\mathbf{T}_0$ and the key $\mathsf{k} = (\mathsf{rt}_0, \sigma_0)$ by the rules of Construction 4.15 and Construction D.11, and outputs only $c = H(\mathsf{code}(\mathsf{k}, \mathsf{dg}_Z))$.

Π^Γ_Z is Π_Z with this committer and any prover, V_Z and Ext_Z unchanged.

Theorem 5.17 (Extraction and extractability for the explicit committer). [machine-checked] *Let Z be an admissible configuration (Definition 3.8) at which the parameter checks pass, $\mathsf{Chk}(Z) = 1$ (Definition 4.14), and Γ a coin map (Definition 5.16), both fixed before H is sampled. Let*

$$\beta(q) \;=\; \frac{16\,(q + 2^{23})}{2^{128}} + \frac{(q + 2^{48} + 2^{23} + 2)^2}{2^{256}}.$$

In the random-oracle model (Assumption 4.10 and Assumption 4.11), Π^Γ_Z

1. *has correctness of extraction (Definition 2.9) for* Ext_Z *with error* $\beta(2^{35} + 2)$*;*
2. *has extractability of the commitment (Definition 2.10) for* Ext_Z *with error* β*;*
3. *has no two openings with error* 2β.

Proposition 5.18 (Sharper errors). [machine-checked] *Let*

$$\gamma(q) \;=\; \frac{3\,(q+2^{49}+4)^2}{2^{256}}.$$

Under the hypotheses of Theorem 5.17, its parts 1 and 2 hold with γ in the place of β. Part 1 also holds, with β and with γ, for every r, a secret seed or coins, that is fixed before H is sampled, in the place of $\mathsf{r} \leftarrow \mathsf{Rnd}$.

Theorem 5.19 (Soundness after a commitment by the explicit committer). [machine-checked] *Let Z be an admissible configuration with policy pol_Z (Definition 3.8 and Definition 3.4), Γ a coin map (Definition 5.16), $M \in \mathcal{I}$ and $\mathcal{A}_P$ a q_P-query algorithm, all fixed before H is sampled. In the random-oracle model (Assumption 4.10 and Assumption 4.11), let $\mathsf{r} \leftarrow \mathsf{Rnd}$, $c = \mathsf{Commit}^H_{Z,\Gamma}(M,\mathsf{r})$, $(x,y,\pi) \leftarrow \mathcal{A}_P^H(c,\mathsf{r})$ and $\mathsf{acc} = V_Z^H(c,x,y,\pi)$, and let*

$$\varepsilon_{\mathrm{com}}(q_P) \;=\; \frac{16\,T}{2^{128}} + \frac{(T+2)^2}{2^{256}} + \beta(2^{35}+2), \qquad T = q+1+2^{23},\;\; q = q_P+2^{35}+2,$$

with β as in Theorem 5.17.

1. $\Pr[\,\mathsf{acc}=1$ *and y does not meet* pol_Z *on x under M*$\,] \le \varepsilon_{\mathrm{com}}(q_P)$.
2. *If* pol_Z *is exact match,* $\Pr[\,\mathsf{acc}=1$ *and y is empty or not a greedy output of M on x*$\,] \le \varepsilon_{\mathrm{com}}(q_P)$.

Remark 5.20 (Sizes of the errors). Numerically,

$$\gamma(2^{35}+2) \approx 2^{-156}, \qquad \beta(2^{35}+2) \approx 2^{-89}, \qquad \beta(q) \approx \frac{q+2^{23}}{2^{124}}, \qquad \varepsilon_{\mathrm{com}}(q_P) \approx 2^{-88} + \frac{q_P}{2^{124}}.$$

Here $2^{35}+2$ bounds the explicit committer's queries; for one test that a pair opens c, β uses the looser bound 2^{48}, and γ uses it twice. The function β weakens γ to the form of the error of Theorem 5.1. Each part of Theorem 5.17, and part 1 of Theorem 5.19 with β or with γ, is a single Lean theorem, joined as Theorem 5.1 is with the further theorem on admissibility, and the three parts of Theorem 5.17 are joined here on paper; each statement of Proposition 5.18, and part 2 of Theorem 5.19, is an application of named Lean theorems to one another, which no single Lean theorem states. Part 2 of Theorem 5.19 is Lean's theorem at exact match taken, on paper, at the explicit committer's $2^{35}+2$ queries and the one query of step 2 of V_Z (Construction 4.15).

Remark 5.21 (Scope of the statements on the commitment).

- *Object.* The explicit committer models one part of the release's committing program. It was compared, in values only, with the committing program of other programs†, at the reduced configuration (Table 4), where none of these theorems is stated: what ran was a faster Lean definition of the committer, proved equal to it, with two further Lean definitions, for the table and for the coins, which no theorem on the committer mentions. Nothing is proved about the release's committing program (Limitation G.1, Implementation unverified).
- *Checks.* $\mathsf{Chk}(Z) = 1$ is machine-checked only at the configurations of Proposition 5.6. There the hypothesis of Theorem 5.17 is met. Where $\mathsf{Chk}(Z) = 0$, Theorem 5.19 is vacuous (Remark B.22, Proof outlines for the commitment).
- *Acceptance.* The explicit committer outputs no key file and has no honest prover, so no proof is shown to be accepted under its commitment. Who made a given commitment is outside the statements (Limitation 8.7, Unattributed commitment). No test covers the three properties of Theorem 5.17.

That the properties of Theorem 5.17 fail for a committer that replaces every digit matrix by the all-zero matrix, and that a committer whose commitment is a constant also meets the bound of Theorem 5.19, which constrains a committer only on the event that V_Z accepts, is Proposition B.24.

5.6 Proof outline

The proof of Theorem 5.1 has four parts (Appendix A).

Compilation. Steps 4 to 9 of V_Z are the output of the Lean library's compiler (Construction D.11) at the three IOPs (Remark B.7, Which compiler). The lemma on this compiler (Lemma A.15) rests on an argument over separate oracles (Lemma A.11), which fixes an event Bad on traces (Definition A.5): a new answer repeats a value of the trace, changes the state of an IOP from 0 to 1, or gives an OOD seed with a collision. A Q-query algorithm meets Bad with probability at most $Q\,\varepsilon_{\mathrm{rd}} + 2\,(Q+1)^2/2^{256}$ (Lemma A.8). Outside Bad, an accepting run of a compiled verifier gives the relation of its IOP at the words under its roots (Lemma A.9). Hence the first table decodes, the copy decodes to the same matrix A, and $(A, \xi) \in \mathsf{Req}_Z$ (Lemma A.10).

Rounds. Proposition 5.14 gives $\varepsilon_{\mathrm{rd}} = 2^{-128}$ where the parameter checks pass; elsewhere the three verifiers reject (Lemma A.22).

Meaning. If $(A, \mathsf{build}(x, y, \mathsf{fill})) \in \mathsf{Req}_Z$, then y meets pol_Z on x under $\mathsf{wt}(A)$ (Lemma A.33). This proves Theorem 5.11; at exact match, y is a greedy output (Proposition 3.5).

Error. The compiled verifiers make $q_V \le 2^{20}$ queries to the separate oracles of Construction D.11, which is not a bound on their queries to H (Remark B.13, Lists sent at once). The public readings make eight queries to H, and step 2 one, for dg_Z (Construction D.11; Lemma A.35); the proof adds one query that announces rt_0, and to each query four queries for OOD seeds (Lemma A.11; Definition A.4). Hence $Q = 5\,(q + q_V + 10) \le 5T$, since the 2^{23} in T covers $q_V + 9$. Then $Q\,\varepsilon_{\mathrm{rd}} \le 5\,T/2^{128}$, and where $\varepsilon(q) < 1$ the quadratic term is at most $6\,T/2^{128}$. The factor 16 is not tight: the arithmetic lemma is stated for round errors up to $2\,\varepsilon_{\mathrm{rd}}$. The term $(T+2)^2/2^{256}$ accounts for replacing the key by its hash value c (Lemma A.14 and Lemma A.36).

6 Zero knowledge and completeness

This section is on zero knowledge, formally index privacy (Definition 2.14), and on completeness; what is proved is exact honest-verifier simulation up to private aborts before compilation (Definition 6.24; Theorem 6.27) and zero knowledge with request leakage after it (Definition 6.34; Theorem 6.35 and Theorem 6.37; Remark 6.40). For the Lean provers, honest provers of the three IOPs written in Lean as definitions, completeness and zero knowledge are machine-checked in the forms that Section 6.5 states, before and after compilation, with the limits stated there (Limitation 8.3 (Zero knowledge of definitions in Lean); Limitation 8.9 (Completeness on served requests only)); that the forms of zero knowledge are index privacy is not stated. On zero knowledge three further objects have machine-checked results. On completeness, further: Remark 6.49.

- *The committed tables of* $\mathsf{FlashProve}_Z$*,* before compilation. With fresh randomness and within the opening budget, their OOD answers and opened columns are independent of the weights (Theorem 6.3).
- *The whole-key argument* (Section 6.3). Its index privacy reduces to five conditions on its provers, five of twenty (Theorem 6.6). In Lean the compiled zero-knowledge statement of Section 6.5 is obtained through the theorem on all twenty conditions (Theorem C.25): inside its proof all twenty conditions are met by the Lean provers of Section 6.5, at one copy of the table. No Lean theorem states that by name; we have it from a reading of the proof's calls. Every limit of the results before compilation holds for that statement as well (Remark 6.41). With constant-message provers, fresh random masks, one request to a proof and the whole key, not a 256-bit hash, as the commitment, and against any verifier, what it shows of the tables is simulated from the leakage alone, up to a stated error (Corollary 6.7).
- *The IOP of the link proof,* before compilation. At admissible Z its honest prover has an exact honest-verifier simulator, given the two tables' OOD samples and opened columns, which is not a statement that the weights are hidden. At admissible Z with $\mathsf{Chk}(Z) = 1$, the link's verifier accepts the same prover at every challenge that passes proof of work, for all weights, at one fixed public-files digest, at which V_Z is not proved ever to be called (Theorem 6.9).

6.1 Leakage

The leakage (Definition 2.13) is what a simulator is given: $\mathsf{pp}(M)$, N and each batch's revealed instances. What a client of the released programs sees by design, including timing and failures, is listed in Remark G.10.

Remark 6.1 (What the leakage contains). The simulator may depend on Z, which is quantified before M in every statement. In the statements on index privacy the simulator is given the leakage (Definition 2.13); what the simulators of Section 6.5 are given is listed there. In the machine-checked statements on index privacy pp is constant, and each concerns one commitment. Every revealed instance lies in $\mathcal{S}_M$ (Limitation G.11, Outcome leakage), and the released tools compute the scale file of Z and lines of its parameter file from the weights (Section 3.3); these two files encode μ_Z (Definition 3.7). No statement bounds what the revealed instances or these files reveal. The environment is quantified after M and may depend on it. So its instances can carry information about the weights into both views, and no statement bounds that.

6.2 Committed tables

The protocol has copies because a table is shown to hide the weights for at most t opened columns. In the table experiment an unbounded $\mathcal{A}$ names an OOD seed for each table before the table is sampled, and receives OOD answers, chosen columns, and runs of the three IOPs with constant-message provers. It is within the opening budget if at most t columns of each table are opened, where each run is charged 62 or 64 columns.

Definition 6.2 (Constant-message prover). A prover for an IOP (Definition 2.3) is a *constant-message prover* if in every round it sends one fixed message, the same for all inputs, coins and challenges, and never aborts. Its external oracle strings are the committed tables of its input.

Theorem 6.3 (Committed tables before compilation). [machine-checked] *Let Z be admissible (Definition 3.8), with opening budget $t = 44{,}040{,}191$. For all matrices A, A' over $\mathbb{F}$ and every algorithm $\mathcal{A}$ within the opening budget (Definition C.6), the outputs of the table experiments with fresh randomness for each table and constant-message provers (Definition C.5),*

$$\mathsf{Tab}_Z(A, \mathcal{A}) \quad \textit{and} \quad \mathsf{Tab}_Z(A', \mathcal{A}),$$

are identically distributed.

Remark 6.4 (No prover message is covered). By Theorem 6.3, what an unbounded, adaptive $\mathcal{A}$ within the opening budget sees of the committed tables is perfectly independent of the weights. The theorem is not index privacy.

- No message that an honest prover computes from A is covered, alone or jointly with the tables.
- Each table has one OOD sample, and its randomness is used at one configuration. Randomness derived by a pseudorandom function, as in the released programs, is not covered. An OOD seed is chosen before its table is sampled. A seed computed from the table, such as a hash of its root, as in the released programs, is not covered.

For the OOD answers Theorem 6.3 rests on Lemma C.9, which follows [CFW26, version of February 25, 2026, Section 9.2].

Proposition 6.5 (The opening budget is tight for one table). [machine-checked] *Let $t = 44{,}040{,}191$ and $s = 1$, at the table sizes of Z_0 (Definition 3.7). Let $A = 0$ and let A' have the single nonzero entry $A'[0][0] = 1$, both in range (Definition 4.1); let ς° be the OOD seed with point $(1, \ldots, 1)$ and factor 1. For every set $Q \subseteq D$ of $t + 1$ positions, the distribution of the pair of the OOD answers at ς° and the columns $(\mathbf{T}[\phi])_{\phi \in Q}$ of a committed table $\mathbf{T}$ (Definition 4.2) differs between A and A'.*

6.3 After compilation

The *whole-key argument* $\Pi_Z^{\mathrm{wk}}[\mathbf{P}]$ (Definition C.20) differs from Π_Z as follows.

- Matrices A and encoded requests ξ take the place of indices and instances.
- The commitment is the whole key: the roots and OOD samples of $\mathbf{T}_0, \ldots, \mathbf{T}_K$ (each sample at a seed that is a hash of its table's root), an offline proof and K link proofs.
- A proof is for one request, on a uniformly chosen copy; N_{c} bounds the requests simulated on one copy.

- The proofs are compiled runs (Definition C.18) of provers $\mathbf{P} = (P_{\text{key}}, P_{\text{link}}, P_{\text{req}})$, which are parameters.
- The verifier runs steps 6 to 9 of V_Z, at fixed values $(\mathsf{dg}, \mathsf{par})$ of its public readings.

Theorem 6.6 requires five conditions of each prover P_* (in full: Definition C.21):

1. its external oracle strings are the committed tables;
2. a simulator Sim_*, given the IOP instance, the challenges and the opened columns, reproduces what P_* discloses, except with probability ζ_*^{sim};
3. P_* aborts for a reason that depends on its input with probability at most ζ_*^{ab};
4. the prefixes of its leaf salts are public;
5. all coins are uniform.

Theorem 6.6 (Reduction of index privacy to five conditions). [machine-checked] *Let Z be admissible, $K \le 710{,}324$ and $N_{\text{c}} \le 688{,}127$, and fix any values $(\mathsf{dg}, \mathsf{par})$. Suppose that provers $\mathbf{P}$ for the three IOPs of $\mathsf{FlashProve}_Z$ and simulators Sim_* satisfy the five conditions of Definition C.21 on a set $\mathcal{J}$ of matrices and a set $\mathcal{Y}$ of pairs (A, ξ), with errors $\zeta_*^{\text{sim}}, \zeta_*^{\text{ab}}$. Let $\mathcal{S}_A \subseteq \{\xi : (A, \xi) \in \mathcal{Y}\}$ for $A \in \mathcal{J}$. Then the whole-key argument $\Pi_Z^{\text{wk}}[\mathbf{P}]$ (Definition C.20), with constant pp, has index privacy on $\mathcal{S}$ for batches of at most 1 (Definition 2.14), with error*

$$\begin{aligned}\zeta(q, N) \;&= K \cdot \Pr\big[\mathrm{Bin}(N, 1/K) > N_{\text{c}}\big] \;+\; \zeta_{\text{key}}^{\text{sim}} + K\zeta_{\text{link}}^{\text{sim}} + N\zeta_{\text{req}}^{\text{sim}} \\ &\quad +\; \zeta_{\text{key}}^{\text{ab}} + K\zeta_{\text{link}}^{\text{ab}} + N\zeta_{\text{req}}^{\text{ab}} \;+\; \zeta_{\text{hash}}(q, N),\end{aligned}$$

where ζ_{hash} is the hashing error (Definition C.24).

Corollary 6.7 (Committed tables after compilation). [machine-checked] *Let Z be admissible with $\mathsf{Chk}(Z) = 1$, $K \le 710{,}324$ and $N \le N_{\text{c}} \le 688{,}127$, and fix any values $(\mathsf{dg}, \mathsf{par})$. Let $\mathbf{P}$ consist of constant-message provers (Definition 6.2), with fresh uniform randomness and these parameters of the compiled runs (Definition C.18 and Definition C.19): search limit 2^{40}; $\ell_{\text{s}} = 320$ uniform bits in each leaf salt; all-zero public prefixes; leaf 0 of the batch tree. Let $\mathcal{S}$ assign to every matrix over $\mathbb{F}$ any set of requests, and let*

$$\zeta_{\max}(q, N) = \frac{q\, q_{\text{hon}} + (9 + 3 \cdot 2^{40})\, q_{\text{hon}}\big(2(K+1) + q_{\text{hon}}\big)}{2^{256}}, \quad q_{\text{hon}} = (K+1)\, 2^{32} + 2^{37} + K\, 2^{25} + N\, 2^{39}.$$

1. *$\Pi_Z^{\text{wk}}[\mathbf{P}]$ has index privacy on $\mathcal{S}$ for batches of at most 1 (Definition 2.14), with an error ζ such that $\zeta(q, N) \le \zeta_{\max}(q, N)$. For $K = N = 1$ and $q \le 2^{64}$,*

$$\zeta_{\max}(q, 1) \;\le\; 2^{-135}.$$

2. *In the real experiment, for every matrix, every environment valid at it and every q-query V^*,*

$$\begin{aligned}\Pr[\pi_{\text{key}} = \perp \text{ or some } \pi_{\text{link}} = \perp] &\le (K+1) \cdot 71 \cdot 2^{-256}, \\ \Pr[\text{some } \pi_{\text{req}} \text{ in the view} = \perp] &\le N \cdot 46 \cdot 2^{-256} + \zeta_{\max}(q, N).\end{aligned}$$

The corollary's search limit is 2^{40}, that is, $2^{40} + 1$ attempts in each proof-of-work search. The corollary's bound, with constant-message provers, fresh random masks, one request to a proof and the whole key as the commitment, and where the parameter checks pass, has these two values for $q \le 2^{64}$. At one copy and one request, $\zeta_{\max} \le 2^{-135}$, as the corollary states. At the largest K and N that the corollary allows, $K = 710{,}324$ and $N = 688{,}127$ (N counts requests on all copies together), $\zeta_{\max} < 2^{-97}$, evaluated by hand.

Remark 6.8 (What holds after compilation). By Corollary 6.7, with constant-message provers, fresh random masks, one request to a proof and the whole key, not a 256-bit hash, as the commitment, what the commitment and the proofs show of the committed tables (roots, OOD samples, opened columns) is simulated up to $\zeta_{\max}$. This holds against any verifier within the corollary's bounds. Besides the verifier's queries the simulator is given the leakage alone, not the weights.

1. *Conditional.* Of 20 conditions (Theorem C.25) fifteen are proved (a count of conditions, not of work): those that mention no prover, coin or simulator; five of the fifteen hold by type or default. The other five concern the three honest provers, their simulators and coins; the Lean provers meet them inside the proof of the compiled zero-knowledge statement of Section 6.5, at one copy of the table; no Lean theorem states that by name, and we have it from a reading of the proof's calls (Open problem 8.13).
2. *Constant messages.* Fixed messages are trivially independent of the weights. Hence, for the provers' own messages, a result on index privacy is meaningful only together with completeness of the same provers (for the Lean provers: Section 6.5). The corollary covers nothing that the three provers (in Corollary C.29: the other two) would compute from the weights in their rounds.
3. *The whole key.* The commitment of Π_Z has 256 bits, not the whole key. No statement transfers either result to Π_Z.
4. *Single requests.* Both results are for one request to a proof. Lean's statements on completeness and zero knowledge do not cover batches, which the released programs use (Open problem 8.14).
5. *Proofs are produced.* A constant-message prover never aborts, so its compiled run is $\perp$ only if a proof-of-work search exhausts its limit (Definition C.18). Part 2 bounds the probability of that; without it, part 1 could be a statement on views that hold no proof.

6.4 The link proof

Theorem 6.9 (Link proof before compilation). [machine-checked] *Let Z be admissible (Definition 3.8), and let P_{link} be the honest prover of $\mathsf{IOP}_Z^{\text{link}}$ (Construction D.9), with uniform coins. For every matrix A over $\mathbb{F}$ and all pads, random coefficients and OOD seeds of $\mathbf{T}_0$ and $\mathbf{T}_j$:*

1. *P_{link} never aborts, and one honest-verifier simulator for all of them (Definition C.2), given the OOD samples (σ_0, σ_j), the challenges ch, and the at most 62 columns that the verifier opens in each table, is exact at every ch.*
2. *Let $\mathsf{Chk}(Z) = 1$, and let the proof-of-work bits of ch be those that the compiled verifier (Construction D.11) marks at $(\mathsf{dg}, \mathsf{par}) = (0^\lambda, \emptyset)$. Then the verifier of $\mathsf{IOP}_Z^{\text{link}}$ accepts P_{link} at every ch that passes the proof-of-work checks.*

Remark 6.10 (Not a statement on the weights). Theorem 6.9 concerns one of the three provers, at the default configuration's size, which every admissible Z has. By part 1, to an honest verifier the link proof's messages add nothing to what the two tables' OOD answers and opened columns show. These are computed from the weights and the tables' randomness (on their distribution: Theorem 6.3), so by itself part 1 does not state that the weights are hidden. Part 1 extends Corollary 6.7 to P_{link}, with the other two provers constant-message provers, at $(\mathsf{dg}, \mathsf{par}) = (0^\lambda, \emptyset)$, values at which V_Z is not proved ever to be called (Corollary C.29). Nor is it proved at these values that the compiled verifier accepts P_{link}; on acceptance after compilation: Section 6.5.

6.5 Completeness and zero knowledge of the Lean provers

This subsection concerns honest provers of $\mathsf{IOP}_Z^{\text{key}}$, $\mathsf{IOP}_Z^{\text{link}}$ and $\mathsf{IOP}_Z^{\text{req}}$ that are written in Lean as definitions, and the argument compiled from them, which proves one request per proof. The main results are the theorems on that argument, Theorem 6.29, Theorem 6.31, Theorem 6.35, and Theorem 6.37; of the definitions of this subsection their statements need only Definition 6.11, Definition 6.12, Definition 6.14, Definition 6.15, and Definition 6.34, so a reader may pass from Remark 6.18 to the paragraph "After compilation". Zero knowledge is stated in two forms: before compilation as exact honest-verifier simulation up to private aborts (Definition 6.24 and Theorem 6.27), after compilation as zero knowledge with request leakage (Definition 6.34, Theorem 6.35, and Theorem 6.37).

The theorems and propositions below are machine-checked, and no written proof is given: the formal statements are the Lean declarations that Appendix F lists under each number. Some printed statements say less than their Lean forms; where more than an instantiation is needed, the remark that follows the statement gives the step. Their Lean forms have one further hypothesis, that the three byte strings of Z decode without

error; admissibility implies it by a further Lean theorem, and the statements below join the two. In the remarks a *request* is an instance (x, y), and y is its *claimed output*.

Hypotheses and objects.

Definition 6.11 (Configuration in use). Three conditions on a configuration Z (Definition 3.7) are used as hypotheses.

(Admissible configuration) $\mathsf{Adm}(Z) = 1$ (Definition 3.8).

(Checks pass) $\mathsf{Chk}(Z) = 1$ (Definition 4.14).

(Exact match) The policy pol_Z of an admissible Z is exact match (Definition 3.4).

The files of Z are *in use* if Z is admissible, $\mathsf{Chk}(Z) = 1$ and its policy is exact match; Z itself is then also called *in use*. At files in use the policy is exact match, which fixes the policy's three numbers; the five numbers ϵ_1, ϵ_2, ϵ_F, M_F, K_F stay free.

Definition 6.12 (Served instance, label, bit). Let μ be valid model parameters and $M \in \mathcal{I}$ (Definition 3.1). An instance (x, y) is *well-formed* if x and y are non-empty lists of tokens of $\mathcal{V}$ with $|x| + |y| \le L + 1$, that is, if $\mathsf{build}(x, y, \emptyset)$ is defined (Definition 4.7). It is *served* at M if $(M, x, y) \in \mathsf{Serv}_\mu$ (Definition 3.11), and *refused* otherwise. The *bit* $\mathsf{bit}_\mu(M, x, y) \in \{\mathsf{served}, \mathsf{refused}\}$ says which. The *label* of a well-formed instance is

$$\mathsf{lab}_\mu(M, x, y) \;=\; \begin{cases} \mathsf{served} & \text{if } (M, x, y) \in \mathsf{Serv}_\mu, \\ \mathsf{failed} & \text{if } (x, y) \text{ cannot be served at } M, \\ \mathsf{other\ tokens} & \text{if } (x, y) \text{ can be served at } M \text{ and } y \text{ is not a greedy output of } M \text{ on } x. \end{cases}$$

An instance that is not well-formed has the label other tokens.

Remark 6.13 (Served and can be served).

1. *Two conditions.* "Can be served" (Definition 3.11) requires only that the model be defined on the *window* of (x, y), the first L tokens of $x \,\|\, y \,\|\, 0\,0 \cdots$. "Served" requires in addition that y be a greedy output of M on x. The bit is membership in Serv_μ, not the weaker condition. Every served instance is well-formed.
2. *The Lean form.* In Lean the label is defined through the model's *result* on (x, y): the tokens that its run on the window chooses for the positions of y, or no result if the model is not defined on the window. A well-formed instance is served if the model has a result on it and y is that result; its label is failed if there is no result, and other tokens if the result differs from y. Lean theorems, at valid μ, give the form of Definition 6.12 by propositional steps.
3. *Zeros after the output.* The window includes the zeros that follow y. Hence a greedy output y of M on x has the label failed if the model is not defined on the whole window.
4. *No token.* The label names no token: other tokens says only that the result differs from y.

Definition 6.14 (Lean provers). Let Z be admissible. For $M \in \mathcal{I}$ let $A(M)$ be a digit matrix determined by M (Definition 4.1), and put

$$\mathsf{Serv}_Z^{\mathrm{iop}} \;=\; \big\{\, (A(M),\, \mathsf{build}(x, y, \emptyset)) \;:\; (M, x, y) \in \mathsf{Serv}_{\mu_Z} \,\big\}.$$

The *Lean provers* $\mathbf{P}_Z^{\mathrm{lp}} = (P_{\mathrm{key}}^{\mathrm{lp}}, P_{\mathrm{link}}^{\mathrm{lp}}, P_{\mathrm{req}}^{\mathrm{lp}})$ are provers with abort sets (Definition C.2) for $\mathsf{IOP}_Z^{\mathrm{key}}$, $\mathsf{IOP}_Z^{\mathrm{link}}$ and $\mathsf{IOP}_Z^{\mathrm{req}}$, the third for one encoded request. They are the Lean terms that Appendix F names for this definition; the paper does not print them. The input of each consists of a matrix A, the randomness and the OOD seed of each committed table on which it runs and, for $P_{\mathrm{req}}^{\mathrm{lp}}$, a natural number, the *copy number* (Construction 4.15, step 9).

Definition 6.15 (Compiled Lean argument). Let Z be admissible. The *compiled Lean argument* $\Pi_Z^{\mathrm{lp}} = (\mathsf{Rnd}_Z^{\mathrm{lp}}, \mathsf{Commit}_Z^{\mathrm{lp}}, P_Z^{\mathrm{lp}}, V_Z)$ is a commit-and-prove argument (Definition 2.6) for the relation of greedy outputs at μ_Z, with V_Z and Ext_Z of Construction 4.15.

- $\mathsf{Rnd}_Z^{\mathrm{lp}}$ samples one number r uniformly from a fixed range, without queries: r determines the committer's coins.
- $\mathsf{Commit}_Z^{\mathrm{lp}}(M, \mathsf{r})$ forms $\mathbf{T}_0$ and one copy $\mathbf{T}_1$ for $A(M)$, makes π_{key} and π_{link} as compiled runs (Definition C.18) of $P_{\mathrm{key}}^{\mathrm{lp}}$ and $P_{\mathrm{link}}^{\mathrm{lp}}$, and outputs the 256 bits $H(\mathsf{code}(\mathsf{k}, \mathsf{dg}_Z))$, where $\mathsf{k} = (\mathsf{rt}_0, \sigma_0)$.
- $P_Z^{\mathrm{lp}}(M, \mathsf{r}, x, y)$ repeats this run, queries included, and makes π_{req} as a compiled run of $P_{\mathrm{req}}^{\mathrm{lp}}$ on $\mathbf{T}_1$ and $\mathsf{build}(x, y, \emptyset)$; for (x, y) not well-formed, on a fixed encoded request in no pair of $\mathsf{Serv}_Z^{\mathrm{iop}}$. It outputs the *empty proof*, the empty list of tokens with five empty byte strings, if one of the three proof strings is $\perp$, else a proof with $\mathsf{fill} = \emptyset$, k, π_{key}, the copy's number 0 (the key's list of copies starts at place 0 and does not hold T_0, so this is T_1), rt_1, σ_1, π_{link} and π_{req}.

Remark 6.16 (Arguments with the same verifier). Π_Z^{lp} differs from the other arguments of this paper as follows.

1. Π_Z *with the committer and prover of Construction D.12.* The verifier and the extractor are the same. That committer forms K copies, and that prover serves batches of up to R instances on a copy that it chooses within the opening budget. Π_Z^{lp} has one copy, proves one instance per proof, and keeps no count and no state. $P_{\mathrm{link}}^{\mathrm{lp}}$ is the Lean term of which Theorem 6.9 speaks; no statement compares $P_{\mathrm{key}}^{\mathrm{lp}}$ or $P_{\mathrm{req}}^{\mathrm{lp}}$ with the honest provers of Construction D.9.
2. *The Lean term for* Π_Z *in the statements of Section 5* (Remark 4.16, Honest parties). Its committer and prover are constant functions. Π_Z^{lp} is a second Lean term, with the same verifier. Index-binding knowledge soundness depends on the verifier alone, so Theorem 5.1 holds for Π_Z^{lp} with the same error. For no other property is an equivalence of the two terms stated.
3. *The whole-key argument* (Definition C.20). Its commitment is the whole key. The commitment of Π_Z^{lp} has 256 bits; π_{key}, the copy's root and OOD sample, and π_{link} are parts of each proof.

Proposition 6.17 (The hypotheses are met). [machine-checked]

1. *The default configuration* Z_0 *is in use (Definition 6.11).*
2. *For every admissible* Z *there is a pair* $(A, \xi) \in \mathsf{Serv}_Z^{\mathrm{iop}}$ *(Definition 6.14) with* A *in range.*
3. *For every admissible* Z *with* $\mathsf{Chk}(Z) = 1$ *there are matrices* $A \neq A'$ *in range and an encoded request* ξ *such that* (A, ξ) *and* (A', ξ) *lie in* $\mathsf{Serv}_Z^{\mathrm{iop}}$.

Remark 6.18 (Constructed weights). That some request is served (Proposition 6.17, parts 2 and 3) is shown at constructed weights, by lemmas and not by Theorem 6.23 and Theorem 6.29. In part 1, Z_0 is the Lean value that Definition 3.7 names, the configuration as written out in Lean, not the release's files (Remark D.2; Remark 5.7).

Before compilation. A list of challenges of one of the three IOPs *passes* if all its proof-of-work bits are 0 (Definition 2.5); the verifiers reject every other list. A star in a subscript, as in P_*, Sim_*, α_* and η_*, is one of key, link, req, and SD is the statistical distance.

Definition 6.19 (Whole run). A *whole run* of $\mathsf{FlashProve}_Z$ (Construction 4.13) with a prover $\tilde{P}$ consists of the following rounds, in this order.

1. $\tilde{P}$ sends a word $\mathbf{T}_0^*$ as an oracle, V samples an OOD seed ς_0 (Definition 4.3), and $\tilde{P}$ sends answers $\mathbf{a}_0$; put $\sigma_0 = (\varsigma_0, \mathbf{a}_0)$.
2. The rounds of $\mathsf{IOP}_Z^{\mathrm{key}}$ on σ_0 and $\mathbf{T}_0^*$.
3. As in step 1, a word $\mathbf{T}_1^*$ with $\sigma_1 = (\varsigma_1, \mathbf{a}_1)$.
4. The rounds of $\mathsf{IOP}_Z^{\mathrm{link}}$ on (σ_0, σ_1) and $\mathbf{T}_0^*, \mathbf{T}_1^*$.
5. $\tilde{P}$ names an encoded request ξ.
6. The rounds of $\mathsf{IOP}_Z^{\mathrm{req}}$ on (ξ, σ_1) and $\mathbf{T}_1^*$.

The run is *accepted* if the verifiers of the three IOPs accept.

Theorem 6.20 (Soundness of a whole run before compilation). [machine-checked] *Let Z be admissible with $\mathsf{Chk}(Z) = 1$. In a whole run (Definition 6.19) let G be the event that*

$$\mathsf{Dec}(\mathbf{T}_0^*, \sigma_0) = \mathsf{Dec}(\mathbf{T}_1^*, \sigma_1) \neq \bot \qquad \textit{and} \qquad (\mathsf{Dec}(\mathbf{T}_0^*, \sigma_0), \xi) \in \mathsf{Req}_Z,$$

where Req_Z is the request relation (Definition 4.8). For every deterministic prover $\tilde{P}$, over uniform OOD seeds and uniform lists of challenges that pass,

$$\Pr\big[\textit{ the run is accepted and } \mathsf{G} \textit{ does not hold}\big] \;\leq\; 67{,}391 \cdot 2^{-96} \;<\; 2^{-79}.$$

Remark 6.21 (Reading the bound on a whole run). Let Z be admissible with $\mathsf{Chk}(Z) = 1$, as in Theorem 6.20.

1. *Round by round.* It is machine-checked, as one Lean statement, that against every prover the protocol before compilation is round-by-round sound (Definition A.1) for Req_Z, the relation under the policy of Z. Each round's error, that of the round in which an OOD seed is sampled included, is at most 2^{-128} of all that round's challenges, proof-of-work bits among them (Proposition 5.14).
2. *The factors.* Among the challenges that pass, a round's error is therefore at most 2^{-96}, since no round has more than 32 bits of proof of work. A whole run has 67,391 rounds. Lean's bound is for a deterministic prover; for a randomized one it follows by averaging over the prover's coins, a step on paper.
3. *Not a figure for the compiled argument.* 2^{-96} is the bound for the rounds with the most proof of work, applied to every round. It is not a bound for the compiled argument, whose error is that of Theorem 5.1.
4. *The relation.* Req_Z is taken at the policy of Z, with the relaxed bound of Definition 4.8; at exact match it implies that each output token is the model's.
5. *The extractor.* The round-by-round statement of item 1 has an extractor for the weights. It is straight-line: the decoding function Dec, applied to the first table and its OOD sample. It extracts the weights alone, and no running time is claimed for it.

Definition 6.22 (Completeness of provers for the three IOPs). Let $\mathbf{P} = (P_{\text{key}}, P_{\text{link}}, P_{\text{req}})$ be provers with abort sets (Definition C.2) for the three IOPs of $\mathsf{FlashProve}_Z$, $\mathcal{Y}$ a set of pairs (A, ξ), and $\alpha \in [0, 1]$. A run of P_* *fails* at given coins and challenges ch if ch lies in the abort set, or if the verifier of the IOP rejects the transcript for some values of its nonces. $\mathbf{P}$ is *complete on $\mathcal{Y}$ with error α* if there are $\alpha_{\text{key}} + \alpha_{\text{link}} + \alpha_{\text{req}} \leq \alpha$ such that, for every $(A, \xi) \in \mathcal{Y}$, all randomness and all OOD seeds of committed tables $\mathbf{T}_0, \mathbf{T}_j$ for A, and every copy number:

1. the run of P_{key} on $\mathbf{T}_0$, of P_{link} on $(\mathbf{T}_0, \mathbf{T}_j)$, and of P_{req} on $\mathbf{T}_j$ and ξ, each with the tables' OOD samples, fails with probability at most α_{key}, α_{link}, α_{req}, over the prover's uniform coins and a uniform ch that passes;
2. at all coins, challenges and nonces, the plain parts of P_{req} hold the copy number.

Theorem 6.23 (Completeness before compilation). [machine-checked] *Let Z be admissible with $\mathsf{Chk}(Z) = 1$. Then the Lean provers $\mathbf{P}_Z^{\text{lp}}$ (Definition 6.14) are complete on $\mathsf{Serv}_Z^{\text{iop}}$ with error 2^{-41} (Definition 6.22).*

Definition 6.24 (Exact simulation up to private aborts). Let $\mathbf{P}$ and $\mathcal{Y}$ be as in Definition 6.22, and $\eta_{\text{key}}, \eta_{\text{link}}, \eta_{\text{req}} \in [0, 1]$. Honest-verifier simulators $\mathsf{Sim}_{\text{key}}, \mathsf{Sim}_{\text{link}}, \mathsf{Sim}_{\text{req}}$ (Definition C.2), the third also given the copy number, are *exact for $\mathbf{P}$ on $\mathcal{Y}$ up to private aborts* $(\eta_{\text{key}}, \eta_{\text{link}}, \eta_{\text{req}})$ if the following holds for every A in range, all randomness and OOD seeds of the tables, with the tables' OOD samples at these seeds, and every copy number; for P_{req}, items 2 and 3 are required at the ξ with $(A, \xi) \in \mathcal{Y}$.

1. The external oracle strings of P_* are the committed tables of its input.
2. Sim_* is exact at every ch that passes and lies outside the private part of the abort set of P_*.
3. A uniform ch that passes lies in that private part with probability at most η_*.
4. If $(A, \xi) \notin \mathcal{Y}$, the abort set of P_{req} contains every ch.

Definition 6.25 (Table experiment with served bits). Let $\mathbf{P}$ and $\mathcal{Y}$ be as in Definition 6.22. In the table experiment $\mathsf{Tab}^{\mathbf{P}}(A, \mathcal{A})$ (Definition C.5) let ς_0 be fixed in advance, let the honest verifier's challenges be uniform among those that pass, let each run of the third IOP be on one encoded request ξ, and let $\mathcal{A}$ attach a bit to ξ, which $\mathbf{P}$ does not read. The next two conditions are required for all answers that $\mathcal{A}$ could be given, whether or not they occur.

$\mathcal{A}$ is *truthful* at A if the bit always says whether $(A, \xi) \in \mathcal{Y}$. $\mathcal{A}$ is *within the life* (K, N) *of one commitment* if it asks for at most K copies, at most one run of the first IOP, on $\mathbf{T}_0$, at most one run of the second for each copy, at most N runs of the third on each copy and none on $\mathbf{T}_0$, and no single column.

Definition 6.26 (Simulated table experiment). Let $\mathsf{Sim}_{\mathrm{key}}, \mathsf{Sim}_{\mathrm{link}}, \mathsf{Sim}_{\mathrm{req}}$ be honest-verifier simulators as in Definition 6.24. The *simulated experiment* $\mathsf{Tab}^{\mathsf{Sim}}(A', \mathcal{A})$ is the experiment of Definition 6.25 with two changes. The tables are committed tables for A'. The output of a run is that of Sim_* on the IOP instance, ch, the symbols that the verifier reads in these tables and, for a request, the copy number; at a request whose bit says that it is not in $\mathcal{Y}$, the output is $\perp$ and no column is read.

Theorem 6.27 (Honest-verifier zero knowledge before compilation). [machine-checked] *Let Z be admissible with $\mathsf{Chk}(Z) = 1$. There are honest-verifier simulators $\mathsf{Sim}_{\mathrm{key}}, \mathsf{Sim}_{\mathrm{link}}, \mathsf{Sim}_{\mathrm{req}}$ such that:*

1. *they are exact for the Lean provers $\mathbf{P}_Z^{\mathrm{lp}}$ on $\mathsf{Serv}_Z^{\mathrm{iop}}$ up to private aborts $(0, 0, 2^{-174})$ (Definition 6.24);*
2. *for all $K \le 710{,}324$ and $N \le 688{,}127$, every assignment of copy numbers to the copies, every ς_0, all matrices A, A' in range, and every unbounded $\mathcal{A}$ that is truthful at A and within the life (K, N) of one commitment (Definition 6.25 and Definition 6.26),*

$$\mathsf{SD}\big(\mathsf{Tab}^{\mathbf{P}_Z^{\mathrm{lp}}}(A, \mathcal{A}),\ \mathsf{Tab}^{\mathsf{Sim}}(A', \mathcal{A})\big) \ \le\ KN \cdot 2^{-174} \ <\ 2^{-135}.$$

Remark 6.28 (Reading the honest-verifier statement).

1. *What the simulators are given.* The simulators are fixed first, and are given neither the weights nor a table's randomness nor a prover's coins. They may depend on the files of Z. In part 1 each is given the verifier's messages, the OOD samples, the columns of the tables that the verifier reads and, for a request, the request with its claimed output and the copy's number (Definition 6.24).
2. *The exception.* Part 1 excepts the challenges at which the prover aborts for a reason that depends on its input, the private part of its abort set. This has probability 0 offline, 0 for the link proof and at most 2^{-174} for a served request.
3. *Part 1 alone.* The OOD samples and the columns are computed from the weights, so run by run this is not by itself a statement that the weights are hidden (Remark 6.10, Not a statement on the weights). In part 2 they are those of tables for A' (in Lean, any matrix A', in range or not).
4. *The budget.* Part 2 is over the life of one commitment: up to 710,324 copies with 688,127 requests each, a budget that the statement fixes (one offline run, one link proof for each copy, requests on the copies alone, asked in any order). What an unbounded observer sees is within 2^{-135} of what it sees of the simulators working from tables of any other fixed weights. There the simulators are also given, for each request, the bit whether it is served. These are the largest numbers that fit the opening budget $t = 44{,}040{,}191$ of each table. Nothing ties a committer or a prover to this budget.

After compilation. The committer and the prover are those of Π_Z^{lp}, compiled from the Lean provers; the verifier is V_Z, and the hash function is a random oracle.

Theorem 6.29 (Completeness after compilation). [machine-checked] *Let Z be in use (Definition 6.11), fixed before H is sampled. In the random-oracle model (Assumption 4.10 and Assumption 4.11), the compiled Lean argument Π_Z^{lp} (Definition 6.15) is complete on Serv_{μ_Z} with error 2^{-40} (Definition 2.12 and Definition 3.11).*

Remark 6.30 (Reading completeness).

1. *In words.* Where the files of Z are in use, at every well-formed request whose claimed output is the model's own result, over the hash function and the committer's and the prover's coins, V_Z rejects the Lean prover's proof with probability at most 2^{-40}.

2. *Two bounds.* 2^{-41} (Theorem 6.23) and 2^{-40} are bounds over two distributions. The first is over the provers' coins and the challenges that pass, at each encoding randomness, each OOD seed and each copy's number, not on average. The second is over H and all coins, and includes the proof-of-work searches that exhaust their limit.

3. *Served instances only.* At a request that is not served, that is, one that is ill-formed or on which the model has no result or another result than the claimed output, Theorem 6.23 and Theorem 6.29 state nothing. In particular, no statement of completeness covers the well-formed requests on which the model has no result (Open problem 8.12). At a real failure the prover returns the empty proof, the same as for a claimed output that is not the model's. That a request is refused is unchecked, as the label is: an empty proof proves nothing, and a prover can return it at a request that it could serve. The compiled verifier rejects it, under every hash function and at all public files (Proposition 6.33). A service that chooses to say "failed" does not prove it.

Theorem 6.31 (Soundness with extractability, openings and extraction). [machine-checked] *Let Z be in use (Definition 6.11), fixed before H is sampled. With β as in Theorem 5.17, let*

$$\varepsilon_{\mathrm{b}}(q) \;=\; \max\Big\{\frac{16\,(q+2^{24})}{2^{128}} + \frac{(q+2^{24}+2)^2}{2^{256}},\; \beta(q)\Big\}.$$

In the random-oracle model (Assumption 4.10 and Assumption 4.11), the compiled Lean argument Π_Z^{lp} (Definition 6.15), with the extractor Ext_Z,

1. *is index-binding knowledge sound for the relation of greedy outputs at μ_Z (Definition 2.7) and has extractability of the commitment (Definition 2.10), each with error ε_{b};*

2. *has no two openings (Definition 2.10) with error $2\,\varepsilon_{\mathrm{b}}$;*

3. *has correctness of extraction (Definition 2.9) with error $3\,(2^{48}+2^{49}+10)^2/2^{256} < 2^{-155}$.*

Remark 6.32 (Reading the theorem on the commitment).

1. *Which parts are single Lean theorems.* Two statements are single Lean theorems to the letter: part 1 for some extractor, and part 3. Part 1 for Ext_Z by name, and part 2, are not. They follow a fortiori from one Lean theorem, which states, for Ext_Z, index-binding knowledge soundness with the error ε of Theorem 5.1, extractability of the commitment with error $\gamma^{\mathrm{lp}}(q) = 3\,(q+2^{49}+10)^2/2^{256}$, and no two openings with error $2\gamma^{\mathrm{lp}}$. Parts 1 and 2 as printed follow on paper: $\varepsilon \le \varepsilon_{\mathrm{b}}$ and $\min(\gamma^{\mathrm{lp}}, 1) \le \beta \le \varepsilon_{\mathrm{b}}$, each by a Lean lemma, and a probability is at most 1.

2. *In words.* Nobody opens a commitment of the compiled committer to other weights than the extractor gives, except with probability about $q/2^{124} + 2^{-100}$, where q counts the hash queries of whoever made the commitment and of the opener together, and no commitment is opened to two different weights except with twice that. For the honest committer, within its budget of 2^{48} hash queries (Proposition 6.33), the extractor gives back the committed weights except with probability about 2^{-155}.

3. *Openings.* To open (Definition 2.10) is to name the weights and one number, r.

4. *What the commitment covers.* The 256-bit commitment covers the first table's root, its OOD sample and the digest dg_Z. It does not cover π_{key}, the copy or π_{link}.

5. *No prover.* Parts 1 and 2 depend on no prover. They hold for every argument that the same compilation yields from any three provers, provided that all leaf salts of its tables have one length, the same whatever the public readings return. Part 3 holds there with error $3(N+2^{49}+10)^2/2^{256}$, for any bound N on the committer's own queries.

6. *The extractor.* Ext_Z is a function, not shown computable.

7. *The programs.* Nothing is stated about the committer of the released programs (Open problem 8.11).

8. *Figures.* The error of Theorem 5.1 is about $q/2^{124} + 2^{-101}$, a different figure from ε_{b}. No figure here is 128 bits.

Proposition 6.33 (Empty proofs, queries and opened columns). [machine-checked] *The following holds of* Π^{lp}_Z *(Definition 6.15).*

1. *Let* Z *be in use (Definition 6.11),* $M \in \mathcal{I}$*, and* r *any commitment randomness. If an instance* (x, y) *is refused at* M *(Definition 6.12), then* $P^{\mathrm{lp}}_Z(M, \mathsf{r}, x, y)$ *outputs the empty proof, for every* H *and all coins.*
2. *Let* Z *be in use. One run of* $\mathsf{Rnd}^{\mathrm{lp}}_Z$*, one of* $\mathsf{Commit}^{\mathrm{lp}}_Z$ *and one of* P^{lp}_Z *together make at most* 2^{48} *queries to* H*.*
3. *With* $\nu_{\mathrm{off}}, \nu_{\mathrm{first}}, \nu_{\mathrm{copy}}, \nu_{\mathrm{on}}$ *and* $t = 44{,}040{,}191$ *as in Definition C.6 and Theorem 6.3,* $\nu_{\mathrm{off}} + \nu_{\mathrm{first}} \le t$ *and* $\nu_{\mathrm{copy}} + 688{,}127 \cdot \nu_{\mathrm{on}} \le t$*.*
4. *For every configuration* Z*,* $V^H_Z(c, x, y, \pi) = 0$ *for the empty proof* π*, every* H*, every* c *and every instance* (x, y)*.*

Definition 6.34 (Zero knowledge with request leakage). Let Π be a commit-and-prove argument with index set $\mathcal{I}$ (Definition 2.6), $\mathcal{W}$ a set of triples (M, x, y), ℓ a function of (M, x, y), and $\zeta\colon \mathbb{N}^2 \to [0, \infty]$. In the experiments of Definition 2.13 let every batch consist of one instance, which is revealed; an environment is $\mathcal{W}$*-valid at* M if $(M, x, y) \in \mathcal{W}$ for each of its instances (x, y). A set of instances stands for the set of all triples whose instance lies in it. In $\mathsf{Ideal}^{\ell}_{\mathsf{Sim}}$ the leakage is $\mathsf{pp}(M)$, N and, for each instance (x, y), the instance and the *request leakage* $\ell(M, x, y)$. The argument Π has *zero knowledge with request leakage* ℓ *on* $\mathcal{W}$*, with error* ζ, if a simulator Sim exists such that, for all $M \in \mathcal{I}$, all environments Env that are $\mathcal{W}$-valid at M, all $q, N \in \mathbb{N}$ and every q-query V^*,

$$\mathsf{SD}\big(\mathsf{Real}_{\Pi}(M, \mathsf{Env}, V^*, N),\ \mathsf{Ideal}^{\ell}_{\mathsf{Sim}}(M, \mathsf{Env}, V^*, N)\big) \ \le\ \zeta(q, N),$$

and in the ideal experiment equal queries receive equal answers. $\Pi[\ell]$ denotes Π with a prover that outputs $(\ell(M, x, y), \pi)$ in place of π and a verifier that ignores the first component.

Theorem 6.35 (Zero knowledge on served instances). [machine-checked] *Let* Z *be in use (Definition 6.11), fixed before* H *is sampled. In the random-oracle model (Assumption 4.10 and Assumption 4.11), the compiled Lean argument* Π^{lp}_Z *(Definition 6.15) has zero knowledge with constant request leakage on* Serv_{μ_Z} *(Definition 3.11 and Definition 6.34), with an error* ζ *such that*

$$\zeta(q, N) \ \le\ 2^{-154} \qquad \text{for all } q \le 2^{64} \text{ and } N \le 688{,}127.$$

Remark 6.36 (Reading zero knowledge on served instances).

1. *In words.* At all files in use, for one commitment, up to 688,127 answered requests and 2^{64} hash queries, and against any verifier within these two bounds: if every request is served, a simulator that gets each request and the largest number of answered requests, and beside these nothing of the weights, makes a view within 2^{-154} of the real one.
2. *The files are not counted.* The simulator is chosen after the files of Z and may depend on them, so what these files reveal about the weights is not counted. The released tools compute the scale file of Z and lines of its parameter file from the weights (Remark 6.1, What the leakage contains). Nor are the requests: each served request holds the model's own output on its prompt, and the simulator gets the request, so what the requests reveal about the weights is not counted. "Nothing of the weights" concerns the experiment alone: there pp and the request leakage are constant.
3. *Membership is not hidden.* Whether an instance is served depends on the weights, and the theorem does not hide it. The verifier does not name its requests: the environment does, as a function of the verifier's view. The environment is chosen after the weights, may depend on them, and is the same in both experiments. That each of its instances is served at M is a hypothesis on the environment; nothing in either experiment checks it. Every such environment is covered, so a verifier can pass a choice of request to the environment through its coins and queries. A verifier that gets a reply to a request that is not served is outside this theorem and inside Theorem 6.37.
4. *One simulator.* One simulator serves all weights, all valid environments and all verifiers, and with probability 1 it gives no two answers to one query. Its running time is not modelled.

5. *The error.* The theorem states that an error function with this bound exists and names none; beyond the two bounds nothing is stated. N counts the answered requests (Remark 6.39); at a served request the answer is a proof or, where a proof-of-work search exhausts its limit, the empty proof.

6. *Scope.* The theorem is for fresh random masks and coins, one commitment, one copy of the table, one request per proof and exact match (Remark 6.42, Scope of the results). The weights are fixed before the hash function is sampled, and the verifier has no hash query before the commitment (Remark 6.45, The model). The prover and the committer are Lean definitions (Remark 6.43); the view is that of Remark 6.39, item 4.

7. *Which requests.* At every valid μ some request is served (Proposition 3.12(a): at constructed weights; more at Z_0: Remark 3.13). At the model parameters of Z_0 there are valid weights, constructed too, at which no instance can be served (Proposition 3.12(b)): at such weights the theorem concerns the commitment alone. No instance at a real model's weights is proved to be one that can be served (Remark 3.13).

Theorem 6.37 (Zero knowledge at every instance). [machine-checked] *Let Z be in use (Definition 6.11), fixed before H is sampled. In the random-oracle model (Assumption 4.10 and Assumption 4.11), the compiled Lean argument Π^{lp}_Z (Definition 6.15) has zero knowledge with request leakage bit_{μ_Z} (Definition 6.12) on the set of all instances (Definition 6.34), with an error ζ such that*

$$\zeta(q, N) \;\le\; 2^{-157} + 688{,}127 \cdot 2^{-174} < 2^{-154} \qquad \text{for all } q \le 2^{64} \text{ and } N \le 688{,}127.$$

Corollary 6.38 (Zero knowledge with the label shown). [machine-checked] *Under the hypotheses of Theorem 6.37, the argument $\Pi^{\mathrm{lp}}_Z[\mathsf{lab}_{\mu_Z}]$ has zero knowledge with request leakage lab_{μ_Z} (Definition 6.12) on the set of well-formed instances (Definition 6.34), with an error that satisfies the bound of Theorem 6.37.*

Remark 6.39 (Reading zero knowledge after compilation).

1. *In words.* At all files in use and every request, well-formed or not,

   - for one commitment, on one copy of the table, up to 688,127 answered requests and 2^{64} hash queries,
   - and against any verifier within these two bounds that has no hash query before the commitment (Remark 6.45),
   - a simulator that gets each request, the largest number of answered requests and, of the weights, only one bit for each request (served or refused)
   - makes a view within $2^{-157} + 688{,}127 \cdot 2^{-174} < 2^{-154}$ of the real one; each answer is a proof or the empty proof.

   As a corollary (Corollary 6.38), for a service that chooses to say "failed", the same holds at every well-formed request with one label (served, failed or other tokens) given to the simulator and shown with each proof.

2. *The simulator.* The simulator is fixed before the weights, may depend on the files of Z, and answers the hash queries in the hash function's place. Its running time is not modelled. The public parameters of the index, pp, are constant for this relation (Definition 3.2), so they carry nothing about the weights.

3. *The leakage.* The notion is only as strong as ℓ is small: if Π has zero knowledge with a constant request leakage, then $\Pi[\ell]$ with $\ell(M, x, y) = M$, whose prover outputs the weights, has zero knowledge with request leakage ℓ.

4. *The commitment and the view.* The commitment in both experiments is the 256-bit commitment of Definition 6.15, not the whole key. The view consists of this commitment, each answered request with what the prover outputs for it, the query–answer trace and the coins of V^*.

5. *The two bounds.* N counts the requests that the prover answers, by a proof or by the empty proof. Elsewhere in this paper an answered request is called a reply. Nothing in the prover counts the answered requests, and beyond the two bounds nothing is stated.

6. *The error.* 2^{-157} bounds the error of the compilation at these numbers; 2^{-174} is the bound on private aborts of Theorem 6.27, counted once for each answered request. A proof-of-work search that exhausts its limit is part of what is simulated: it gives the empty proof at a served request.

7. *Derivation.* Theorem 6.37 follows from Theorem 6.35 and part 1 of Proposition 6.33 by a lemma on every commit-and-prove argument. Suppose that every served instance lies in the relation and that the prover outputs one fixed value at every refused instance. Then zero knowledge with constant request leakage on the served triples gives zero knowledge with request leakage bit on all instances, with the same error function: the simulator, given "refused", outputs that value. With the ζ of Theorem 6.35 this gives the bound 2^{-154}. The sum in the display is the bound of a second Lean theorem, whose proof takes two steps of the same form at that number. Taken at that number, the proof term of Theorem 6.35 gives the sum as its bound too; no named Lean theorem states this, and the step has not been compiled. The converse, from Theorem 6.37 to Theorem 6.35, is not proved.

Remark 6.40 (Relation to index privacy). Zero knowledge with request leakage (Definition 6.34) differs from index privacy (Definition 2.14) in three respects: each batch is one revealed instance; the set $\mathcal{W}$ need not lie in the relation, and in Theorem 6.37 it contains instances that are not served; and the simulator is given the request leakage. In Theorem 6.35 only the first respect is left: its environments are those valid at M for $\mathcal{S}_M$ the set of instances served at M, and a constant leakage carries nothing. It is not index privacy for batches of at most one either: an empty batch and a hidden instance, which Definition 2.14 admits, are not covered; for this form Lean has the other direction alone, and no Lean theorem derives index privacy from it. That zero knowledge in the forms of Theorem 6.27 and Theorem 6.37 is index privacy is not stated: no Lean theorem derives index privacy of Π^{lp}_Z from Theorem 6.37.

Remark 6.41 (From the IOPs to the argument).

1. *Limits carry over.* Completeness and zero knowledge after compilation follow from the two results before compilation by a conditional theorem on the compilation, so every limit of these two holds for them as well.

2. *Zero knowledge.* The Lean proof of Theorem 6.37 passes through Theorem C.25 on the whole-key argument: inside its proof all twenty conditions are met by the Lean provers, at one copy of the table. No Lean theorem states that by name; we have it from a reading of the proof's calls (Open problem 8.13).

3. *From the whole key to 256 bits.* The statement of Theorem 6.37 itself concerns the 256-bit commitment. The step from the whole key to that commitment is made inside the proof, in two parts. The whole-key argument is first carried from its separate oracles at fixed $(\mathsf{dg}, \mathsf{par})$ to the one oracle H, with the readings computed from it; this leaves the bound unchanged, and it is used as a bound for all values of $(\mathsf{dg}, \mathsf{par})$. Then a verifier against Π^{lp}_Z is run by a verifier against this argument, which computes $H(\mathsf{code}(\mathsf{k}, \mathsf{dg}_Z))$ from the key by one query of its own and forms each proof from the key and a proof of that argument. So the bound of Theorem C.25 is used at $q+1$ queries in place of q, and no term is added.

4. *Completeness.* Completeness does not pass through the reduction of Remark 6.49.

Limits common to these results.

Remark 6.42 (Scope of the results). Theorem 6.23, Theorem 6.27, Theorem 6.29, Theorem 6.35, and Theorem 6.37 and part 3 of Theorem 6.31 are for fresh random masks and coins, one commitment, and one request per run and per proof, not for batches. The theorems after compilation hold at the exact-match policy, and Theorem 6.29, Theorem 6.35, and Theorem 6.37 for one copy of the table. Before compilation exact match is not a hypothesis; an output that the policy of Z would accept and that is not the model's own is not served, and nothing is stated about it. Every admissible configuration has the layout file of Z_0 and its parameter file except for eight numbers, with $\theta \geq 2^8$ (Definition 3.8), so every result that asks for an admissible Z is at the default configuration's size.

Remark 6.43 (Definitions in Lean). The results concern the Lean provers exactly as written, default values included. By their definition in Lean, each prover's coins are uniform and fresh in each run, and, if $(A, \xi) \notin \mathsf{Serv}^{\mathrm{iop}}_Z$, the abort set of $P^{\mathrm{lp}}_{\mathrm{req}}$ at (A, ξ) contains every list of challenges. That the provers are not

trivial follows from the results together (Remark 6.46, The two together). These are definitions, not programs that run: for instance, the model's run on a request is obtained by a non-constructive choice. Nothing is stated about the released programs (the C++ verifier and committer, the CUDA prover).

Remark 6.44 (Not hidden).

1. *The bit before compilation.* Before compilation one bit per request is not hidden: whether the request is served, which depends on the weights. The simulator is given that bit, and where a request is not served the prover shows nothing. A bit says whether the claimed output is the model's result on that request, so over many requests the bits tell what the model answers.
2. *The bit after compilation.* After compilation one bit per request is revealed (served or refused). The bit depends on the weights, and it is declared, not hidden: for every request of its choice a verifier learns whether the claimed output is the model's result. (It reads the bit off the reply: a refused request gets an empty proof; so, with small probability, does a served one, where a search for a nonce runs out.)
3. *Many requests.* So over many requests, chosen adaptively, the verifier learns what the model outputs on inputs of its choice: whoever tries each possible token for a prompt learns the model's next one from this bit alone, unless all are refused.
4. *What the bit keeps back.* Against three labels the bit hides one thing: whether a refusal means that the model has no result or that its result is other tokens. The bit hides that only from a verifier that asks few requests. A request's window (the 1,024 tokens that the model's run reads: the prompt, the claimed output, then zeros) can itself be sent as a prompt, with one more token as the claimed output, which the run does not read: a verifier that tries every value of that token and is refused each time learns that the model has no result on that window, and one that is served once learns that it has one (argued from the definitions; no theorem states it). Each such search costs up to 202,048 of the 688,127 replies: one for each token of the vocabulary (Definition 3.1).
5. *Further.* A true output is refused if the model has no value further on in the window. No statement bounds what many bits reveal together. In the corollary one label per request is revealed (served, failed or other tokens), and it depends on the weights.

Remark 6.45 (The model).

1. *Order of choices.* In all results after compilation the files of Z are fixed before the hash function is sampled; in Theorem 6.29, Theorem 6.35, and Theorem 6.37 and in part 3 of Theorem 6.31 so are the weights. In Theorem 6.35 and Theorem 6.37 the verifier has no hash query before the commitment.
2. *Second hash function.* A second, fixed hash function inside the format, H', is not treated as random (Assumption 4.11).
3. *Running time.* The simulators' running time is not modelled.
4. *The configuration and timing.* Neither completeness nor zero knowledge says anything about what the files of Z themselves reveal about the weights, or about timing.
5. *Figures.* The figures on zero knowledge are statistical distances (2^{-135}, 2^{-154}) or bounds on a probability (2^{-174}), not a security level.

Remark 6.46 (The two together). A prover that sends fixed messages and aborts at every request that is not served, or whose abort set has a public part that contains every list of challenges, also satisfies a statement of the form of Theorem 6.27. Completeness of the same provers excludes this, so the two results are to be read together. One Lean theorem, at every Z in use, states round-by-round soundness (Remark 6.21, item 1), Theorem 6.23 and part 1 of Theorem 6.27 of one term for $\mathsf{FlashProve}_Z$ and one triple of provers. That term is the one from which V_Z, the verifier of Theorem 5.1, is compiled. Likewise Theorem 6.29, Theorem 6.31, Theorem 6.35, and Theorem 6.37 concern one committer, one prover and one verifier, those of Π_Z^{lp}; in Corollary 6.38 the label is added to the prover's output, and the verifier ignores it (Definition 6.34).

Remark 6.47 (Definitions of the properties). The definitions of this subsection restate definitions of the Lean library. Before compilation, the definitions of completeness and of honest-verifier zero knowledge are this

library's own and follow no published definition word for word; after compilation Theorem 6.29 and Theorem 6.35 are stated through `CompleteOn` and `ZeroKnowledgeOn`, the texts of `Complete` and `ZeroKnowledge`, each with one condition more, that the request can be served (Definition 3.11); with the condition that the output is the model's, these are the served requests; the forms of Theorem 6.37 and Corollary 6.38 are the library's own. The definitions before compilation, and the Lean statements of Theorem 6.23, Theorem 6.27, Theorem 6.37 and Corollary 6.38 other than the six top theorems and the final statement's six fields, have not been independently reviewed ("Who wrote and checked this").

Remark 6.48 (How they are checked). Of the library's six top theorems the independent checker has accepted `sound` (Remark 6.16), `repliesFit` and `refusedHandsOutNoProof` (Proposition 6.33) by name, over a build of the released Lean files; over such a build the three on binding, completeness and zero knowledge have been checked by Lean's kernel alone, and they rest on equations between closed terms that the kernel decides by computation. For those of the Lean theorems behind them that are named in `lean/CHECK-RESULTS.md`, Lean prints no axiom beyond its three standard ones. The reference run (the 109 calls of Appendix H.3) has been made over the released Lean files, and the independent checker has been run, on the soundness theorem's export and on those of the theorems named at this remark's head, as the exporter writes them over the released Lean files' build; what it accepted: `lean/CHECK-RESULTS.md`; of the theorems of this section only those named at this remark's head are among what it accepted. A whole run of the library's full check over all but one of the released Lean files (Remark B.17) passed at every step; its last line says that no theorem got a verdict on binders or on `Classical.choice` (the whole line, and what the check does not cover: `lean/CHECK-RESULTS.md`; Remark B.17). What these checks are: `lean/CHECK-RESULTS.md`; the limits of the independent checker's pass, and the version of Lean: Remark B.17. The Lean modules of Theorem 6.23, Theorem 6.27, Theorem 6.29, Theorem 6.31, Theorem 6.35, Theorem 6.37, and Corollary 6.38, of Proposition 6.17, part 1, and of Proposition 6.33, parts 1 and 4, import the separate library of Remark 5.7; by import lines at least one of the modules on which the re-checker stopped lies beneath each of them. By a walk over the constants of a build of the released Lean files, proofs included (a program's finding, no theorem; it covers twenty-three modules), the Lean theorems `finalSoundAndBinding_value`, `finalHandsOutNoProof_value` and `finalRefusedHandsOutNoProof_value` reach no constant of these modules; each of the theorems on completeness and zero knowledge listed above reaches at least one, but for Theorem 6.35, whose one Lean theorem was not among those walked; the modules of a third library, on which these theorems rest too, were not put to the re-checker; of the other Lean theorems that the listed statements cite it is not known.

6.6 A reduction for completeness, and what is open

Remark 6.49 (Completeness in Lean). Besides the results of Section 6.5, on completeness the following is machine-checked.

1. *A reduction.* Let μ be the model parameters (Definition 3.6) that the files of Z fix. If thirteen conditions hold at (Z, α), then V_Z, with the committer and prover built from a proving side given as a parameter, is complete on Serv_μ with error α: an output that the model itself gives is rejected with probability at most α. Three conditions are on the files: they parse, Z is admissible, and $\mathsf{Chk}(Z) = 1$.

   Ten are on the proving side. One, the largest, asks at every instance true under Z's policy, whether it can be served or not, that each of the three IOP verifiers accept its prover's run, at OOD seeds that are not excluded and at every challenge list that satisfies each proof-of-work check, outside an abort set and an exceptional set; the probabilities of the three sets enter α. It also asks that each prover's input oracles encode the index, that a challenge list lie in the union of these two sets whenever zeroing its proof-of-work bits gives a list in it, and that the request prover send the copy number handed to it.

   Three fix how the messages and coins of the key and request provers fit the byte format, and five constrain a salt length, a tree depth and the three search limits. The last asks that α dominate these probabilities, the excluded seeds' twice, and the failure probability of the searches, added up. The only acceptance assumed is that of the three IOP verifiers.

2. *Conditions that hold.* The three on the files, at Z_0 and at every Z that differs from it only in the policy, in its three numbers, within the bounds of Proposition 5.6 (Remark 5.7, How the proposition is checked). For admissible Z with $\mathsf{Chk}(Z) = 1$: what the largest condition asks of the link proof, for P_{link} with

empty exceptional set and an abort set of probability 0 (Theorem 6.9); the three search limits, at 2^{b+6} attempts per b-bit proof of work, where the searches add at most $71 \cdot 2^{-92}$ to α; and the salt length and the depth, which are met by choice. For the same Z, in the whole-key argument at the fixed digest 0^λ, wherever $\pi_{\text{link}} \neq \perp$, the verifier of $\mathsf{IOP}_Z^{\text{link}}$, not the compiled one, also accepts P_{link} at the challenges derived by hashing (Remark C.30; Remark 5.7).

3. *Searches succeed.* In the whole-key argument with constant-message provers and at least 2^{40} attempts per search, which is more than in item 2, Corollary 6.7 bounds $\Pr[\pi = \perp]$ for each kind of proof π; on acceptance it is silent.

Whether the Lean provers (Section 6.5) meet the ten conditions is not stated here. All thirteen are shown together only for idle provers, which always abort; then $\alpha \geq 3$ and the bound is vacuous. In the reduction the parts of the offline and the online prover are not shown, so that it gives no bound below 1. The proving side is a parameter, not that of Π_Z; the reduction does not bound its queries, and no statement relates it to the released programs. Outside Serv_μ (Definition 3.11) nothing is stated. For no valid μ is Serv_μ empty (Proposition 3.12(a)), but its members are exhibited at constructed weights (more at Z_0: Remark 3.13). For one μ there are valid weights M, constructed too, for which Serv_μ contains no triple (M, x, y) (Proposition 3.12(b)); by a further Lean theorem that μ is the model parameters of Z_0.

Completeness and zero knowledge of the whole protocol, with batches, are open problems (Open problem 8.12 and Open problem 8.14).

7 Evaluation

7.1 Set-up

The implementation has a committer, a prover and a verifier in C++ and CUDA (the *C++ verifier* (Limitation 8.1, Lean verifier only)), an integration with vLLM [KLZ23], and the command `lean-verify`, which evaluates the executable form (Definition 5.4) on the prover's proof files. Weights are *synthetic*, generated by the implementation, or *public*: a third party's mostly 4-bit quantization of Llama 4 Maverick [Uns25], not Meta's checkpoint, converted to the integer model (Construction D.3).

Timed runs use one H100 (80 GB) of an eight-GPU host with 96 CPU cores: in the committing and serving runs the programs were bound to 24 of these cores, all of the GPU's NUMA node, under a limit of 26 cores and 463.9 GB of memory for the container; as reported, when a machine was assigned the host's other seven GPUs were in use by other tenants; whether the cores were reserved to a run, and what ran beside it later, the records do not tell; the prover's process held at most 257.1 GB of host memory while serving at 24 requests a batch on the public weight files (269.7 GB in its container, file cache not counted); on synthetic weights both were higher (265.5 GB and 278.3 GB), and at 64 a batch on the public files the container reached 278.3 GB; the container held more with the middleware's check beside the prover, and both held more in runs not on the released programs† (Appendix H.2); the committer's process held at most 286.9 GB (299.6 GB). A largest size or peak is rounded up, the lower end of a range cut.

Each request is proved on a window of 1,024 tokens and receives its own proof. Lean's statements on completeness and zero knowledge do not cover batches. A *run* is one commitment and one or two executions of the prover on one host; where there are two, its figure is their mean. *Masking on* means that the committer samples the secret seed and no test option is set (Appendix H.2); fitted and reduced configurations are defined in Section 3.3.

A result with no mark is of a run that Appendix H.1 names (the runs of `lean-verify`: the next paragraph): a run on the *released* programs, which are the committer, the prover and the C++ verifier as the release's sources build them (the programs that made these runs were built from sources that differ from the released ones in comments, in the names of identifiers and files, in quoted texts that hold such a name and in the wording of messages, and in nothing else: Appendix H.1, "Which programs ran"); the *timed runs* are among these, and have a part of their own there. A result with no mark that Appendix H.1 does not name is to be taken as not on the released programs (Appendix H.1, "What this paper tells of other programs"). A dagger (†) marks a result *not on the released programs*, of any *other programs*, which is not shown to hold for the released programs; a double dagger (‡) marks the test programs of the second and the third test of hiding

(Table 6), built from the sources of other programs (two of the fault injection's three from a copy of these in which four files differ: Section 7.5), which do not ship.

The runs of `lean-verify` told here were made over Lean files that differ in code from the released ones, but for the calls of the reference run and of the second set of 41, which were made over the released Lean files (Appendix H.3). Those marked † used scripts that differ from the released ones, in code too (one lacked a bound on the length of a number that it reads), and a parser compiled from sources that differ from the released ones, in code too (its hashing class among what differs; the rest not compared file by file). The calls of a separate build of the executable form are another thing (Appendix H.3). Appendix H gives the conditions, the provenance and the limits of each table (Limitation G.7, No independent review).

7.2 Cost

Table 3: Mean, sample standard deviation, least and most of n runs. Rates in tokens per second, cut, never rounded up; times in seconds. The rate with whole outputs checked cannot be set against the steady rate at exact match; its difference from its control shows nothing.

Quantity	n	Mean	s.d.	Least to most
Rate at exact match, batches of 24	3	111.23	0.92	110.20 to 111.99
time per request (a batch's time over 24)	3	9.206	0.077	9.143 to 9.292
rate with start-up counted	3	103.60	0.84	102.65 to 104.24
Rate at exact match, one request in a batch	3	17.41	0.15	17.24 to 17.51
Rate, whole outputs checked under a clipped gap, batches of 24	3	108.67	2.8	105.41 to 110.41
checked output tokens alone (calculated)	3	39.1	1.0	37.9 to 39.7
the same requests, last position alone checked	3	108.73	2.7	105.57 to 110.37
Committing, once: the first table and one copy (seconds)	3	2,323.77	18	2,307.45 to 2,343.76
Verifying one proof, C++ verifier, called with its option for one thread, on the prover's machine (seconds)	3	21.14	0.03	21.11 to 21.17

Table 3 is for the public weights, with masking on (read off the runs' own summaries, not tested: Appendix H.2); its unmarked rows are of the timed runs. At exact match a proof covers the model's run over the whole prompt and checks one output token, the one that follows the prompt. Rates are steady-state rates: they count every prefill token of a window, checked or not, and exclude start-up and committing. A steady rate counts every interval between two releases. With start-up counted, in batches of 24 at exact match, each of the three runs passes 100 tokens per second (102.65 to 104.24); of the three sessions with whole outputs checked, and of their control, one each stays under it (99.33; 98.12). At the steady rate the slowest two executions gave 110.07 and 110.33.

Three runs made side by side say little about other times and machines: three runs of the same kind on other programs†, on another day on other machines, gave 105.3 ± 3.9 tokens/second (100.85 to 108.06); there each service's last interval is not counted; whether the programs, the machines, the day or that rule made the difference is not shown (Appendix H.2). A batch is taken from the queue just before the responses of the batch before it are released, so a request waits a little longer than the time between two releases: 220.7 to 225.9 seconds from a batch's start to its release, from the second batch on, in each case 1.6 to 2.3 seconds more than the interval that ends at that release. That wait ends at a proof on which no verifier has run. Of what ships, the middleware of the vLLM integration hands on no response that has a proof before the C++ verifier has accepted every proof of its batch (Limitation G.16); that time is not in these figures. In one run of its own on the released programs, none of the timed runs, on stored proofs of one batch on synthetic weights, with no inference engine in front and no prover running beside it, on 24 CPUs of a 32-CPU machine, the middleware's check of the 24 proofs, at its default there of six checks at once with one thread each, took 94.0 seconds from the hand-over of the proofs to the batch's release; no spread is known. Beside a serving prover on the same 24 CPUs of another machine, in one more run on synthetic weights and on stored proofs, not that prover's own, at the middleware's default, six checks at once with one thread each, each of five checks took 88.74 to 89.15 seconds (alone on that machine the check took 92.1 seconds before the services and 90.3 after

them; what else ran there was not held the same), and the prover's steady rate was 109.64 tokens per second, against 110.11 in that run's second service with no check beside it; in the timed runs on synthetic weights two such services part by 0.1 to 0.7 % with nothing beside them, so no slowing can be told from one run.

On other programs†, in the runs that answer to the three rows of Table 3 under a clipped gap (105.38, 37.9 and 105.26), the rates differed, row by row, by 8.0 to 9.0 % (the most over the least), and no cause is shown. The difference between the means there and in the timed runs is matched by the control's and is smaller than the spread there; the timed runs had the lighter CPU load; no effect of the programs is shown.

The configuration of these runs is the default one (the clipped-gap rows and their control apart from their policy) as fitted to the public weights; the three configuration files of one set of timed exact-match runs on these weights pass the scope test for an admissible configuration, which Theorem 5.1 requires, by one evaluation of a test that does not ship (no theorem); whether they are byte for byte the files behind the runs reported here is not shown, and the parameter checks were not evaluated at them (Remark D.5, No real weights shown). The jobs' own calls of the C++ verifier, behind the timed stretches, accepted every proof; in these jobs the executable form was given none. Under the clipped-gap policy the accepting last line is `WITHIN LIMITS`, not `YES`.

Batches of 64 are not to be relied on with such a GPU: on synthetic weights, in the timed runs, one call of the prover on 64 requests and one attempt at a service of five such batches, one run each, ended at a failed CUDA call, the service in its first batch, with 0 of 64 and 0 of 320 requests proved; every attempt at 64 on synthetic weights on record failed, eight in all, six of them on other programs† (Appendix H.2). The three runs at 64 on the public weights completed, with all 960 proofs accepted and at most 80,629 MiB of 81,559 MiB in use. At exact match, in batches of 24, a proof file has 18.74 to 18.81 MB.

7.3 The executable form on the prover's proofs

At the format's hash function, at the default configuration's size and on synthetic weights, the executable form accepted (by `lean-verify`; four of them by a separate build of the executable form alone, Table 4), in calls that ran to their end, every proof from an honest run of the prover counted here. At exact match 363 were made by one prover built from other sources†, the part handed over of the 723 that it made (not the 723 that the three timed runs on synthetic weights made, 241 a run): 360 with one output token (96 requests of one shape from one list, each proved under the same three commitments, 24 of them twice under each) and three of one stored instance.

Eighteen were made by other provers than that one: seven of that instance, nine of three others, and two beside the malicious proofs of the true claim of Appendix H.3, one of them of one of those three.

Under the clipped gap of Z_1 it accepted one by that prover† and four by others, as reported, all of one further instance.

A second set of 41 honest proofs, from the timed runs' own files on synthetic weights, counted apart from the 109 calls of Appendix H.3, has been put to the Lean verifier: each of three runs' files holds proofs numbered 000 to 119 and one more; taken were that one and 000, 009 and so on, to 108 in two runs (13 each) and to 099 in the third (12): $3 + 13 + 13 + 12 = 41$; it accepted each, at exact match, one call each. These are other files than the 41 of the 363 that are among the 109 calls. Like the 109 calls, the calls on them were made over the released Lean files; the command had accepted each of them before, over Lean files that differ from the released ones; 7 of their 14 requests are among those of the 47 honest proofs of the 109. Appendix H.3 gives their provenance.

Under the clipped gap `lean-verify` has met honest proofs alone; a separate build of the executable form rejected the two proofs of false claims of Table 4.

`lean-verify` and a separate build of the executable form together rejected all 26 proofs of claims false by the run's own policy (Table 4) — 24 and two — all from one test program, claims fixed beforehand, each rejected where the proof file is read.

With the form of the command marked † in Section 7.1, the library built in advance, three calls† on one stored proof, made one at a time for the timing, one after another on one shared CPU machine (an Intel Xeon Platinum 8375C, 64 CPUs), each accepted it, in 1,393 to 1,440 seconds, with at most 14.13 GB of memory in a call's largest process, in these three calls. What an evaluation trusts, how narrow these proofs are, and what a reader can repeat: Appendix H.3.

Table 4: Evaluations of the executable form. These calls were made over Lean files that differ from the released ones, but for the calls of the reference run and of the second set of 41, which were made over the released Lean files (Appendix H.3). A dagger marks one of the other programs: the one that made the input or, where it stands on a verifier, that verifier.

Input	Decisions
Honest proofs at exact match†	18 of 18 accepted, 2 of them by a separate build of the executable form† alone
Honest proofs under the clipped gap of Z_1†	4 of 4 accepted, 2 of them by a separate build of the executable form† alone
Honest proofs by one prover built from other sources†, as reported, at exact match	363 of 363 accepted (360 with one output token, of 96 requests, and three of one stored instance, each covering 1,001 positions; the C++ verifier of those runs accepted each first, as reported)
The same under the clipped gap	1 of 1 accepted (an acceptance that says less; not blind)
Proofs of false claims†: 24 at exact match, 2 under the clipped gap	26 of 26 rejected, the 2 under the clipped gap by a separate build of the executable form† alone
Altered inputs†	by a separate build of the executable form† alone: 32 of 41 rejected; 9 accepted, all altered proofs of claims true by the run's policy, altered at Merkle-path entries that the verifier's walk does not take, by a reading (two of them had been expected to be rejected: `lean/GAPS.md`, gap 14; Remark B.12)
`lean-verify`†: 16 honest proofs at exact match, 2 under the clipped gap, the 24 at exact match, 11 malicious proofs of the true claim	16 and 2 accepted, 34 rejected, one unparsed, without a decision
Also to a C++ verifier†, besides a separate build of the executable form†: 43 files	the same decision on each (at 3, the C++ verifier's as reported)
Also to a C++ verifier built from other sources†: the 42 (the 41 below and the one file the form could not read), the 22 malicious proofs of the true claim of Appendix H.3 with their two honest proofs and two controls, the 364 and three controls	the same decision wherever the form gave one (432 of 435 sets)
Also to another C++ verifier†: the 41 (seven honest proofs, the 24, ten malicious proofs of the true claim; no altered inputs) and the nine proofs of three instances	the same decision on each
Four matched alterations of one honest proof†, to both verifiers	the same decision at 1; at 3 a C++ verifier† rejected the raw proof, a separate build of the executable form† accepted the parsed one
Explicit committer (Definition 5.16; what ran: Remark 5.21) against the committing program†, reduced configuration	digests, root, OOD answers and commitment equal in two runs; one public seed, synthetic weights, nothing else compared

7.4 What has run under each policy

Exact match requires that the integer model itself be served: of the 96 greedy outputs of each of two floating-point engines (16-bit, FP8), none matched the integer model's greedy token at every token position (scored without proofs†). A vLLM plug-in is designed to serve it, one request at a time: not on the released programs†, on the public weights and one H100, beside a helper process, it took 0.58 to 0.63 seconds per token after prompts of at most 5 tokens, and 0.58 to 1.46 over all 48 outputs. Clipped-gap parameters fitted on honest greedy outputs accept most outputs with one token replaced (Table 5) and most sampled outputs, 86 of 96 at temperature 0.7 and 85 of 96 with top-p 0.9 (scored likewise, 16-bit fit) (Limitation 8.8, Clipped gap is weak). Serving and proving have not run together as one service (Appendix H.4).

Table 5: Runs by policy, on the public weights. The runs with proofs (the first row's on other programs†, the second row's among the timed runs) use the honest prover and test neither soundness nor weight hiding.

What ran	Outcome
Exact match: vLLM serves the integer model and is stopped; the prover then proves every served token; three runs†	48 of 48 outputs, to the same 16 synthetic prompts in each run, received a proof that the C++ verifier accepted
Clipped gap: stored outputs of a floating-point engine (Table 3)	all proved; the 48 outputs were selected as meeting the policy: no acceptance rate
Scored, no proofs†: one token replaced by the second choice; by a random token	28 of 32 and 24 of 32 meet the policy (16-bit fit, FP8 fit); 27 of 32 and 21 of 32

7.5 Tests of weight hiding

Table 6: The five tests behind the judgement of Section 7.5, one run of each: at the reduced configuration, on synthetic weights, under exact match, on selected cases, so no rates (four further runs of the first, on a real model's weights: Appendix H.5). None of the five searches for a leak: they test that the masks are there and that a loss is noticed. Every "stopped" was observed in a test program or, in the fourth test, in a program that held test-only code (Appendix H.5).

Test	Outcome
OOD answers recomputed by a separate program	all 1,536 answers as prescribed by the rules that the test carries; no pad occurs twice or is 0. That the pads are random it does not show.
Injected losses of the OOD answers' masks‡, at all-zero weights	8 of 8 (none a loss of a single pad) stopped the committer, all by one self-check; with it disabled†, at a parameter file that counted no proof of work, an unmasked commitment passed every other check
Fault injection‡: 295 selected losses of random values	GPU memory: 130 of 130 stopped, 122 by one self-check alone. CPU memory: 123 of 165 stopped, 12 changed no byte, 30 unnoticed
Riskier losses†, outside that selection (its setting: as reported)	263 of 263 stopped; 9 of 9 at neighbouring samples, which no self-check covers, unnoticed
Both shipped audits, with the provider's secrets	accepted honest files; in trial calls, which make the same checks, rejected three false secrets. They compare 7.66 %† (at the default configuration's size) to 33.00 % of the bytes by value, and no value that depends on the weights (run by run: `docs/limits.md`).

Section 6 concerns the protocol; for the programs, their goal, weight hiding (Definition H.1), is tested. We judge that weight hiding likely holds at the default configuration, in honest runs, if the provider keeps its operating conditions and the judgement's own conditions (listed in `docs/limits.md`, not in this paper) hold. This judgement is given of the released programs: the first and the fifth of its five tests (Table 6) have run on them, under exact match, with the outcomes that Table 6 and Appendix H.5 tell, both at the reduced configuration on synthetic weights and the first at the default configuration too, once on synthetic weights and once, as fitted to them, on the public weights; the fifth ran once. The second and the third ran on test programs and the fourth on other programs, all three built from sources that differ from the released ones. The one reading of whole files for hiding was of a text from which the released one differs. It took in 384 files: all that a production build of the committer and the prover from that text reads, but for the C++ verifier's own 36. The released sources differ in code in 68 of the 384, and in 136 more in comments, names and the wording of messages alone; two more of the 384 are not in the released sources; in the 68 the differences in code were read for hiding, by difference alone, apart from renamed names and reworded messages, and the files were not read whole again (Appendix H.5). That judgement rests on the design, what is proved in Lean for the protocol (Section 6; what that leaves out: Limitation 8.2 and Limitation 8.3), code review and tests; it is not a proof, it does not cover a real model's weights, and neither the programs nor the judgement has had an independent review. Only one short search has looked for a leak: in two sets of honest published files,

three proofs among them, at the reduced configuration, not on the released programs†. It made several looks: at the OOD answers, for guessable seeds, and over the opened columns of the stored tables; of the proofs it studied nothing else by value. It found nothing, which is worth little: on the first of these sets the first look that it made cannot see any fault of a pad. Beyond that search no proof has been searched for a leak, and no file at the default configuration.

None of that judgement's five tests searches for a leak: they test that the masks are there and that a loss is noticed. The tests behind it ran at the reduced configuration, on synthetic weights; only one of the five has also run on a real model's weights, once on the released programs and three times on other programs†, at the default configuration as fitted to those weights. That judgement reaches, on less evidence, a configuration that differs from the default one in a clipped-gap policy's three numbers alone. Its whole scope, its conditions, and what is not shown or untested: Appendix H.5.

A *self-check* is a program's recomputation of masked values; an *audit* is a shipped tool that recomputes part of the published bytes from the provider's secrets. Of the fault injection's three test programs two were made from a copy, in which four files differ, of the sources of other programs. In Table 6 losses were unnoticed in three places: 30 of 165 injected losses of random values in CPU memory; 9 of 9 at samples that no self-check covers; and, with one self-check disabled, at a parameter file that counted no proof of work, a commitment without masks, which passed every other check.

In one trial, not on the released programs, on a weight file with a tensor of 16 bits, which that judgement does not cover, a party that could write that file but not read it learned the magnitudes of two weights of their choice (Limitation G.18, One trial with such a file).

7.6 Negative tests of the verifiers

Table 7: Negative tests of the C++ verifier, on synthetic weights at the small example's setting unless stated.

Test	Outcome
A malicious prover†, at the reduced form of the default configuration	113 of 113 falsifications rejected, each beside the true output
A malicious committer† alters a committed table at 1, 16 or 256 of 8,388,608 positions	24 whole calls accepted, in each of two tests, with the honest request and output
One table† altered at 257 to 524,287 positions	45 of 80 commitments accepted, each within the unique-decoding radius of the unaltered table
Not the verifier program: test programs† that call the shared C++ library beneath it directly	false claims accepted, against the library's stated contract, by at least two ways (Section 7.6)
Altered copies of honest files, altered calls; public weights, default configuration	240 of 240 and 9 of 9 rejected; where the altered bytes lie is not reported: no evidence on the check of the OOD sample

In no reported test did the C++ verifier program accept an output that violates the run's own policy (Table 7 and Table 4); most of these tests ran not on the released programs. On proof files the released verifier's record is chiefly this (the whole of it: Appendix H.1): it rejected all 57 that malicious provers made at the default configuration's size, at exact match and under one commitment, on synthetic weights, and that were put to it, 24 of them claiming a false output and 33 the true output with a falsification inside; and, at the small example's setting, the 228 acceptances on 57 kept sets of files in which an edited committer had altered one stored table (at 1 to 131,072 of its 8,388,608 positions; the four full checks on each set; six of the sets a second making of six others) all carry a true claim, by a reading, not machine-checked (the release's `docs/limits-all-rows.tsv`, rows LIM-63.1 and SO-15.7; Appendix H.1). Few falsifications test whether a false claim can pass: each was fixed beforehand, and a rejection shows that one is caught, not that none passes.

The fourth row of Table 7 is not a test of that program: there, test programs call the library beneath it directly, against the library's stated contract (`docs/limits.md`): a list of rules for C++ code that calls it, among them to name no private member of the library, to declare nothing at global scope before its headers, and to keep `operator new`. They run on small inputs of their own, which are not configurations. At the

128-bit target, on these small inputs, they had false claims accepted by at least two ways: in one, at all 9 tries that were carried through; in another, on synthetic data, at 8 of 48 tries against a control of 0 of 48. There may be others.

That the verifier program cannot reach the library's open ways rests on readings of that program's source, not on a test; the source read differs in places from the released one (Appendix H.1).

The altered tables that were accepted decode, by separate programs' counts of the changed columns (no decoder was run; Appendix H.1), to the unaltered weights, which the design allows: an accepted proof is meant to show that the committed table decodes to the weights, not that its bytes are their honest encoding. No malicious program was built from the released sources, and none searches for a false proof (Appendix H.6).

7.7 The integer model against floating point

On the public weights the integer model's greedy token differs from that of the reference, public copies of the 16-bit weights (two, the same bit for bit) evaluated in float32, at 6.81 % of token positions on 216 texts and at 9.41 % on the 24 natural passages among them (Python code†, no proofs; the same two shares, to the digits shown, in each of three runs). The count behind the first share was not the same in each run, and the cause is not known. All texts are built on WikiText-2 [MXBS17], on which the scales were fitted, and no task benchmark was run (`docs/limits.md`) (Limitation G.9, Quality unmeasured).

7.8 Comparison

Per GPU, proving alone takes 71.4 times (65.2 to 76.4 over all pairs of runs), 93.0 times (92.2 to 94.1 over all pairs of runs) and 20.1 times (19.7 to 20.5 over all pairs of runs) the GPU time of stock vLLM [KLZ23] for prefill of 24 windows at once, prefill at the highest load tried, and decoding at 512 streams ($n = 3$; the engine's $n = 30$, 8, 5). The engine served the FP8 checkpoint on eight H100s, on other machines; the multiples are calculated from the first rate of Table 3, omit committing and the serving that exact match requires (Section 7.4), and could be higher, since the engine's rates are lower bounds.

Table 8 sets our figures beside those of four earlier systems. By a simulation of 17 such machines [GHL26, Table 6], DeepProve's rate for GPT-2 rises to 1,855 tokens per minute.

Llama 4 Maverick is a mixture of experts: every second layer has 128 routed experts, of which a token uses one, together with a shared expert. Of the 400.7 billion committed weights, one token's forward pass uses 16.15 billion (computed from the parameter file; the whole output head and the one embedding row that a token reads are counted). Each request's proof also does work over all 400.7 billion committed weights, once per request and not once per token. The rate is therefore not comparable, parameter for parameter, with the rates for dense models.

8 Limitations and open problems

8.1 Limitations

The release lists the gaps that we know of, and no attack on the protocol is known to us at any gap while its conditions are kept, the opening budget among them (Limitation 8.4, Opening budget). Almost no attack was sought: that none is described is not a finding of safety. Table 16 names the release's pages.

Theorems and proofs concern the protocol, or Lean's IOPs and V_Z; measurements and tests concern the implementation. Nothing is proved about the released programs, and no security level is claimed for them. Between the theorem and the command `lean-verify` the gap is smaller than between the theorem and the C++ programs (Limitation 8.1 (Lean verifier only); Limitation 8.2 (Protocol match unproved)): `lean-verify` evaluates the proved verifier's executable form (Remark 5.9, What a client gets).

Limitation 8.1 (Lean verifier only). Theorem 5.1 concerns V_Z, not the C++ verifier, which clients run by default. Nothing is proved about that program, and no soundness is claimed for the programs as they run: an acceptance has the theorem's meaning only if V_Z gives it, and neither 124 bits per query nor 2^{-128} per round is stated of the C++ programs. No theorem states that the two decide alike on any file, or bounds a program's probability of accepting a false claim.

Table 8: Proofs of language-model inference: each system's results as its authors report them, without ratios, because the machines, models and proved statements differ.

System, model, input	Hardware	Proving	Proof	Verifying	Statement
This work (Table 3): integer model of the size of Llama 4 Maverick (400.7B weights, mixture of experts; 16.15B used per token); 1,024 tokens, batches of 24	one H100 (80 GB), with 24 CPU cores and up to 278.3 GB of host memory (Section 7.1)	111.23 ± 0.92 tokens/second (n = 3), counted in prefill tokens; 9.206 ± 0.077 seconds per request (n = 3), as throughput; released 220.7 to 225.9 s after the batch's start; committing apart, about 39 minutes	18.74 to 18.81 MB	21.14 ± 0.03 seconds called with its option for one thread, on the prover's machine (n = 3), by the C++ verifier	non-interactive; the greedy token that follows the prompt
VerInf [VerInf], as its README reports: its integer model of Llama 4 Maverick; 1,000 tokens	one DGX Spark	14.3 h; its README leaves open whether that includes committing	93.6 GB at 40 opened columns, below the 80 that its README calls deployment grade	17.7 h on 20 cores	interactive; a bound on the "unexplained information" of a continuation
zkLLM [SLZ24]: OPT-13B; LLaMA-2-13B; 2,048 tokens	one A100	713 s; 803 s; committing apart, about 1,270 s; 986 s	160 to 188 kB	3.71 s; 3.95 s	interactive; fixed point, scaling factor 2^{16}
DeepProve [GHL26, Table 2]: GPT-2 (124M); Gemma 3 (270M); 512 tokens; BaseFold commitment	one 16-core AMD Ryzen 9, 128 GB	2.94 min; 5.9 min (174.32; 86.76 tokens per minute), without the inference and without committing (a setup of under 0.91; under 2.64 min)	24.33; 43.6 MiB	1.65 s; 2.13 s; its abstract gives 1 to 3.7 s	all generated tokens; 12-bit quantization
Attestable [Attestable], by the passages of its blog post that we have read, as first results: Gemma 4 31B	one H100	53 tokens per second on one sequence of 16K tokens; 77 at batch four, across four of 4K	4.35 to 7.92 MiB, over all its successful benchmark configurations	157 to 648 ms on its reference CPU verifier	floating point, but linear operations in 8-bit integers; 100 bits of security

Limitation 8.2 (Protocol match unproved). That Lean's IOPs (Remark B.7, Which compiler) are the protocol of the C++ programs is not proved, nor that they are the technical report's [TP]: no theorem relates them, and the only evidence is the runs. In the one comparison of hash calls, on the stored proof at Z_0, a logging build of the C++ verifier† made 881,280 of the executable form's 1,166,983 calls, with equal answers; the rest are explained only in part, a gap that the release rates as heavy. On that proof both decide alike; on three altered copies of an honest proof (one bit turned in a digest the form never reads) the executable form accepted the parsed file where a C++ verifier and a parser, both of other programs, rejected the raw proof; the claim was true (`lean/GAPS.md`, gap 66). These calls were made over Lean files that differ from the released ones. Over the released Lean files the decision on the stored proof has been repeated (Appendix H.3); the comparison of hash calls and the calls on the altered copies have no run over the released Lean files. V_Z has sixteen tests that the C++ verifier lacks (`lean/GAPS.md`, gap 25).

Limitation 8.3 (Zero knowledge of definitions in Lean). The provers for which zero knowledge is machine-checked (Section 6.5) are the Lean provers: definitions in Lean, exactly as written, default values included. On the C++ programs nothing is stated. It is stated at the default configuration's size, after compilation at the exact-match policy, with fresh random masks, for one commitment and one request per proof, not for batches; before compilation for an honest verifier; after compilation in the random-oracle model, on one copy of the committed table, up to 688,127 replies and 2^{64} hash queries. All of it holds where Z is admissible with $\mathsf{Chk}(Z) = 1$, after compilation in use (Definition 6.11). Not hidden is one bit per request, before and after compilation: whether the request is served, which depends on the weights; in the corollary, one label per request (served, failed or other tokens). A bit says whether the claimed output is the model's result on that request, so over many requests the bits tell what the model answers. Nothing is said of what the files of Z themselves tell of the weights, or of timing. The simulators' running time is not modelled (`docs/limits.md`). Nor is derived randomness covered (Remark G.15, Pseudorandom masks). At a refused request the Lean prover returns the empty proof. The programs' service answers "failed" for every request that it does not prove, whatever the cause; a client can partly tell the causes apart (`docs/limits-all-rows.tsv`, row HW-532). That answer is neither the bit nor the corollary's label "failed", and no theorem ties them. That the released programs hide the weights is the design's aim, not an established property (Section 1.1); our judgement of it: Section 7.5.

Limitation 8.4 (Opening budget). A committed table can open only t columns before its random coefficients are used up, with no margin (Proposition 6.5). Beyond that, its proofs give whoever holds them exact linear equations, permanently and unnoticed: $t + k$ columns, $1 \le k \le d\,2^{n_0}$, give k independent equations modulo p in each row's weight digits and pads and, for $k > 3s$, at least $k - 3s$ in its digits alone (Proposition C.16). By a reading of other sources, a row that the prover keeps from one segment of a request's proof to the next is encoded anew for each segment, and each encoding has 219 random coefficients, for the 219 positions that a proof opens in it, without slack: a number calculated from the default configuration and taken from no run; that it is enough is not proved, and it is enough only if each new encoding's random coefficients are fresh, which by the code's own list of conditions no function there tests. A commitment serves boundedly many batches. The released programs count batches on each copy, at most 688,127 at the default configuration (calculated), and no program counts proofs (Limitation 8.2, Protocol match unproved). Behind the programs' counter no Lean theorem stands: nothing in Lean keeps count of the columns that a table has opened. The 688,127 of the compiled zero-knowledge result (Section 6.5) counts another thing: replies with one request to a proof, on one copy of the committed table. So a provider within the programs' count is not thereby within that result. No program detects a count copied, restored or rolled back (`docs/limits.md`).

Limitation 8.5 (Provider-chosen configuration). The provider, not the verifier, chooses the configuration: all three files for the programs, and for Theorem 5.1 whatever may vary in an admissible configuration. Theorem 5.1 needs it fixed in advance, before the oracle is sampled and independently of the adversary (Open problem 8.16); one tuned to the concrete hash function is outside Theorem 5.1. The C++ verifier accepts forms of configuration that no statement covers, some of the release's examples among them.

Limitation 8.6 (Scales can disable blocks). The scale file of an admissible configuration can disable any chosen subset of the model's 96 attention and feed-forward blocks, up to all of them, whatever the weights, so the client must read it. This was worked out by hand and is not a theorem.

Limitation 8.7 (Unattributed commitment). As in every commit-and-prove argument, a commitment does not say whose weights, or what model's, were committed: the provider chooses which weights, if any, it fixes, and a client has only the word of whoever published it. A client needs the commitment before it sends a prompt.

Limitation 8.8 (Clipped gap is weak). Under a clipped-gap policy an acceptance states only "within limits", which is much less than exact match (unless $\theta = 2^8 < C$, where an output meets the policy if and only if it meets exact match). Otherwise, meeting the policy bounds the sum of the counted tokens' clipped costs and the number of them that cost more than the clip, not each token (Remark 3.10, What a policy bounds), and, by design, does not show whether any given counted token is the greedy token. With $C = \theta$ the bound on the sum says nothing; if *cap* is also at least the number of counted tokens, as at $(256, 256, 1024)$, the count says nothing either. V_Z does not reject such a configuration, and there only the token after a full window is certified (`lean/THEOREM.md`).

Limitation 8.9 (Completeness on served requests only). Lean proves soundness of V_Z; of its completeness it proves, besides the reduction of Remark 6.49, only that, on served requests, it rejects the Lean prover's proof with probability at most 2^{-40}, in the random-oracle model, where the files of Z are in use (Z admissible, $\mathsf{Chk}(Z) = 1$, exact-match policy), for one request to a proof and one commitment (Section 6.5). Where the model has no result, or another result than the claimed output, nothing is stated. No honest proof exists for a window on which the model is not defined. The Lean prover is a definition: that V_Z accepts the programs' honest proofs is tested, on proofs made on other programs and on 41 of the timed runs' own on synthetic weights, at the two stored configurations Z_0 and Z_1 alone (Remark B.6, Acceptance), not proved. These calls were made over Lean files that differ from the released ones, but for the calls of the reference run and of the second set of 41, which were made over the released Lean files (Appendix H.3). Nor does Theorem 5.11 show that V_Z accepts any non-greedy output. A gap in completeness could only cost a visible rejection: it cannot make V_Z accept a false claim beyond the bound of Theorem 5.1, under its hypotheses, provided the functions that form the claim are right (Remark B.10, Claim from files).

Limitation 8.10 (Released commitment). The released commitment has 32 bytes and hashes the three configuration files, the first table's Merkle root, its out-of-domain answers and the out-of-domain seed. That it ties the provider to one set of weights is not proved for the released programs: what is proved of a commitment is proved of the committer compiled from the Lean provers (Section 6.5) and of an explicit committer that models part of Commit_Z (Open problem 8.11). Of a commitment a proof can certify at most the matrix to which the first table decodes, not that the table's bytes are its honest encoding, and the commitment does not fix the offline proof in the key file: one set of weights can have several commitments, one commitment several key files, and a proof is tied to its copy.

Appendix G states the further limitations: the implementation is unverified; only greedy outputs in a fixed window are covered, and prefixes count; the hash function is modelled as a random oracle, and adversaries are classical; there is no independent review, there are few runs, and the model's quality is unmeasured; outcomes are visible by design; and eleven limitations concern how the released programs are operated. Among the eleven: a further commitment to the same weights, which renewal or a second GPU requires, is inside the judgement of Section 7.5 only on thin ground and on five conditions, and no test has searched two commitments together for leaks (Limitation G.12, Further commitments).

8.2 Open problems

Open problem 8.11 (Extraction and binding for the released committer). [open] Prove correctness of extraction and extractability of the commitment (Definition 2.9 and Definition 2.10) for Commit_Z of the released programs. For the committer compiled from the Lean provers both are proved (Section 6.5). The hash function is drawn at random: these are statements in the random-oracle model. They hold at public files in use; the extractor is a function, not shown computable. The committer is a Lean term that no theorem ties to a program. For Commit_Z of the released programs nothing is proved: the explicit committer (Definition 5.16) models only part of it, and was compared, in values alone, with the committing program of other programs† (Table 4).

Each property that names the honest parties requires its own proof (Remark 4.16, Honest parties).

Open problem 8.12 (Completeness beyond served requests). [open] State and prove completeness of the compiled term, V_Z with the Lean committer and prover (Section 6.5; Remark 4.16 (Honest parties)), at well-formed requests on which the model has no result, and prove it for batches.

Open problem 8.13 (The twenty conditions as a theorem). [open] State, as a theorem of its own, that the three Lean provers (Section 6.5) meet the twenty conditions of Theorem C.25.

Zero knowledge of the Lean provers is machine-checked (Section 6.5). For the results on the whole-key argument the step to Π_Z is not proved; the compiled statement of Section 6.5, which is on Π_Z^{lp} (Definition 6.15), needs no such step for the commitment, which is the 256-bit one, or for the verifier; for the honest parties of Π_Z the step is not proved (Open problem 8.14). Batches are not covered.

Open problem 8.14 (Zero knowledge of the whole protocol). [open] Prove index privacy of Π_Z with its honest parties, for batches, on the instances that can be served.

Open problem 8.15 (Many proofs without the factor N). [open] Find q_H such that every run of V_Z makes at most q_H queries to H. Bound the error for N proofs sent one after another, each after the decision on the one before.

Given such a q_H, which no theorem provides, the bound $\varepsilon(q + Nq_H)$, without the factor N, is machine-checked for lists sent at once (Remark B.13, Lists sent at once).

Open problem 8.16 (A configuration chosen after the oracle). [open] Prove index-binding knowledge soundness (Definition 2.7) of Π_Z against a committer that chooses the admissible configuration Z after querying H.

9 Related work

Proofs of language-model inference. zkLLM [SLZ24] is interactive, uses a Hyrax commitment [WTSTW18] and hides the parameters from a semi-honest verifier. Our integer softmax and attention follow zkLLM, zkGPT [QSL25] and ZKML [CWS24]. zkAgent [WLL26] checks the greedy token by range-checking the logit differences over the whole vocabulary, as zkCNN [LXZ21, Section 4.3] does for max pooling; at exact match our protocol's last segment does the same, by looking up the digits of the differences.

DeepProve [GHL26] proves all generated tokens of an output, for GPT-2 and Gemma 3 models of 124M and 270M parameters. ZKTorch [CTK25] compiles a model into basic blocks and accumulates their proofs. ZK-DeepSeek [Wan25] turns DeepSeek-V3, with 671 billion parameters, into an integer model, composes proofs recursively into one of constant size, without a trusted setup, and reports times for several components. By its abstract, which is all that we have read of it, zkPyTorch [XLF25] compiles PyTorch models into circuits. By the passages of its blog post that we have read, Attestable [Attestable] reports early results for a hash-only zero-knowledge system over committed weights.

Proofs for the same model. VerInf [VerInf], which precedes this work, reports in its README an interactive, hash-based zero-knowledge proof, on one DGX Spark and 1,000 tokens, for its own integer model of Llama 4 Maverick, the model whose shape we use. It proves a different statement from ours, a bound on the "unexplained information" of a continuation, and it hides both the tokens and the weights. Our choice of public 4-bit weight files [Uns25] comes from VerInf, and our rule for converting the stored weights to integers grew out of VerInf's.

One committed model, many proofs. In the literature's terms the task of Π_Z is close to that of a function-hiding functional commitment [BNO21]: the committed function is the model, and Corollary B.14 has the form of evaluation binding. We state the likeness only; their definition has not been compared with ours clause by clause. [GGQZ25] builds such a commitment over KZG for convolutional networks, and zkCNN [LXZ21, Section 4.1] already has the prover commit to a network's parameters before proving predictions. By the passages that we have read, Artemis [LVAKH24] is a commit-and-prove SNARK for any homomorphic polynomial commitment, and reports cutting the overhead of tying proofs of inference to the model's commitment.

Attacks on committed models. Nothing in FlashProve prevents the Hollow-LLM attack [GLL26], in which weights of the declared size yield valid proofs but compute no more than a much smaller model (Limitation 8.7, Unattributed commitment). [CPD24] recovers a model's last layer, up to a linear map, from many vectors of its logits; whether vectors cut off at the clip suffice is not known, and as far as we know this has not been tried on these programs (Limitation G.21, Clipped gap reveals logits).

Alternatives to proofs. A trusted execution environment has a reported overhead below 7% on Hopper GPUs for most typical queries [ZYDAZ24], but the client must trust the hardware and its vendor. TOPLOC [TOPLOC] hashes intermediate activations into 258 bytes per 32 new tokens, and its check is

reported to be much faster than the inference. Token-DiFR [DiFR] scores the tokens themselves, at no added cost to the provider. In both, the verifier runs the model again, so it must hold the weights. Our clipped-gap policy builds on Token-DiFR's logit margin at temperature 0, but our verifier needs no weights (Remark 3.10, What a policy bounds).

Reported numbers. Table 8 sets our figures beside those of four earlier systems, VerInf (as its README reports them), zkLLM, DeepProve and Attestable. Appendix E.1 and Appendix E.2 give the sources of our masks, segments, batches and policies.

Who wrote and checked this

Under human direction, AI systems wrote this paper. The software and the Lean code are also almost entirely their work. The exceptions include code that they adapted from public sources, among them ArkLib and Mathlib (Appendix E.4) and the BLAKE2 reference code [BLAKE2]; the release's `THIRD_PARTY.md`, `NOTICE` and `lean/CREDITS.md` name the sources and what was taken. AI systems also wrote nearly all of the technical report [TP], except where it marks words as taken from a source.

For a machine-checked statement, Lean's kernel accepts a proof of each Lean statement that Appendix F names for it, with two provisos: that kernel is the pinned version's, of which Remark B.17 tells what is publicly known, and the re-checker stopped, for the depth of recursion, on constants in 23 modules of a separate library (counted over Lean files that differ from the released ones) which the build's kernel accepts at a thread stack larger than the default (Remark 5.7, How the proposition is checked); Remark 6.48 names the statements of Section 6.5 whose modules import that library. The kernel does not check what a statement means. That a Lean statement says what this paper's statement says is a reading; nothing machine-checks it. Six definitions of properties in the library (`Sound`, `SoundAndBinding`, `SoundOpened`, `Complete`, `ZeroKnowledge`, `CompleteSoundZeroKnowledge`; `lean/VerifiableInference/Statements.lean`) were shaped with a person. The library's six top theorems, which the release's page `lean/STATEMENTS.html` shows first, are `sound`, `soundAndBinding`, `completeOnCanServe`, `zeroKnowledgeOnCanServe`, `refusedHandsOutNoProof` and `repliesFit`; Appendix F names them under Remark 6.16, Theorem 6.31, Theorem 6.29, Theorem 6.35, and Proposition 6.33. The first two are stated through the first two of these definitions. The third and the fourth are stated through this library's `CompleteOn` and `ZeroKnowledgeOn`, the texts of `Complete` and `ZeroKnowledge`, each with one condition more, that the request can be served (Definition 3.11); with the condition that the output is the model's, these are the served requests. The last two are this library's own, and so are the system, the bounds and the conditions on the files in all six. The library also has the *final statement*, one Lean structure with six fields (`lean/AUDIT.md`), all proved by one Lean theorem; Appendix F names it under Theorem 6.29, Proposition 6.33, and Corollary 6.38. `completeOnCanServe` and `zeroKnowledgeOnCanServe` are no fields of it.

AI systems made every test, run, comparison, calculation, reading and review that this paper reports: where it says "by a reading", "by hand" or "as reported", the reader, calculator or reporter is an AI system. Such checking as there was, of the match between the English statements and the Lean ones and of the written proofs, tests and judgements, was made by AI readers of this project; no written proof has been read whole by one reader, and, as far as we know, no person has checked any of these (Limitation G.7, No independent review).

Acknowledgments: We thank Alessandro Chiesa, Giacomo Fenzi, Guy Weissenberg, Miranda Christ, Rahul Dalal, and Ben Mayer for their input and collaboration on FlashProve. We also thank James Petrie for VerInf [VerInf].

References

[ABF26] G. Arnon, D. Boneh, G. Fenzi. Open Problems in List Decoding and Correlated Agreement. Cryptology ePrint Archive 2026/680 (print dated April 8, 2026).

[ACFY24a] G. Arnon, A. Chiesa, G. Fenzi, E. Yogev. STIR: Reed-Solomon proximity testing with fewer queries. CRYPTO 2024; Cryptology ePrint Archive 2024/390 (sections as in the version of January 27, 2025).

[ACFY24b] G. Arnon, A. Chiesa, G. Fenzi, E. Yogev. WHIR: Reed-Solomon proximity testing with super-fast verification. EUROCRYPT 2025; Cryptology ePrint Archive 2024/1586 (sections as in the version of November 21, 2024).

[ACY22] G. Arnon, A. Chiesa, E. Yogev. A PCP theorem for interactive proofs and applications. EUROCRYPT 2022; Cryptology ePrint Archive 2021/915.

[AER24] G. Angeris, A. Evans, G. Roh. A note on Ligero and logarithmic randomness. Cryptology ePrint Archive 2024/1399.

[AHIV17] S. Ames, C. Hazay, Y. Ishai, M. Venkitasubramaniam. Ligero: lightweight sublinear arguments without a trusted setup. ACM CCS 2017; Cryptology ePrint Archive 2022/1608.

[ArkLib] ArkLib Contributors. ArkLib. A public library for the proof assistant Lean (Apache-2.0), github.com/Verified-zkEVM/ArkLib; commit 2817d8c7 (Appendix E.4).

[Attestable] Attestable. Proving LLMs at scale. Blog post, attestable.com/blog/proving-llms-scale (2026).

[BCFGRS17] E. Ben-Sasson, A. Chiesa, M. A. Forbes, A. Gabizon, M. Riabzev, N. Spooner. Zero knowledge protocols from succinct constraint detection. TCC 2017, pp. 172–206.

[BCFW25] B. Bünz, A. Chiesa, G. Fenzi, W. Wang. Linear-time accumulation schemes. TCC 2025; Cryptology ePrint Archive 2025/753 (version of May 28, 2025).

[BCGJM18] J. Bootle, A. Cerulli, J. Groth, S. Jakobsen, M. Maller. Arya: nearly linear-time zero-knowledge proofs for correct program execution. ASIACRYPT 2018; full version: Cryptology ePrint Archive 2018/380.

[BCGM25] S. Bordage, A. Chiesa, Z. Guan, I. Manzur. All polynomial generators preserve distance with mutual correlated agreement. CCC 2026; Cryptology ePrint Archive 2025/2051, version of February 6, 2026.

[BCHKS25] E. Ben-Sasson, D. Carmon, U. Haböck, S. Kopparty, S. Saraf. On proximity gaps for Reed-Solomon codes. Cryptology ePrint Archive 2025/2055, version of November 6, 2025; the full version of [BCHKS26].

[BCHKS26] E. Ben-Sasson, D. Carmon, U. Haböck, S. Kopparty, S. Saraf. On proximity gaps of Reed-Solomon codes. STOC 2026, pp. 1157–1167; doi 10.1145/3798129.3800827.

[BCIKS20] E. Ben-Sasson, D. Carmon, Y. Ishai, S. Kopparty, S. Saraf. Proximity gaps for Reed–Solomon codes. FOCS 2020; full version: Cryptology ePrint Archive 2020/654, version of July 3, 2021.

[BCRSVW18] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, N. P. Ward. Aurora: Transparent Succinct Arguments for R1CS. EUROCRYPT 2019; Cryptology ePrint Archive 2018/828 (print dated May 8, 2019).

[BCS16] E. Ben-Sasson, A. Chiesa, N. Spooner. Interactive oracle proofs. TCC 2016-B; Cryptology ePrint Archive 2016/116.

[BFRW25] B. Bünz, G. Fenzi, R. D. Rothblum, W. Wang. TensorSwitch: nearly optimal polynomial commitments from tensor codes. CRYPTO 2026; Cryptology ePrint Archive 2025/2065, version of November 15, 2025.

[BGKS20] E. Ben-Sasson, L. Goldberg, S. Kopparty, S. Saraf. DEEP-FRI: sampling outside the box improves soundness. ITCS 2020; Cryptology ePrint Archive 2019/336.

[BGKTTZ23] A. Block, A. Garreta, J. Katz, J. Thaler, P. Tiwari, M. Zając. Fiat-Shamir Security of FRI and Related SNARKs. ASIACRYPT 2023; Cryptology ePrint Archive 2023/1071.

[BGTZ23] A. Block, A. Garreta, P. Tiwari, M. Zając. On soundness notions for interactive oracle proofs. Cryptology ePrint Archive 2023/1256.

[BLAKE2] BLAKE2 reference source code package (reference C implementations): blake2b-ref.c, blake2.h, blake2-impl.h. Copyright 2012, Samuel Neves. github.com/BLAKE2/BLAKE2. Used under CC0 1.0.

[BNO21] D. Boneh, W. Nguyen, A. Ozdemir. Efficient functional commitments: how to commit to a private function. Cryptology ePrint Archive 2021/1342.

[CCHLRR18] R. Canetti, Y. Chen, J. Holmgren, A. Lombardi, G. Rothblum, R. Rothblum. Fiat-Shamir From Simpler Assumptions. Cryptology ePrint Archive 2018/1004 (print dated October 23, 2018).

[CF24] A. Chiesa, G. Fenzi. zkSNARKs in the ROM with Unconditional UC-Security. TCC 2024; Cryptology ePrint Archive 2024/724 (print dated September 4, 2024).

[CFQ19] M. Campanelli, D. Fiore, A. Querol. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. ACM CCS 2019; Cryptology ePrint Archive 2019/142.

[CFS17] A. Chiesa, M. A. Forbes, N. Spooner. A zero knowledge sumcheck and its applications. Cryptology ePrint Archive 2017/305.

[CFW26] A. Chiesa, G. Fenzi, G. Weissenberg. Zero-knowledge IOPPs for constrained interleaved codes. Cryptology ePrint Archive 2026/391. Places are those of the version of February 25, 2026.

[CFWpc] A. Chiesa, G. Fenzi, G. Weissenberg. Personal communication, 2026.

[CGSY23] A. Chiesa, Z. Guan, S. Samocha, E. Yogev. Security Bounds for Proof-Carrying Data from Straightline Extractors. TCC 2024; Cryptology ePrint Archive 2023/1646 (print dated September 11, 2024).

[CHMMVW19] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, N. Ward. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. EUROCRYPT 2020; Cryptology ePrint Archive 2019/1047 (print dated January 1, 2020).

[CK11] M. Chase, S. Kamara. Structured Encryption and Controlled Disclosure. ASIACRYPT 2010; full version: Cryptology ePrint Archive 2011/010.

[CMS19] A. Chiesa, P. Manohar, N. Spooner. Succinct arguments in the quantum random oracle model. TCC 2019; Cryptology ePrint Archive 2019/834 (definitions as in the version of January 14, 2020).

[CO25] A. Chiesa, M. Orrù. A Fiat–Shamir Transformation From Duplex Sponges. Cryptology ePrint Archive 2025/536 (version of March 27, 2026).

[COS20] A. Chiesa, D. Ojha, N. Spooner. Fractal: post-quantum and transparent recursive proofs from holography. EUROCRYPT 2020; Cryptology ePrint Archive 2019/1076.

[CPD24] N. Carlini, D. Paleka, K. Dvijotham, T. Steinke, J. Hayase, A. F. Cooper, K. Lee, M. Jagielski, M. Nasr, A. Conmy, I. Yona, E. Wallace, D. Rolnick, F. Tramèr. Stealing part of a production language model. ICML 2024 (PMLR 235); arXiv:2403.06634.

[CSZS25] W. Cai, T. Shi, X. Zhao, D. Song. Are you getting what you pay for? Auditing model substitution in LLM APIs. arXiv:2504.04715 (2025).

[CTK25] B.-J. Chen, L. Tang, D. Kang. ZKTorch: compiling ML inference to zero-knowledge proofs via parallel proof accumulation. arXiv:2507.07031 (2025).

[CWS24] B.-J. Chen, S. Waiwitlikhit, I. Stoica, D. Kang. ZKML: an optimizing system for ML inference in zero-knowledge proofs. EuroSys 2024.

[CY24] A. Chiesa, E. Yogev. Building cryptographic proofs from hash functions. 2024. https://snargsbook.org. Section numbers are those of Version 1.2 and may differ from those of the posted version.

[DG24] B. E. Diamond, A. Gruen. Proximity gaps in interleaved codes. IACR Communications in Cryptology, 2025; Cryptology ePrint Archive 2024/1351.

[DiFR] A. Karvonen, D. Reuter, R. Rinberg, L. Marks, A. Garriga-Alonso, K. Warr. DiFR: inference verification despite nondeterminism. arXiv:2511.20621 (2025).

[DP23] B. E. Diamond, J. Posen. Proximity testing with logarithmic randomness. IACR Communications in Cryptology, 2024; Cryptology ePrint Archive 2023/630.

[Fen26] G. Fenzi. How to prove more false statements: Fiat-Shamir limitations on (generated) R1CS. Cryptology ePrint Archive 2026/1838.

[GGQZ25] Y. Guo, Z. Guo, W. Qu, J. Zhang. Architecture-private zero-knowledge proof of neural networks. Cryptology ePrint Archive 2025/2211.

[GHL22] C. Gentry, S. Halevi, V. Lyubashevsky. Practical non-interactive publicly verifiable secret sharing with thousands of parties. EUROCRYPT 2022; full version: Cryptology ePrint Archive 2021/1397.

[GHL26] N. Gailly, I. Hishon-Rezaizadeh, T. Liu, N. Mainardi, D. Papadopoulos, C. Papamanthou, C. Pappas, S. Srinivasan, Z. Youell, Y. Zhang. DeepProve: verifiable end-to-end large language model inference. ACM CCS 2026; Cryptology ePrint Archive 2026/1112, version of May 30, 2026.

[GKR08] S. Goldwasser, Y. T. Kalai, G. N. Rothblum. Delegating computation: interactive proofs for muggles. STOC 2008.

[GLL26] C. Gong, B. Liu, M. Li. Hollow-LLM attack: computationally trivial weights in zero-knowledge verification of LLM inference. IEEE Symposium on Security and Privacy 2026; arXiv:2607.28884.

[GLSTW23] A. Golovnev, J. Lee, S. Setty, J. Thaler, R. S. Wahby. Brakedown: linear-time and field-agnostic SNARKs for R1CS. CRYPTO 2023; full version: Cryptology ePrint Archive 2021/1043.

[Gru24] A. Gruen. Some improvements for the PIOP for ZeroCheck. Cryptology ePrint Archive 2024/108.

[GW20] A. Gabizon, Z. J. Williamson. plookup: a simplified polynomial protocol for lookup tables. Cryptology ePrint Archive 2020/315.

[Hab25] U. Haböck. A note on mutual correlated agreement for Reed-Solomon codes. Cryptology ePrint Archive 2025/2110.

[Hol19] J. Holmgren. On Round-By-Round Soundness and State Restoration Attacks. Cryptology ePrint Archive 2019/1261.

[Jo26] S. Jo. Interleaving stability for mutual correlated agreement and curve decodability. Cryptology ePrint Archive 2026/891.

[KLZ23] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, I. Stoica. Efficient memory management for large language model serving with PagedAttention. SOSP 2023; arXiv:2309.06180.

[KRS25] D. Khovratovich, R. D. Rothblum, L. Soukhanov. How to prove false statements: practical attacks on Fiat-Shamir. CRYPTO 2025; Cryptology ePrint Archive 2025/118.

[LFKN92] C. Lund, L. Fortnow, H. Karloff, N. Nisan. Algebraic methods for interactive proof systems. Journal of the ACM 39(4), 1992.

[LNS21] V. Lyubashevsky, N. K. Nguyen, G. Seiler. Shorter lattice-based zero-knowledge proofs via one-time commitments. PKC 2021; full version: Cryptology ePrint Archive 2020/1448.

[LVAKH24] H. Lycklama, A. Viand, N. Avramov, N. Küchler, A. Hithnawi. Artemis: efficient commit-and-prove SNARKs for zkML. arXiv:2409.12055 (2024).

[LXZ21] T. Liu, X. Xie, Y. Zhang. zkCNN: zero knowledge proofs for convolutional neural network predictions and accuracy. ACM CCS 2021; Cryptology ePrint Archive 2021/673.

[LY26] Y. Lim, S. Yong. Seal, then sample: sampled layerwise proofs for verifiable LLM inference from GPT-2 to 70B. arXiv:2609.27367 (2026).

[Mathlib] Mathlib. A public library for the proof assistant Lean (Apache-2.0), github.com/leanprover-community/mathlib4; its files name their own authors. The Lean library builds on one pinned revision of it (commit 5e932f97); 13 files with 18 declarations adapted from a later one (tag v4.34.0) come with this repository (lean/ThirdParty/MathlibBackport/).

[MXBS17] S. Merity, C. Xiong, J. Bradbury, R. Socher. Pointer sentinel mixture models. ICLR 2017; arXiv:1609.07843.

[QSL25] W. Qu, Y. Sun, X. Liu, T. Lu, Y. Guo, K. Chen, J. Zhang. zkGPT: an efficient non-interactive zero-knowledge proof framework for LLM inference. USENIX Security 2025; Cryptology ePrint Archive 2025/1184.

[RFC7693] M.-J. Saarinen (ed.), J.-P. Aumasson. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693, RFC Editor, November 2015.

[RFC8439] Y. Nir, A. Langley. ChaCha20 and Poly1305 for IETF protocols. RFC 8439, Internet Research Task Force (IRTF), June 2018.

[RRR16] O. Reingold, G. N. Rothblum, R. D. Rothblum. Constant-round interactive proofs for delegating computation. STOC 2016, pp. 49–62.

[SGO26] P. K. Sanjaya, C. Giannoula, V. Oktavian, M. Saeedi, G. Sines, G. Saileshwar, N. Vijaykumar. zkComposer: decomposing proof construction to scale zkML. arXiv:2607.08095 (2026).

[SLZ24] H. Sun, J. Li, H. Zhang. zkLLM: zero knowledge proofs for large language models. ACM CCS 2024; arXiv:2404.16109.

[Sta21] E. Ben-Sasson, D. Brownstein, D. Carmon, L. Goldberg, D. Levit, D. Medini, A. Shtaierman, E. Yogev. ethSTARK documentation, version 1.2 (July 2023). Cryptology ePrint Archive 2021/582.

[Tha13] J. Thaler. Time-optimal interactive proofs for circuit evaluation. CRYPTO 2013; full version: Cryptology ePrint Archive 2013/351; arXiv:1304.3812.

[TOPLOC] J. M. Ong, M. Di Ferrante, A. Pazdera, R. Garner, S. Jaghouar, M. Basra, M. Ryabinin, J. Hagemann. TOPLOC: a locality sensitive hashing scheme for trustless verifiable inference. ICML 2025 (PMLR 267); arXiv:2501.16007.

[transformers] The Hugging Face team. Hugging Face transformers, version 5.17.0 (Apache-2.0), github.com/huggingface/transformers.

[TP] Proving a Language Model's Replies under Committed Weights: Protocol, Proofs and Evaluation. Technical report, 2026. The technical report is not in this repository, and a reader of this release cannot open it (Appendix E.1).

[Uns25] Unsloth. Llama-4-Maverick-17B-128E-Instruct-GGUF (GGUF quantizations of Llama 4 Maverick; the file set UD-Q4_K_XL). huggingface.co/unsloth/Llama-4-Maverick-17B-128E-Instruct-GGUF.

[VCVio] VCV-io. A public library for the proof assistant Lean (Apache-2.0), github.com/Verified-zkEVM/VCV-io; its files name their own authors.

[VerInf] J. Petrie. VerInf. Source code and README, github.com/JamesPetrie/VerInf, commit 89903c96 (August 2026).

[Wan25] Y. Wang. Zero-knowledge proof based verifiable inference of models. arXiv:2511.19902 (2025).

[Wan26] Z. Wang. NanoZK: privacy-preserving verifiable inference for large language models via layerwise zero-knowledge proofs. ICICS 2026; arXiv:2603.18046.

[WLL26] L. Wang, H. Lou, C. Li, Y. Yu, Y. Hu. zkAgent: verifiable LLM agent execution via one-shot transcript proofs. Cryptology ePrint Archive 2026/199 (version of May 1, 2026; the version of February 2026 is titled zkAgent: verifiable agent execution via one-shot complete LLM inference proof). Sections and equations are numbered as in the version of February 2026.

[WTSTW18] R. S. Wahby, I. Tzialla, a. shelat, J. Thaler, M. Walfish. Doubly-efficient zkSNARKs without trusted setup. IEEE Symposium on Security and Privacy 2018; Cryptology ePrint Archive 2017/1132.

[XLF25] T. Xie, T. Lu, Z. Fang, S. Wang, Z. Zhang, Y. Jia, D. Song, J. Zhang. zkPyTorch: a hierarchical optimized compiler for zero-knowledge machine learning. Cryptology ePrint Archive 2025/535.

[XZZPS19] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, D. Song. Libra: succinct zero-knowledge proofs with optimal prover computation. CRYPTO 2019; Cryptology ePrint Archive 2019/317.

[YLFKM21] T. Yurek, L. Luo, J. Fairoze, A. Kate, A. Miller. hbACSS: how to robustly share many secrets. NDSS 2022; Cryptology ePrint Archive 2021/159.

[ZXHSZ22] J. Zhang, T. Xie, T. Hoang, E. Shi, Y. Zhang. Polynomial commitment with a one-to-many prover and applications. USENIX Security 2022, pp. 2965–2982.

[ZYDAZ24] J. Zhu, H. Yin, P. Deng, A. Almeida, S. Zhou. Confidential computing on NVIDIA Hopper GPUs: a performance benchmark study. arXiv:2409.03992 (2024).

A Proof of the soundness theorem

Theorem 5.1 rests on the check of its Lean proof by Lean's kernel; the printed proof is given to explain it. Each lemma restates a Lean lemma on which that proof rests, with its hypotheses and constants; part 1 of Lemma A.17 is true, but no Lean statement has its form. A proof marked *outline* names Lean's lemmas and does not argue them. For each machine-checked statement, Appendix F names the Lean declarations that it renders.

A.1 Overview

Table 9: The chain of reductions, from the theorem downwards, with what each adds to the error or to q.

Step	By	Cost
Greedy outputs to the policy	Proposition 3.5	none
Weights and tokens to $(A, \xi) \in \mathsf{Req}_Z$	Lemma A.33	none
Parser and public readings	Lemma A.15 and Lemma A.35	$1 + 8$ queries
Commitment c to the key k	Lemma A.14	$(q+11)^2/2^{256}$
Digests from H to constants	Lemma A.13	none
H to separate oracles	Lemma A.12	none
A false encoded request accepted to Bad	Lemma A.11	factor 5; $q_V + 1$ queries
Bad to single rounds	Lemma A.8	$\varepsilon_{\mathrm{rd}}$ for each query; $2(Q+1)^2/2^{256}$
Rounds to codes and polynomials	Proposition 5.14	$\varepsilon_{\mathrm{rd}} \le 2^{-128}$

With $Q = 5\,(q + 9 + q_V + 1)$ and $q_V \le 2^{20}$, the steps give the error

$$Q \cdot 2^{-128} \;+\; \frac{2\,(Q+1)^2}{2^{256}} \;+\; \frac{(q+11)^2}{2^{256}}. \tag{1}$$

In $T = q + 1 + 2^{23}$ the 1 is the parser's query. The summand 2^{23} covers q_V, the eight queries of the public readings and the announcing query, and $Q \le 5T$. The term $16\,T/2^{128}$ of $\varepsilon(q)$ covers the first two terms where $\varepsilon(q) < 1$, and $(T+2)^2/2^{256}$ the third (Lemma A.36).

A.2 From the argument to three IOPs

Round-by-round soundness was introduced in [CCHLRR18, Definition 5.3]; [Hol19, Definition 2.5] restates it with a doomed set of partial transcripts. Related work on doomed sets: [BGKTTZ23, BGTZ23]. Round-by-round knowledge soundness is due to [CMS19, Definition 8.5], as round-by-round knowledge, in a form that differs from ours in detail; [CMS19, Definition 8.4] states round-by-round soundness for IOPs. Compare [CFW26, version of February 25, 2026, Definitions 3.6 and 3.7]; some of the Lean files that Definition A.1 and Definition A.2 follow are among those marked as written from that work (Table 13). That work has the notion from [BCFW25], extended to relaxed relations.

Definition A.1 (Round-by-round soundness). Let $\mathcal{L}$ be a set of pairs $(\mathsf{in}, \mathsf{ext})$ of an IOP instance and external oracle strings (Definition 2.3). An IOP is *round-by-round sound* for $\mathcal{L}$ with *round errors* $(\varepsilon_m)_m$ if there is a function State from such pairs with a partial transcript to $\{0, 1\}$ such that:

1. for $(\mathsf{in}, \mathsf{ext}) \notin \mathcal{L}$, the empty transcript has state 0;
2. no prover message changes the state from 0 to 1;
3. the challenge of round m does so with probability at most ε_m;
4. the verifier rejects every complete transcript of state 0.

In a batch IOP each IOP instance has the state of its local transcript, and in a shared round ε_m bounds the probability that the state of some instance of the batch changes from 0 to 1.

Definition A.2 (Early part, knowledge soundness). Let Q be a batch IOP and early a map on IOP instances, the *early part*. Q is round-by-round sound for $\mathcal{L}$ *with early part* early and round errors $(\varepsilon_m)_m$ if there is a function State that meets the conditions on prover messages and challenges (Definition A.1) and, for every $(\mathsf{in}, \mathsf{ext}) \notin \mathcal{L}$:

1. the empty transcript has state 0 at $(\mathsf{early}(\mathsf{in}), \mathsf{ext})$;

2. the verifier rejects in on complete transcripts of state 0 at $(\mathsf{early}(\mathsf{in}), \mathsf{ext})$.

Let $\mathcal{L}^{\mathsf{wit}}$ be a set of such pairs with a witness, wit a *witness map*, and Q' be Q with a verifier that also rejects each complete transcript trs with $(\mathsf{in}, \mathsf{ext}, \mathsf{wit}(\mathsf{in}, \mathsf{ext}, \mathsf{trs})) \in \mathcal{L}^{\mathsf{wit}}$. Q is *round-by-round knowledge sound* for $\mathcal{L}^{\mathsf{wit}}$ under wit if Q' is sound in this sense for $\mathcal{L} = \emptyset$.

Definition A.2 is used with the empty witness alone (Definition A.6). There, round-by-round knowledge soundness is round-by-round soundness with an early part.

Terms of Construction D.11. The *separate oracles* are Merkle oracles for leaves and for inner nodes, of trees over oracle strings and of the batch tree, and, for each of the three IOPs, a Fiat–Shamir oracle fs and an oracle g. An answer of fs gives a round's challenge and a *chaining value*, which the query of the next round holds. The first query of a compiled verifier holds a list cms of digests. The *early part* of an encoded request ξ is ξ with y^+ replaced by 0. $\mathsf{Seed}(\mathsf{rt})$ is the OOD seed that two queries to fs derive from a root. The oracles ann and *spare* occur in the analysis alone (Definition A.3). The function H' enters the proof as one fixed function (Assumption 4.11; Remark 4.12).

Definition A.3 (The argument over separate oracles). Fix Z and a value pv of the public readings $(\mathsf{dg}_Z, \mathsf{par})$ of Construction 4.15. The argument $\Pi^{\circ}_{Z,\mathsf{pv}}$ queries the separate oracles of Construction D.11, which are independent and uniform. Further oracles exist, which its verifier never queries; among them are ann, with a single possible answer, and a *spare* oracle on bit strings.

- A matrix A in range takes the place of the index, an encoded request ξ that of an instance; the relation is Req_Z.
- A commitment is a key $\mathsf{k} = (\mathsf{rt}_0, \sigma_0)$.
- A proof is $\pi^{\circ} = (j, \mathsf{rt}_j, \sigma_j, \pi_{\mathrm{key}}, \pi_{\mathrm{link}}, \pi_{\mathrm{req}})$.
- The verifier V° performs steps 6 to 9 of V_Z at the value pv.
- The extractor is $\mathsf{Ext}^{\circ}(\mathsf{k}, \mathsf{tr}) = \mathsf{Dec}(\mathsf{Wd}(\mathsf{tr}, \mathsf{rt}_0), \sigma_0)$, with Wd as in Definition A.4.

The number q_V bounds the queries of V° to the separate oracles, on every input.

Definition A.4 (Word under a root, rooted query, seed-first trace). Let tr be a trace over the separate oracles.

1. The *word under* a digest rt, $\mathsf{Wd}(\mathsf{tr}, \mathsf{rt})$, has at each position the symbol of the leaf query that tr determines under rt, else a fixed symbol.
2. rt is *announced* in tr if tr contains the query $\mathsf{ann}(\mathsf{rt})$.
3. A *start query* is the first query to fs of step 1 of Construction D.11. A query to fs is *rooted* in tr if it is a start query, or its chaining fields are chaining values of rooted queries of the preceding round; queries of Seed are not rooted. A rooted query determines a partial transcript: challenges from answers, plain parts from queries, oracle parts as the words under the queried roots.
4. tr is *seed-first* if each start query is preceded by the queries of Seed for the first two digests of its cms. Its *seed-first completion* $\overline{\mathsf{tr}}$ inserts these queries, at most four.

Definition A.5 (The event Bad). Fix state functions of the three IOPs (Definition A.1 and Definition A.2). A trace lies in Bad if an entry that is not contained in the prefix pre before it causes one of four events.

1. *Merkle event.* The answer of a Merkle oracle is a digest that occurs in pre.
2. *Chaining event.* The chaining value of an answer of fs, or an answer of g, occurs in pre.
3. *State event.* A rooted query to fs has an answer that changes the state of its partial transcript from 0 to 1; in a shared round, that of some encoded request under the queried root of the batch tree. External oracle strings are the words in pre under the first digests of cms; OOD seeds are read off pre, OOD answers off cms.
4. *Seed event.* The seed of the second query of $\mathsf{Seed}(\mathsf{rt})$ is a collision of $\mathsf{Wd}(\mathsf{pre}, \mathsf{rt})$.

By definition, Bad is closed under extension of traces.

Definition A.6 (Conditions on soundness). Let $\varepsilon_{\mathrm{rd}} \geq 0$. The sets are those of Construction 4.13.

(Sound-key). $\mathsf{IOP}^{\mathrm{key}}_Z$ is round-by-round sound for its set, with round errors at most $\varepsilon_{\mathrm{rd}}$ (Definition A.1).

(Sound-link proof). The same holds of $\mathsf{IOP}^{\mathrm{link}}_Z$.

(Sound-request). $\mathsf{IOP}^{\mathrm{req}}_Z$, as a batch IOP for at most R encoded requests, is round-by-round knowledge sound for its set, with the empty witness, with the early part of ξ, and with round errors at most $\varepsilon_{\mathrm{rd}}$ (Definition A.2).

In the first two, a shared round's error is counted once for each leaf of the tree of the call's requests.

Definition A.7 (Conditions of the compiler). The fifteen *conditions of the compiler*, at $\varepsilon_{\mathrm{rd}} \geq 0$ and a value pv of the public readings, are the three of Definition A.6 and twelve *side conditions*:

(No-decoding-at-a-collision). $\mathsf{Coll}(\mathbf{T}^*, \varsigma)$ implies $\mathsf{Dec}(\mathbf{T}^*, (\varsigma, \mathbf{a})) = \bot$.

(Local), for each IOP. The verifier reads oracle strings only at declared positions.

(Coherent), for each IOP. A plain part is the pair of its two components' plain parts; each oracle position lies in one component.

(Rare-collisions). For every word, the OOD seed read off uniform hash blocks is a collision with probability at most $\varepsilon_{\mathrm{rd}}$.

(Layout). Queries determine rounds and openings; uniform answers give uniform challenges.

(Answers-code), two conditions. The code of OOD answers is injective and of one length.

(Hashed-key). The implication of Lemma A.14, for $\Pi^{\circ}_{Z,\mathsf{pv}}$.

The conditions on soundness are the subject of Appendix A.3. (Layout) is stated informally: it summarizes the laws of four Lean structures, which this paper does not print. (Hashed-key) holds by Lemma A.14.

Lemma A.8 (Bad is rare). [machine-checked] *Assume the conditions of Definition A.7 other than (Hashed-key), and form* Bad *(Definition A.5) with the state functions that (Sound-key), (Sound-link proof) and (Sound-request) of Definition A.6 provide. Then the trace* tr *of every q-query algorithm over the separate oracles satisfies*

$$\Pr\big[\,\mathsf{tr} \in \mathsf{Bad}\,\big] \;\leq\; q\,\varepsilon_{\mathrm{rd}} + \frac{2\,(q+1)^2}{2^{\lambda}}.$$

Proof. We bound the probability that the answer to a new query causes an event, given pre, and take a union bound.

Step 1: Merkle and chaining events. The answer, or its chaining value, is uniform in $\{0,1\}^{\lambda}$ and independent of pre, in which each entry holds at most four values that count as occurring. The i-th query causes such an event with probability at most $4i/2^{\lambda}$; the sum over $i < q$ is at most $2q^2/2^{\lambda}$.

Step 2: state events. If a query to the oracle fs of one IOP is rooted, pre determines its partial transcript, the IOP instance and the external oracle strings; among several preceding queries with one chaining value the first is taken. By (Layout) the challenge is uniform, and by the condition on that IOP it changes a state from 0 to 1 with probability at most $\varepsilon_{\mathrm{rd}}$. In a shared round pre determines at most one encoded request for each leaf of the batch tree, and the round's error bounds the event for all of them jointly.

Step 3: seed events. A second query of Seed causes no state event. The word $\mathsf{Wd}(\mathsf{pre}, \mathsf{rt})$ is determined before its answer, and (Rare-collisions) applies. □

Lemma A.9 (Outside Bad). [machine-checked] *Assume the conditions of Definition A.7 other than (Hashed-key). Let* $\mathsf{tr} \notin \mathsf{Bad}$.

1. Stable words. *If* rt *is announced in a prefix* tr_0 *of* tr*, then* $\mathsf{Wd}(\mathsf{tr}_0, \mathsf{rt}) = \mathsf{Wd}(\mathsf{tr}, \mathsf{rt})$.
2. Seeds. *If* tr *contains the queries of* $\mathsf{Seed}(\mathsf{rt})$*, with value* ς*, then* $\mathsf{Coll}(\mathsf{Wd}(\mathsf{tr}, \mathsf{rt}), \varsigma)$ *does not hold.*
3. Accepting runs. *Suppose that* tr *is seed-first and has equal answers at equal queries, and that it contains an accepting run of the compiled verifier of one of the three IOPs and, for each root of an external oracle string, the queries of* Seed*, with the OOD seed of the IOP instance as value. Then the relation of that IOP (Construction 4.13) holds of the IOP instance and the words under these roots in* tr.

Proof. Part 1. Without a Merkle event, no query after tr_0 has as its answer a digest that occurs in tr_0; the tree under rt can neither change nor grow.

Part 2. Without a chaining event, the second query of $\mathsf{Seed}(\mathsf{rt})$ follows the first, in which rt occurs; from then on the word under rt is stable. A collision at ς would be a seed event.

Part 3. Without chaining events, the queries to fs of the accepting run occur in tr in the order of the rounds, and each is rooted. Because tr is seed-first, the OOD seeds are read off the prefix before the start query; by (Answers-code) the OOD answers are read off their injective code. The partial transcripts therefore belong to one IOP instance and to the stable words under the roots.

Suppose for contradiction that the relation fails. The state is 0 at the start and stays 0, since there is no state event, and the IOP's verifier rejects the complete transcript. By (Local), (Coherent) and the verified openings, the compiled verifier decides as the IOP's verifier does. For $\mathsf{IOP}^{\mathrm{req}}_{Z}$ the start query holds only the early part of ξ; this is why (Sound-request) is stated with that part. □

Lemma A.10 (The three relations in sequence). [machine-checked] *Assume the conditions of Definition A.7 other than (Hashed-key). Let* $\mathsf{k} = (\mathsf{rt}_0, \sigma_0)$*, and let* $\mathsf{tr} = \mathsf{tr}_C \parallel \mathsf{ann}(\mathsf{rt}_0) \parallel \mathsf{tr}'$ *be a trace whose answers are those of fixed oracle functions, where* tr' *contains an accepting run of* $V^\circ(\mathsf{k}, \xi, \pi^\circ)$*. If* $\overline{\mathsf{tr}} \notin \mathsf{Bad}$*, then*

$$\mathsf{Ext}^\circ(\mathsf{k}, \mathsf{tr}_C) = A \neq \bot \qquad \textit{and} \qquad (A, \xi) \in \mathsf{Req}_Z.$$

Proof. The inserted entries of $\overline{\mathsf{tr}}$ are queries to fs, and words under roots are the same in tr and $\overline{\mathsf{tr}}$. Let $\mathbf{T}_0^*$ and $\mathbf{T}_j^*$ be the words under rt_0 and rt_j. We apply Lemma A.9.

1. By part 2, neither $\mathsf{Coll}(\mathbf{T}_0^*, \varsigma_0)$ nor $\mathsf{Coll}(\mathbf{T}_j^*, \varsigma_j)$ holds.
2. By part 3 for $\mathsf{IOP}_Z^{\mathrm{key}}$, $\mathsf{Dec}(\mathbf{T}_0^*, \sigma_0) \neq \bot$.
3. By part 3 for $\mathsf{IOP}_Z^{\mathrm{link}}$, $\mathsf{Dec}(\mathbf{T}_j^*, \sigma_j) = \mathsf{Dec}(\mathbf{T}_0^*, \sigma_0) = A$.
4. By part 3 for $\mathsf{IOP}_Z^{\mathrm{req}}$, $(A, \xi) \in \mathsf{Req}_Z$.
5. By part 1, $\mathbf{T}_0^* = \mathsf{Wd}(\mathsf{tr}_C, \mathsf{rt}_0)$, since rt_0 is announced directly after tr_C. □

Lemma A.11 (Index-binding knowledge soundness over separate oracles). [machine-checked] *Assume the conditions of Definition A.7 other than (Hashed-key). Then* $\Pi^\circ_{Z,\mathsf{pv}}$ *is index-binding knowledge sound for* Req_Z*, in the general form of Definition 2.6 and Definition 2.7, over the separate oracles in the place of* H*. The extractor is* Ext°*, and the error is*

$$\varepsilon_{\mathrm{sep}}(q) \;=\; Q\,\varepsilon_{\mathrm{rd}} + \frac{2\,(Q+1)^2}{2^\lambda}, \qquad Q = 5\,(q + q_V + 1).$$

Proof. Join the committer, one query $\mathsf{ann}(\mathsf{rt}_0)$, the prover and V° into one algorithm, with at most $q + 1 + q_V$ queries. Modify it so that it makes the queries of the seed-first completion. This changes no output and yields the trace $\overline{\mathsf{tr}}$, with at most four further queries for every query: at most Q in all. If V° accepts while $\mathsf{Ext}^\circ(\mathsf{k}, \mathsf{tr}_C)$ is $\bot$ or is not related to ξ, then $\overline{\mathsf{tr}} \in \mathsf{Bad}$ by Lemma A.10. Lemma A.8, applied to the modified algorithm, bounds this probability. □

Lemma A.12 (One hash function). [machine-checked] *Let a bijection between bit strings and triples (oracle, query, number of an answer block) be given, under which an answer of a Merkle oracle over oracle strings is one block. If an argument over the separate oracles is index-binding knowledge sound with the extractor* Ext°*, then so is the argument that queries* H *through the bijection, with the same error and the extractor that translates the trace and applies* Ext°*.*

Proof. A uniform H induces independent uniform oracles, and a query to H is one block of one query to a separate oracle. The extractor reads only entries at oracles with one block. □

Lemma A.13 (Public readings). [machine-checked] *Let* Pub *be a deterministic oracle algorithm that repeats no query, and let* S *contain every string that* Pub *queries under some hash function. Let* Pub *and* S *be fixed before* H *is sampled. Suppose that, for every* pv *in the codomain of* Pub*:*

1. *the conditions of Definition A.7 hold at* pv*;*
2. (Spare) *under the bijection at* pv*, every string of* S *codes a query to the spare oracle;*
3. (Same-word) *the extractor of Lemma A.12 at* pv *depends on* pv *only through* dg_Z*.*

Consider the argument over H *with commitments* $(\mathsf{dg}, \mathsf{k})$ *whose verifier computes* $\mathsf{pv} = \mathsf{Pub}^H$*, rejects unless* dg *is the digest in* pv*, and then runs* V° *at* pv *through the bijection at* pv*. It is index-binding knowledge sound for* Req_Z *with error* $\varepsilon_{\mathrm{sep}}$*. Its extractor, at* $(\mathsf{dg}, \mathsf{k})$*, is that of Lemma A.12 at any* pv *with the digest* dg*, and* $\bot$ *if there is none.*

Proof. By induction on the queries of Pub. Its first query $\mathsf{str} \in S$ is fixed before H is sampled, and H can be sampled by first sampling $\mathsf{val} = H(\mathsf{str})$. Fix val, and give $(\mathsf{str}, \mathsf{val})$ to the committer and the prover as a constant. By (Spare), V° does not query str, and deleting the entries at str from a trace changes no word under a root. When no query is left, pv is a constant. The hypotheses hold at every pv, hence at this one, and Lemma A.11 and Lemma A.12 apply with the error $\varepsilon_{\mathrm{sep}}$. □

This step requires that Z is fixed before H is sampled: the induction resamples H at strings that no adversary chooses.

Lemma A.14 (The commitment as the hash of the key). [machine-checked] *Let* Π *be an argument over* H *whose commitments are keys and whose verifier is deterministic. Let* Π *be index-binding knowledge sound with extractor* Ext *and error* ε'*, and let* code *be an injective partial map from keys to bit strings. Consider the argument in which a commitment is* $c = H(\mathsf{code}(\mathsf{k}))$*, a proof contains* k*, and the verifier rejects unless* $H(\mathsf{code}(\mathsf{k})) = c$ *and then decides as the verifier of* Π *does. It is index-binding knowledge sound with error*

$$\varepsilon'(q) + \frac{(q+2)^2}{2^\lambda}.$$

Its extractor applies Ext *to the first key whose code the trace maps to* c.

Proof. Let k^* be the first key whose code the committer's trace maps to c, and let the verifier accept a proof with the key k. If $\mathsf{k} = \mathsf{k}^*$, the outcome is that of the experiment for Π with the committer that outputs k^*; this gives $\varepsilon'(q)$. Otherwise consider, in this order, the committer's queries, one query at c read as a bit string, the prover's queries and the verifier's query at $\mathsf{code}(\mathsf{k})$. The query at c is made in the analysis alone: it makes c a string that has occurred, for the case that the trace holds no k^*. Since code is injective, at some new entry of this list the answer equals a preceding answer or, read as a bit string, a preceding query or the entry's own query. After i entries this has probability at most $(2i+1)/2^\lambda$, and $\sum_{i<q+2}(2i+1) = (q+2)^2$. □

Lemma A.15 (The compilation lemma). [machine-checked] *Let Z be fixed before H is sampled. Assume the hypotheses of Lemma A.13, with* Pub *the public readings of V_Z, and let* code *be injective on pairs* $(\mathsf{dg}, \mathsf{k})$. *Let the public readings and the parser together make at most q_{pre} queries. Let* Rel *be an indexed relation such that, for all A, x, y and* fill,

$$(A, \mathsf{build}(x, y, \mathsf{fill})) \in \mathsf{Req}_Z \implies \mathsf{wt}(A) \text{ is defined and } (\mathsf{wt}(A), x, y) \in \mathsf{Rel}.$$

Then every argument whose verifier accepts only where the public readings, the parser and the compiled steps of V_Z accept is index-binding knowledge sound for Rel, *with error*

$$\varepsilon_{\mathrm{sep}}(q + q_{\mathrm{pre}}) + \frac{(q + q_{\mathrm{pre}} + 2)^2}{2^\lambda}.$$

Its extractor is that of Lemma A.14, formed from that of Lemma A.13, followed by wt.

Proof. Lemma A.14 replaces the commitment $(\mathsf{dg}, \mathsf{k})$ of Lemma A.13 by c. An instance (x, y), with fill from the proof, is mapped to $\xi = \mathsf{build}(x, y, \mathsf{fill})$; by the hypothesis, an instance outside Rel at $\mathsf{wt}(A)$ gives an encoded request outside Req_Z at A. The queries of the public readings and the parser are charged to the prover: q becomes $q + q_{\mathrm{pre}}$. □

Remark A.16 (Which form of the lemma). Two forms of the compilation lemma are machine-checked. One takes the public readings as constants and has sixteen hypotheses: the fifteen of Definition A.7 and the implication for greedy outputs. It does not apply to V_Z, and the proof of Theorem 5.1 does not go through it: the strings that V_Z hashes hold digests that it obtains from H. The other is Lemma A.15. It requires the fifteen at every value of the public readings, the implication for Rel in the place of the sixteenth, and further conditions; for $\mathsf{FlashProve}_Z$ each of these is a theorem, so that Theorem 5.1 has none of them as a hypothesis. Which form Lean's proof uses is known by a reading of the Lean code and by a program that follows the theorem's dependencies; no theorem states it. Limits of both forms: `lean/LIMITS.md`.

Lemma A.17 (The side conditions hold). [machine-checked] *The tag is for parts 2 and 3 alone: no Lean statement has the form of part 1 (the outline below). Let Z be a configuration and $\varepsilon_{\mathrm{rd}} = 2^{-128}$.*

1. *If (Sound-key), (Sound-link proof) and (Sound-request) hold for $\mathsf{FlashProve}_Z$, then all fifteen conditions of Definition A.7 hold at every value* pv *of the public readings.*
2. *The public readings of V_Z are deterministic, repeat no query and make at most eight queries. They satisfy (Spare) and (Same-word) of Lemma A.13.*
3. *The map* code *is injective on pairs* $(\mathsf{dg}, \mathsf{k})$.

Outline. No single Lean statement has the form of part 1: besides the three conditions, the Lean lemma behind it takes nine numeric premises, the numeric tests of (Layout) below. Further Lean lemmas prove these at every Z; they are joined with it only inside the proof of another Lean statement.

(No-decoding-at-a-collision) is step 1 of Definition 4.5. (Rare-collisions) is Lemma A.25: for every configuration, Lean's construction forms committed tables at the sizes of Z_0. (Hashed-key) is Lemma A.14. (Local), (Coherent), (Layout), (Answers-code) and parts 2 and 3 are properties of the verifiers and codings of Construction D.9, Construction D.11, and Construction 4.15, which Lean proves by unfolding them; (Layout) rests also on numeric tests, which Lean proves at every Z. Only four of the fifteen depend on pv. Within (Layout), the coding of a start query of $\mathsf{IOP}_Z^{\mathrm{req}}$ is injective by the encoded request's window and token positions alone, whatever the values of H'. (Spare) is domain separation between the strings of the public readings and the coded queries. □

A.3 Round-by-round soundness of the three IOPs

For sub-protocols, Lean states round-by-round soundness without a state function. The proofs of Lemma A.28 and Lemma A.30 are outlines: for the arithmetization and for two of the three IOPs the printed text is a guide to Lean's proof of Proposition 5.14, not an argument.

Definition A.18 (Doomed, reduction, leads). Consider k rounds, a set $\mathcal{B}$ of complete transcripts, and *counts* $\mathsf{cnt} = (\mathsf{cnt}_1, \ldots, \mathsf{cnt}_k) \in \mathbb{N}^k$. The prover is *doomed towards* $\mathcal{B}$ with these counts if

- $k = 0$ and the empty transcript lies in $\mathcal{B}$; or
- $k > 0$ and, for every first message msg, at most cnt_1 challenges ch_1 are *bad*: after $(\mathsf{msg}, \mathsf{ch}_1)$ the prover is not doomed, with counts $(\mathsf{cnt}_2, \ldots, \mathsf{cnt}_k)$, towards $\{\mathsf{trs} : (\mathsf{msg}, \mathsf{ch}_1, \mathsf{trs}) \in \mathcal{B}\}$.

The *error* of round m is cnt_m divided by the number of its challenges.

A *reduction* consists of rounds and a partial map out from an input statement and a complete transcript to an output statement. It *leads*, with counts cnt, from a condition $\mathcal{C}$ on inputs to a relation $\mathcal{R}$ between inputs and outputs if, from every input stm with $\mathcal{C}(\mathsf{stm})$, the prover is doomed, with these counts, towards the set of all trs such that $\mathsf{out}(\mathsf{stm}, \mathsf{trs})$ is undefined or $\mathcal{R}(\mathsf{stm}, \mathsf{out}(\mathsf{stm}, \mathsf{trs}))$ holds.

Lemma A.19 (State functions; composition). [machine-checked]

1. *If, outside $\mathcal{L}$, the prover of an IOP is doomed towards rejection with counts cnt, then the IOP is round-by-round sound for $\mathcal{L}$ (Definition A.1) with the errors of these counts.*
2. *Let Red_1 lead from $\mathcal{C}_1$ to $\mathcal{R}_1$ with counts cnt, and Red_2 from $\mathcal{C}_2$ to $\mathcal{R}_2$ with counts cnt'. If $\mathcal{C}_1(\mathsf{stm})$ and $\mathcal{R}_1(\mathsf{stm}, \mathsf{stm}')$ imply $\mathcal{C}_2(\mathsf{stm}')$, then Red_1 followed by Red_2 leads from $\mathcal{C}_1$ to the composition of $\mathcal{R}_1$ and $\mathcal{R}_2$, with counts $(\mathsf{cnt}, \mathsf{cnt}')$; likewise for a finite list.*

Proof. For part 1, let the state of a partial transcript be 0 if and only if the prover is doomed after it. Part 2 is by induction on the rounds of Red_1. □

Lemma A.20 (Proof of work; challenges from hash blocks). [machine-checked]

1. *Let Ch be a finite set and $\mathsf{Ch}' \subseteq \mathsf{Ch}$. The number of pairs $(a, z) \in \mathsf{Ch} \times \{0,1\}^\kappa$ with $z = 0$ and $a \in \mathsf{Ch}'$ is $|\mathsf{Ch}'|$.*
2. *Let a sub-protocol be stated with challenges in sets Ch_m, and let the prover be doomed towards $\mathcal{B}$ with at most a share shr_m of Ch_m bad in round m. In $\mathsf{FlashProve}_Z$ the challenge of round m is a tuple of blk_m hash blocks, from which an element of Ch_m and κ_m bits of proof of work are computed. Over these rounds the prover is doomed towards the transcripts whose image lies in $\mathcal{B}$ or in which a proof-of-work test fails, with round errors at most*

$$\left(1 + 2^{-128}\right)^{\mathsf{blk}_m} \mathsf{shr}_m \,/\, 2^{\kappa_m}.$$

Proof. For part 1, $a \mapsto (a, 0)$ is a bijection onto the pairs in question. For part 2, at most fib_m tuples of hash blocks are mapped to one element of Ch_m, where $\mathsf{fib}_m\,|\mathsf{Ch}_m|$ is at most $(1 + 2^{-128})^{\mathsf{blk}_m}$ times the number of tuples. By part 1, proof of work leaves the count unchanged, while the number of challenges grows by the factor 2^{κ_m}. □

The conditions of Definition A.6 do not mention proof of work: they bound the bad challenges among all challenges of a round. A further nonce is a further query, which counts in q.

In the next lemma the implications between claims run backwards: if the claim after a part holds, so does the claim before it. Informally, a false claim therefore remains false, except at bad challenges, until the last part refutes it.

Lemma A.21 (From three parts to a batch IOP). [machine-checked] *Let a batch IOP for at most R IOP instances consist of reductions Red_1, Red_2, Red_3: its first, shared and last part. Let $\mathcal{C}_{\text{first}}, \mathcal{C}_{\text{shared}}, \mathcal{C}_{\text{last}}$ be conditions, $\mathcal{P}_{\text{first}}$ a relation between inputs and outputs of Red_1, and $\mathcal{P}_{\text{shared}}$, $\mathcal{P}_{\text{last}}$ properties of the outputs of Red_1, Red_2. Suppose:*

1. *Red_1 leads from $\mathcal{C}_{\text{first}}$ to $\mathcal{C}_{\text{shared}} \wedge (\mathcal{P}_{\text{shared}} \Rightarrow \mathcal{P}_{\text{first}})$;*
2. *Red_2 leads from $\mathcal{C}_{\text{shared}}$ to $\mathcal{C}_{\text{last}} \wedge (\mathcal{P}_{\text{last}} \Rightarrow \mathcal{P}_{\text{shared}})$, with counts $\mathsf{cnt}^{\text{key}} + R\,\mathsf{cnt}^{\text{own}}$, where the challenges counted in $\mathsf{cnt}^{\text{key}}$ depend only on the external oracle string and its OOD sample;*
3. *Red_3 leads from $\mathcal{C}_{\text{last}}$ to $\mathcal{P}_{\text{last}}$;*
4. *every accepting transcript is a run of the three reductions on the early part of the IOP instance;*
5. *on an accepting transcript, $\mathcal{C}_{\text{first}} \Rightarrow \mathcal{P}_{\text{first}}$ implies membership in the IOP's set;*
6. *every round error is at most ε_{rd}.*

Then the batch IOP is round-by-round knowledge sound in the form of (Sound-request) (Definition A.6).

Proof. By Lemma A.19, from every input with $\mathcal{C}_{\text{first}}$ the prover is doomed towards $\mathcal{P}_{\text{first}}$: the last part gives $\mathcal{P}_{\text{last}}$, hence $\mathcal{P}_{\text{shared}}$, hence $\mathcal{P}_{\text{first}}$. By hypotheses 4 and 5 it is doomed towards the event that the verifier rejects or the IOP instance lies in the set. The three reductions run on the early part; hence the state function of Lemma A.19 is one at the early part, as Definition A.2 requires. In a batch, the challenges counted in $\mathsf{cnt}^{\text{key}}$ form one set for all encoded requests, which share the copy and its OOD sample. □

Lemma A.22 (Where a parameter check fails). [machine-checked] *Let Z be a configuration with $\mathsf{Chk}(Z) = 0$. Then (Sound-key), (Sound-link proof) and (Sound-request) of Definition A.6 hold for $\mathsf{FlashProve}_Z$ with $\varepsilon_{\mathrm{rd}} = 2^{-128}$.*

Proof. Each of the three verifiers rejects every transcript (Construction 4.13). □

Codes. Two bounds strictly below the Johnson radius are used: Lemma 5.13 and a bound on lists. Related work for Lemma A.24: [DP23]; further sources are in Appendix E.

Lemma A.23 (List size strictly below the Johnson radius). [machine-checked] *Consider a linear code over a field, with set of positions D and minimum distance at least $d_{\min}$ positions. Let $e \le |D|$ be a natural number with $|D|\,(|D| - d_{\min}) < (|D| - e)^2$. Let $\mathbf{T}^*$ be a table of words with any set of rows. Then at most*

$$\mathsf{ls}(|D|, d_{\min}, e) \;=\; \left\lfloor \frac{|D|\,(d_{\min} - e)}{(|D| - e)^2 - |D|\,(|D| - d_{\min})} \right\rfloor$$

tables of codewords differ from $\mathbf{T}^*$ in at most e columns.

Outline. This is the Johnson bound for the code of tables, with columns as symbols. □

Lemma A.24 (Folding a table). [machine-checked] *Let $|D|$ be a power of two, at most 2^{32}, let $\ell + t \ge 2$, and put $d_{\min} = |D| - (\ell + t) + 1$. Let $1 \le e' \le e < d_{\min}$ with $|D|\,(|D| - d_{\min}) < (|D| - e')^2$. Let $\mathbf{T}^*$ be a word (Definition 4.5), and $\sigma = (\varsigma, \mathbf{a})$ a sample at which $\mathsf{Coll}(\mathbf{T}^*, \varsigma)$ does not hold. Call $\gamma \in \mathbb{E}^v$ bad if some $u \in \mathbb{E}^{\ell+t}$ satisfies:*

1. *$\mathsf{Enc}(u)$ differs from $\sum_r \mathrm{eq}(\gamma, r)\,\mathbf{T}^*_r$ in at most e' positions;*
2. *the OOD answers of u at ς are $\sum_r \mathrm{eq}(\gamma, r)\,a_{r,i}$;*
3. *step 2 of Definition 4.5 gives no table, or a table whose messages, folded at γ, differ from the message of u.*

Let J be the bound of Lemma 5.13 at degree $\ell + t - 1$ and radius e'. Then the number of bad γ is at most

$$v\left(\lfloor J \rfloor + \mathsf{ls}(|D|, d_{\min}, e')\right) |\mathbb{E}|^{v-1}.$$

Proof. Call $\mathsf{z} \in \mathbb{E}$ *exceptional* for g_0, g_1 if it realizes the event of Definition 2.2 for $g_0 + \mathsf{z}\,g_1$.

Step 1: tables. Let g_0, g_1 be tables with a set Rw of rows, and read agreement as agreement of all rows on a common set of columns. For each exceptional z fix such a set S_{z} for $g_0 + \mathsf{z}\,g_1$. The vectors $a \in \mathbb{E}^{\mathsf{Rw}}$ for which $\sum_r a_r\,g_{0,r}$ and $\sum_r a_r\,g_{1,r}$ both agree with the code on S_{z} form a subspace. It is proper, since some row of g_0 or g_1 does not agree. There are at most $|\mathbb{E}|$ such subspaces. Each has at most $|\mathbb{E}|^{|\mathsf{Rw}|-1}$ elements, and all contain 0; hence some a lies in none. Every z that is exceptional for the tables is exceptional for the two words formed with this a, and the bound J holds for tables.

Step 2: folds. Folding 2^v rows with $\mathrm{eq}(\gamma, \cdot)$ is a sequence of v steps. Step k maps two half tables g_0, g_1 to $(1 - \gamma_k)\,g_0 + \gamma_k\,g_1 = g_0 + \gamma_k\,(g_1 - g_0)$, the line of Step 1 for the tables g_0 and $g_1 - g_0$; these agree with the code on a set of columns if and only if g_0 and g_1 do. At most $v\,\lfloor J \rfloor\,|\mathbb{E}|^{v-1}$ points γ are exceptional in some step.

Step 3: other points. If $\mathsf{Enc}(u)$ is within e' positions of the folded word, then $\mathbf{T}^*$ agrees with a table of codewords on the same columns, and u is the fold of that table. At most $\mathsf{ls}(|D|, d_{\min}, e')$ tables are within e' columns of $\mathbf{T}^*$ (Lemma A.23); since $e' \le e$, each lies in $\Lambda(\mathbf{T}^*)$.

Step 4: the list. Since there is no collision, such a table with the OOD answers $\mathbf{a}$ is the decoded table, which condition 3 excludes. For one with other answers, condition 2 is a nonzero multilinear equation in γ, with at most $v\,|\mathbb{E}|^{v-1}$ solutions. □

Steps 2 to 4 run as the second of the error bounds in the proof of [CFW26, version of February 25, 2026, Lemma 6.5], and for each variable the bound has the form of that lemma's second error. That work, in its Section 2.5, likens its analysis to those of [ACFY24b] and [BCFW25] and takes mutual correlated agreement from prior work; its own care, shared here, is to count the lists on the interleaved code.

Lemma A.25 (Collisions are rare). [machine-checked] *Let the sizes of a committed table, the radius e and the number s of OOD answers be those of Z_0. Then, for every word $\mathbf{T}^*$, the OOD seed that is read off uniform hash blocks is a collision of $\mathbf{T}^*$ (Definition 4.5) with probability at most 2^{-128}.*

Proof. A collision requires two of the at most ls members of $\Lambda(\mathbf{T}^*)$ with different messages and equal answers. The word $\mathbf{T}^*$ is over $\mathbb{F}$, and e is below the minimum distance. A codeword over $\mathbb{E}$ within e positions of a word over $\mathbb{F}$ agrees with it at $\ell + t$ or more positions of $D \subseteq \mathbb{F}$, and is therefore over $\mathbb{F}$. The difference of two members thus has its pad entries in $\mathbb{F}$, where the map from three entries to a pad is injective. Such a nonzero message has all OOD answers 0 at a share of at most shr of the OOD seeds. Lean's bound is $(1 + 2^{-128})^{\mathsf{blk}}\binom{\mathsf{ls}}{2}\mathsf{shr}$. At the sizes of Z_0, $s = 1$, a seed's point has 29 coordinates, $\mathsf{ls} = 26$ and $\mathsf{shr} = 29/|\mathbb{E}|$. Since $\binom{26}{2} \cdot 29 \cdot 2^{178} < p^3 = |\mathbb{E}|$, the bound is about 2^{-178}; Lean compares it with 2^{-178} and then with 2^{-128}. □

Lemma A.25 is in substance [CFW26, version of February 25, 2026, Lemma 9.1], at this protocol's code and OOD seeds; the share shr in the proof above corresponds to the evading error in its Lemma 9.3.

Sub-protocols. In Table 10, ls bounds the list of the oracle message or group at its radius e, mca is a count for mutual correlated agreement at that radius, ν is a number of positions, and van_o, van_b count the seeds at which a nonzero message has all answers 0.

Table 10: Bad challenges in a round of a sub-protocol (Remark A.26).

Round	Challenges	Bad, at most	Source
OOD seeds (Send, Code switch)	seeds	$\binom{\mathsf{ls}}{2}\,\mathsf{van}_\mathrm{o}$, summed over the messages	two members of a list answer alike
Batching	$\mathbb{E}$	the largest exponent	roots of a polynomial
Masked sumcheck, $\varkappa$	$\mathbb{E}$	1	a linear equation
Masked sumcheck, variable	$\mathbb{E}$; a further one, $\mathbb{E}\setminus\{0\}$	the degree of the round polynomial	roots of a polynomial
Sampled element on a line	$\mathbb{E}$	1	a linear equation
Sampled point, k coordinates	$\mathbb{E}^k$	$k\,\lvert\mathbb{E}\rvert^{k-1}$	zeros of a multilinear polynomial
Fold, $\varkappa$	$\mathbb{E}$	ls	one value for each member of the list
Fold, variable i	$\mathbb{E}$	$\mathsf{mca}_{i+1}+\deg\cdot\mathsf{ls}_i$, and mca of the two mask tables while these are folded	Lemma 5.13; the list after i variables
Positions (Code switch, Base case)	D^ν	$(\lvert D\rvert-e-1)^\nu$	distance above e
Code switch in a Proximity test	$D^\nu\times\mathcal{S}_\mathrm{b}$	$(\lvert D\rvert-e-1)^\nu\lvert\mathcal{S}_\mathrm{b}\rvert+\lvert D\rvert^\nu\,\mathsf{ls}\,\mathsf{van}_\mathrm{b}$	also the list, for the batching element
Base case, $\varkappa$	$\mathbb{E}$	$\mathsf{mca}+\mathsf{ls}$, the list for two rows	Lemma 5.13; the list
Lookup, ϖ	$\mathbb{E}$	$2^{k+1}-1$ for trees of depth k	roots of a polynomial
Lookup, joined tables	$\mathbb{E}$	the entries of the joined table	roots of a polynomial
Range test, Θ	matrices	the share 2^{-128} of all matrices	128 independent columns

Remark A.26 (Round counts). Table 10 is read off several machine-checked lemmas on reductions of the Lean library. It is not one Lean statement, and the proof of Theorem 5.1 uses only the common count of Lemma A.28. For Masked sumcheck, Fold, the Code switch of a Proximity test, Base case and Range test, Lean proves that the reduction leads (Definition A.18) from the reduction's condition to the relation that its witness map sends each witness of the output's claims to one of the input's. For OOD seeds and positions it proves the count alone; for Batching and Lookup the entries are the counts that Lean compares with 2^{-128}.

Six rows follow in form, as counts for this protocol's codes and not term for term, error terms of [CFW26, version of February 25, 2026]: OOD seeds, the first error of its Lemma 9.9; the two rows on Fold, the errors of its Lemma 6.5; Positions, the spot checks in its Lemmas 7.4 and 9.9; Code switch in a Proximity test, the other two errors of its Lemma 9.9 (in its overview, the second error of Lemma 2.17); Base case, the first error of its Lemma 7.4.

Of the two rows on Masked sumcheck, the first, on the challenge that joins mask and claim, has the form of the first error of [CFW26, version of February 25, 2026, Lemma 6.5], and the count for the further variable of the second, sampled off one point (there, off two), that of the third error of its Theorem 11.3, each without the list sizes; the count for the other variables is that of a sumcheck round [LFKN92].

The request.

Definition A.27 (Conditions and claims of $\mathsf{IOP}^{\mathrm{req}}_Z$). Fix Z, a word $\mathbf{T}^*_j$ and a sample $\sigma_j=(\varsigma_j,\mathbf{a}_j)$. At a point between two sub-protocols, the *condition* states that neither $\mathbf{T}^*_j$ nor an untested oracle message has a collision at its OOD seeds, and that the verifier's state has the form fixed for that point. $\mathcal{C}_\mathrm{first}$, $\mathcal{C}_\mathrm{shared}$, $\mathcal{C}_\mathrm{last}$ are the conditions at the three parts' starts.

$\mathcal{P}_\mathrm{first}$. Some cells and the token that the verifier holds at the first part's end satisfy each segment's identities, lookups and relaxed bound (Definition 4.8), at the early part of ξ and at $\mathsf{Dec}(\mathbf{T}^*_j,\sigma_j)$.

$\mathcal{P}_\mathrm{shared}$. The claim (h,ψ,η,b) on the table (Construction D.7) holds of $\mathsf{Dec}(\mathbf{T}^*_j,\sigma_j)$.

$\mathcal{P}_{\text{last}}$. $\langle \text{eq}(\psi, \cdot), w\rangle + \eta = b$ and $\langle \text{eq}(\psi^*, \cdot), w\rangle = \langle h, \omega\rangle$.

In the last two, w, ω and mask entries are what their oracle messages decode to, and the state records no failed test. The closed lists of claims and of states' forms are the Lean terms that Appendix F names for this definition (Remark D.10, Given in outline).

Lemma A.28 (The first part of $\mathsf{IOP}^{\text{req}}_Z$). [machine-checked] *Let Z be an admissible configuration with $\mathsf{Chk}(Z) = 1$. The first part of $\mathsf{IOP}^{\text{req}}_Z$, on the early part of ξ, leads (Definition A.18) from $\mathcal{C}_{\text{first}}$ to*

$$\mathcal{C}_{\text{shared}} \ \wedge \ \big(\mathcal{P}_{\text{shared}} \Rightarrow \mathcal{P}_{\text{first}}\big)$$

(Definition A.27). The count of each round m, with challenge set $\mathcal{H}_m$, is $\lfloor |\mathcal{H}_m|/2^{128} \rfloor$.

Outline. Lean fixes, for each point between two sub-protocols, the set of claims that are open there. It divides the sub-protocols into thirteen kinds: Send; Code switch; Lookup; the own rounds of layers, of attention and of the head; the round that sends y^+; the matrix of the Range test; Batching; Batching with Fold; the merging of the head's claims on the table; the Proximity test; and the reduction to one claim on the table.

For each kind it proves that a sub-protocol leads, from every state that the first part can reach and that meets the condition before it, to the condition after it and to the implication that, if the claims open after it hold, so do those open before it; in Lean an explicit map sends values that satisfy the former to values that satisfy the latter. By Lemma A.19 these compose, since what is open after one sub-protocol is open before the next.

The counts have the forms of Table 10, at the sizes that Z fixes; in the round that sends y^+ no challenge is bad. They are transferred to challenges from hash blocks by Lemma A.20 and compared with $|\mathcal{H}_m|/2^{128}$, as natural numbers; for the attention segments this comparison is itself a parameter check. Since a larger count gives a weaker statement, every round is bounded by the common count. What the lemmas require of Z is tested by Adm and Chk. □

Lemma A.29 (The shared part and the last part). [machine-checked] *Let Z be an admissible configuration with $\mathsf{Chk}(Z) = 1$.*

1. *Hypotheses 2 and 3 of Lemma A.21 hold for $\mathsf{IOP}^{\text{req}}_Z$, at the conditions and claims of Definition A.27, with every round error at most 2^{-128}.*
2. *In step 1 of Construction D.7, $\psi^* = (\psi_0^*, \gamma)$ is counted in $\mathsf{cnt}^{\text{key}}$ if γ is bad (Lemma A.24), and in $\mathsf{cnt}^{\text{own}}$ if $w \neq h^\top A$ and $\langle \text{eq}(\psi^*, \cdot),\, w - h^\top A\rangle = 0$. For $N_{\text{req}} \leq R$ requests the count is*

$$\big(v\,(\lfloor J \rfloor + \mathsf{ls}) + N_{\text{req}}\, n\big)\, |\mathbb{E}|^{n-1},$$

   with J and ls as in Lemma A.24.
3. *In the other shared rounds the counts hold for any number of requests.*

Proof. By $\mathcal{C}_{\text{shared}}$, $\mathbf{T}^*_j$ has no collision at ς_j, and Lemma A.24 applies. Let γ not be bad. Once the Proximity test on the group of u has established them, the constraints of steps 3 and 4 of Construction D.7 state conditions 2 and 1 of Lemma A.24 for u. Hence u is the fold of the decoded table. The constraint of step 7 then gives $\omega = A\,\text{eq}(\psi^*, \cdot)$, outside at most $k\,|\mathbb{E}|^{k-1}$ points χ, where $k = \lceil \log_2 d \rceil$.

If $w \neq h^\top A$, the constraint $\langle \text{eq}(\psi^*, \cdot), w\rangle = \langle h, \omega\rangle$ is a nonzero multilinear equation in ψ^*, with at most $n\,|\mathbb{E}|^{n-1}$ solutions. This set depends on the encoded request, and it is counted N_{req} times. With $w = h^\top A$, the first constraint of $\mathcal{P}_{\text{last}}$ is $\mathcal{P}_{\text{shared}}$. The other rounds have counts as in Table 10; Proposition D.8 evaluates steps 1 and 4 at Z_0. □

Lemma A.30 (The two other IOPs). [machine-checked]

1. *Let Z be an admissible configuration with $\mathsf{Chk}(Z) = 1$. Then (Sound-key) of Definition A.6 holds with $\varepsilon_{\text{rd}} = 2^{-128}$.*
2. *Let Z be any configuration. Then (Sound-link proof) holds with $\varepsilon_{\text{rd}} = 2^{-128}$.*

Outline. Part 1. Lean proves a lemma for each of the 21 sub-protocols of $\mathsf{IOP}^{\text{key}}_Z$ and composes them as for $\mathsf{IOP}^{\text{req}}_Z$, with a batch of one.

Part 2. $\mathsf{IOP}^{\text{link}}_Z$ is treated as a whole, with radii e_0 and e_1 for the two tables. The count for γ is the sum of the bounds of Lemma A.24 for the two tables and $v\,|\mathbb{E}|^{v-1}$. For the positions it is the larger of $(|D| - e_0 - 1)^{\nu_{\text{first}}}\,|D|^{\nu_{\text{copy}}}$ and $|D|^{\nu_{\text{first}}}\,(|D| - e_1 - 1)^{\nu_{\text{copy}}}$, and for χ it is $k\,|\mathbb{E}|^{k-1}$ with $k = \lceil \log_2 \ell \rceil$. □

Proof of Proposition 5.14. For $\mathsf{IOP}^{\text{key}}_Z$ and $\mathsf{IOP}^{\text{link}}_Z$ this is Lemma A.30. For $\mathsf{IOP}^{\text{req}}_Z$ we verify the hypotheses of Lemma A.21: the first holds by Lemma A.28, the second and third by Lemma A.29, the fourth by the definition of the verifier, the fifth by Lemma A.31, the sixth by the counts in these lemmas. □

A.4 From the relation to the model

Lemma A.31 (From the open claims to the request relation). [machine-checked] *Let Z be an admissible configuration with $\mathsf{Chk}(Z) = 1$. On every accepting transcript of $\mathsf{IOP}^{\mathrm{req}}_Z$, if $\mathcal{C}_{\mathrm{first}}$ implies $\mathcal{P}_{\mathrm{first}}$ (Definition A.27), then*

$$\mathsf{Dec}(\mathbf{T}^*_j, \sigma_j) = \bot \qquad \text{or} \qquad (\mathsf{Dec}(\mathbf{T}^*_j, \sigma_j), \xi) \in \mathsf{Req}_Z.$$

Proof. Let $\mathsf{Dec}(\mathbf{T}^*_j, \sigma_j) \neq \bot$. Then $\mathcal{C}_{\mathrm{first}}$ holds, by (No-decoding-at-a-collision), and $\mathcal{P}_{\mathrm{first}}$ provides cells. By step 4 for $\mathsf{IOP}^{\mathrm{req}}_Z$ in Construction D.9, the stored token is unchanged after each sub-protocol other than the one that receives it; hence the head's claims concern the token received. By step 3, that token is y^+. □

Remark A.32 (Stored token). The test of step 4 concerns the verifier's own state, not a prover message, and the proof above relies on it. That it changes no decision is known in part by a Lean theorem, by which the decision is the same for the written IOP and a second text of it, and in part by a reading, that the second text is the IOP without the test; at worst it costs completeness, not soundness.

Lemma A.33 (From the request relation to the policy). [machine-checked] *Let Z be an admissible configuration. Let A be in range, and let x, y, fill be lists of tokens such that $\xi = \mathsf{build}(x, y, \mathsf{fill})$ is defined and $(A, \xi) \in \mathsf{Req}_Z$. Then $\mathsf{wt}(A)$ is defined, and y meets pol_Z on x under the weights $\mathsf{wt}(A)$ (Definition 3.4), in the integer model at μ_Z.*

Proof. The relation is stated modulo p, for cells below 2^{18} in absolute value whose lookup keys are entries of public tables. The model parameters μ_Z are valid, since Z is admissible, and A is in range. Under these bounds the two sides of each identity are integers that differ by less than $2^{61} - 1 < p$ in absolute value; a multiple of p of that size is 0, and the identity holds over the integers. By induction on the groups of layers, the cells are the values of the integer model on the window, at the weights $\mathsf{wt}(A)$ and for every fill, and the model's range conditions hold. The head's identities and lookups then state the conditions of Definition 3.4. The bounds for the single identities are Lean's and are not printed. □

Lemma A.34 (The extractor). [machine-checked] *Let Z be an admissible configuration. Then Ext_Z (Construction 4.15) is the extractor of Lemma A.15.*

Proof. By unfolding the two definitions. □

A.5 Adding up the error

Lemma A.35 (Numbers of queries). [machine-checked] *Let Z be any configuration.*

1. *At every value of the public readings, V° makes at most $q_V \leq 2^{20}$ queries to the separate oracles.*
2. *The public readings of V_Z make at most 8 queries to H.*
3. *The parser, step 2 of V_Z, makes at most 1 query to H (Construction D.11).*

A query to a separate oracle costs one query to H for each block of its answer; hence q_V does not bound the queries of V_Z to H (Remark B.13). That $\varepsilon_{\mathrm{rd}}$ denotes 2^{-128} is a definition; that the round errors and the probability of a collision are at most this number is Proposition 5.14 and Lemma A.25.

Lemma A.36 (The arithmetic of the bound). [machine-checked] *Let $q, q_{\mathrm{par}}, q_V \in \mathbb{N}$ and $\varepsilon_{\mathrm{rd}} \geq 0$ with $\varepsilon_{\mathrm{rd}} \leq 2 \cdot 2^{-128}$ and $q_V + 9 \leq 2^{23}$. Put $\bar{T} = q + q_{\mathrm{par}} + 2^{23}$ and $Q = 5\,(q + 8 + q_{\mathrm{par}} + q_V + 1)$, and suppose that*

$$\frac{16\,\bar{T}}{2^{128}} + \frac{(\bar{T}+2)^2}{2^{256}} \;<\; 1.$$

Then

$$Q\,\varepsilon_{\mathrm{rd}} + \frac{2\,(Q+1)^2}{2^{256}} + \frac{(q+8+q_{\mathrm{par}}+2)^2}{2^{256}} \;\leq\; \frac{16\,\bar{T}}{2^{128}} + \frac{(\bar{T}+2)^2}{2^{256}}.$$

Proof. The hypotheses give $16\,\bar{T} < 2^{128}$, $Q \leq 5\bar{T}$ and $q + 8 + q_{\mathrm{par}} + 2 \leq \bar{T} + 2$. Multiplied by 2^{256}, the three terms on the left are at most $10\,\bar{T}\,2^{128}$, $2\,(5\bar{T}+1)^2 \leq 6\,\bar{T}\,2^{128}$ and $(\bar{T}+2)^2$. □

Proof of Theorem 5.1. Let Z be an admissible configuration.

Step 1. The three conditions on soundness hold with $\varepsilon_{\mathrm{rd}} = 2^{-128}$. If $\mathsf{Chk}(Z) = 1$, this is Lemma A.30 and, as in the proof of Proposition 5.14, Lemma A.21; otherwise it is Lemma A.22. Lemma A.17 gives the other hypotheses of Lemma A.15.

Step 2. Let Rel consist of the triples (M, x, y) such that y meets pol_Z on x under M. Lemma A.33 is the implication that Lemma A.15 requires, and Lemma A.34 identifies the extractor. By Lemma A.35, Π_Z is index-binding knowledge sound for Rel with the error (1).

Step 3. Where $\varepsilon(q) < 1$, Lemma A.36 with $q_{\mathrm{par}} = 1$, hence $\bar{T} = T$, bounds (1) by $\varepsilon(q)$. This proves Theorem 5.11.

Step 4. At exact match, an output that meets the policy is a greedy output (Proposition 3.5). □

B Further statements and proofs for Section 5

Here, in Appendix C and Appendix D, and at Proposition 3.12, Lemma 4.9, Lemma 4.18, Lemma 5.13, and Lemma A.35, some machine-checked statements have neither a printed proof nor an outline. Their ground is the kernel's check of the Lean statements that Appendix F names for them.

Proposition B.1 (Policies that every eligible output meets). [machine-checked] *Let μ be valid model parameters (Definition 3.6) and $\mathsf{pol} = (C, \theta, cap)$ an acceptance policy (Definition 3.4).*

1. *If $\mathsf{M}_F \le \mathsf{K}_F + 11$ and $\theta, C \ge 2436593409$, then, for all weights M, every output that is eligible under M meets pol under M. The policy $(2^{32} - 1,\ 2436593409,\ 0)$ is an instance.*
2. *Let $\mu = \mu_{Z_0}$ (Definition 3.8), $\theta \le C < 2^{32}$ and $1 \le n_{\mathrm{ct}} \le L - 1$. Then $\theta = C$ and $cap \ge n_{\mathrm{ct}}$ if and only if, for all weights M, every output with n_{ct} counted tokens that is eligible under M meets pol under M.*

Remark B.2 (All weights). What a clipped-gap policy excludes depends on the committed weights and on numbers that the provider writes into Z.

1. Up to *cap* counted tokens may be any tokens, where the others leave room in the sum (Remark 3.10, What a policy bounds).
2. Some policies of admissible configurations exclude no token (Proposition B.1). V_Z does not reject a configuration with such a policy, and the C++ verifier rejects some such policies but not every one: no verifier sees the model's logits.
3. At $\mathsf{M}_F \le \mathsf{K}_F + 11$ some threshold of a configuration with these exponents, if it is admissible (which has been evaluated at three parameter sets, not proved), excludes no token, whatever the weights; the C++ verifier never prints these two exponents.
4. The weights by which the proof of Proposition B.1 violates a policy are synthetic. Nothing follows for a trained model, under whose weights every eligible output may still meet such a policy.

Proposition B.3 (The example clipped-gap policy). [machine-checked] *Let $\mathsf{pol} = (25856, 512, 1)$, the policy of Z_1 (Definition 3.7), let μ be valid model parameters and $M \in \mathcal{I}$, and let y be an output that is eligible on a prompt x under M, with n_{ct} counted tokens and logit gaps gap_i (Definition 3.4).*

1. *If y meets pol and $n_{\mathrm{ct}} \le 99$, then $\mathrm{gap}_i < 25600$ for all $i < n_{\mathrm{ct}}$, and $\sum_{i<n_{\mathrm{ct}}} \mathrm{gap}_i \le 256\, n_{\mathrm{ct}}$.*
2. *If y meets pol, then for every integer $g \le 25600$,*

$$g \cdot \#\{\, i < n_{\mathrm{ct}} : \mathrm{gap}_i \ge g \,\} \ \le\ 256\, n_{\mathrm{ct}}.$$

3. *If $n_{\mathrm{ct}} \ge 100$ and all counted tokens but at most one are greedy tokens, then y meets pol.*

Remark B.4 (Reading the theorem: particulars). To the items of Remark 5.3:

3. No form of the theorem with a witness is stated, and Ext_Z reads nothing of the prover. It is straight-line, and no running time is stated.
4. The weights that Ext_Z reads off a commitment and the committer's queries need not be the same at all configurations.
5. Configurations with other policies are admissible.

Remark B.5 (Bits per query). The bound $\varepsilon(q)$ of Theorem 5.1 is at most about $16\,(q + 2^{23})/2^{128}$: 124 bits per query. It is never below about 2^{-101}, it is about 2^{-60} at $q = 2^{64}$, and it says nothing from about $q = 2^{124}$ on. A query returns 32 bytes. A call of BLAKE2b that returns 64 bytes, as each call for a Fiat–Shamir challenge does, an attempt at a proof of work among them, counts as two queries, and every other call as one. So 124 bits per query are at least 123 bits per call; this conversion is a reading, not a theorem, and it counts an attempt at a proof of work as a whole call, whereas in the released programs one attempt can cost less than one hash call, so that κ counted bits mean fewer than 2^{κ} calls; by how much is not known.

Remark B.6 (Acceptance). What is known beyond Limitation 8.9 (Completeness on served requests only): the runs in which the executable form accepted honest proofs are at Z_0 and Z_1, at the format's hash function (Section 7.3); no run concerns another hash function, and, besides the reduction of Remark 6.49, the one theorem on acceptance is over a random oracle and on the Lean prover's proofs (Section 6.5). Beyond Proposition 5.6, the parameter checks have been evaluated, and pass, at seven admissible configurations, over Lean files that differ from the released ones, by a report alone (`lean/GAPS.md`, gap 52; `lean/THEOREM.md`); at all others nothing is known. V_Z can reject on Z alone at further places, hundreds by a rough count, not traced one by one; lemmas settle two of them at Z_0, and no statement says that the list is complete. Where Z is admissible, $\mathsf{Chk}(Z) = 1$ and the policy is exact match, as at Z_0, Section 6.5 gives more: some request is served there, at constructed weights, and V_Z rejects the Lean prover's proof for it with probability at most 2^{-40} over the random oracle and the honest coins. So no test whose outcome is a function of Z and H' alone makes V_Z reject every input, and a test that also reads H at strings formed from Z does so with probability at most 2^{-40} over H; at the format's hash function nothing follows for such a test, and no lemma settles a test that only other requests meet. This is an inference from these statements; no Lean theorem states it. At other Z the question is open.

Remark B.7 (Which compiler). Two compilers occur: the Lean library's own (Construction D.11) and the generic compilation of Definition 2.4. Steps 4 to 9 of V_Z (Construction 4.15) are, by definition, the output of the first at a hand-written input (the three IOPs, the assignment of their rounds to hash queries, and the codings) that depends on Z and the digests of step 3. Lemma A.15 is machine-checked for this compiler, and Theorem 5.1 rests on it. That the first is an instance of the second is not machine-checked; Theorem 5.1 does not need it. In the Lean term for Π_Z the model, the committer and the prover are fields, the last two constants (Remark 4.16, Honest parties). A further Lean function, compile (Appendix F at 6.15; its output is the argument of Definition 6.15), which the release's page `lean/STATEMENTS.html` shows, is a layer around the first: it returns V_Z (by a proved equation); soundness, completeness and zero knowledge are all stated of its output; the soundness theorem is proved not through it but through the compilation of Construction D.11 (Lemma A.15). The library also has a second theorem with the statement of Theorem 5.1 to the letter (`sound_of_verify_at_through_the_IOP`), which draws the three protocols' soundness from the one round-by-round statement of Remark 6.21, item 1, and hands it to the same compilation lemma: it is not an independent proof.

B.1 The executable form

Construction B.8 (Algorithm of the executable form). *Coding.* A bit string is *plain* if its length is a multiple of 8 and its first eight bits do not spell the byte 4. For a plain bs, $\mathsf{byt}(\mathsf{bs})$ is the byte string with the bits of bs. Otherwise $\mathsf{byt}(\mathsf{bs})$ consists of the bytes 4 and $|\mathsf{bs}|$ mod 8 and of bs, extended by zeros to whole bytes. Bytes are read from the lowest bit, and a byte string of fewer than 32 bytes is extended by zeros.

Algorithm. On input Z, O, c, x, y and $\pi = (\mathsf{fill}, \pi_1, \ldots, \pi_5)$:

1. If $\mathsf{Adm}(Z) = 0$, return 0.
2. Let $o_0 = O(\mathsf{byt}(\mathsf{code}(Z)))$, and let $o_1, \ldots, o_k$, $k \le 7$, be the values of O at the codings of the further public strings of step 3 of Construction 4.15, formed with o_0 as the digest of Z.
3. If some o_i does not have 32 bytes, return $\perp$. If $\mathsf{Chk}(Z) = 0$, return 0.
4. Unpack $(\pi_1, \ldots, \pi_5)$, with o_0 and without a call to O, into k, j, rt_j, σ_j and the data of three proof strings. If this fails, return 0.
5. Evaluate steps 4 to 9 of Construction 4.15 as tests, with O in the place of the hash function: $\mathsf{build}(x, y, \mathsf{fill})$ is defined; k has a code; the hash value of that code is c; each of π_{key}, π_{link}, π_{req} can be formed from its data; the plain part of π_{req} holds j; ς_0 and ς_j are the seeds of rt_0 and rt_j; the three compiled verifiers accept. Return 1 if all hold, and 0 otherwise.

Corollary B.9 (Program verdict). [machine-checked]

1. *For all Z, all functions O from byte strings to byte strings and all c, x, y, π: if $E_Z^O(c, x, y, \pi) \neq \perp$, then $E_Z^O(c, x, y, \pi) = V_Z^{H_O}(c, x, y, \pi)$.*
2. *Let g be a tuple of eleven byte strings: the three files of a configuration Z_{g}, three files of the client, and five of a proof. Let claim and $\mathsf{verdict}$ be the two Lean functions that the command `lean-verify` evaluates on g, where $\mathsf{claim}(\mathsf{g})$ reads the client's files and is $\perp$ or a tuple (c, x, y, fill), and let π_{g} consist of fill and the five. Then $\mathsf{verdict}(\mathsf{g}) = \perp$ if $\mathsf{claim}(\mathsf{g}) = \perp$, and otherwise, with O^* of Assumption 4.10,*

$$\mathsf{verdict}(\mathsf{g}) \;=\; V_{Z_{\mathsf{g}}}^{H_{O^*}}(c, x, y, \pi_{\mathsf{g}}).$$

Proof of Theorem 5.5 and Corollary B.9. Part 1 follows the cases of Construction B.8. If $\mathsf{Adm}(Z) = 0$ or $\mathsf{Chk}(Z) = 0$, then V_Z rejects at every hash function. If π cannot be unpacked, V_Z rejects (at step 4, on the fixed tuple of

Construction D.11). Otherwise one lemma, the main step, equates the conjunction of these tests with the value of steps 2 to 9 at H_O. If every answer of O has 32 bytes, the algorithm does not return $\perp$; this holds of O^*, and verdict is claim followed by E^{O^*}. □

Remark B.10 (Claim from files). Z, c, x and y must come from the client. That claim reads the client's files as the format's specification intends is not proved, and an error there would make V_Z decide another claim than the client's. The proof needs no trust, nor does the program outside Lean that makes π from the release's proof files: Theorem 5.1 holds for every proof string.

A run also trusts the part of the command that reads files and prints, and Lean's evaluator, compiler and runtime. That none of the library's definitions beneath the executable form is replaced by other code when compiled rests on a walk in Lean over what the command runs, repeated whenever the library is compiled, which lists every constant there that carries replaced or external code (none is this library's). The command is part of the release's source tree, not a stand-alone program. In Corollary B.11 the functions O_H are defined with the axiom of choice, have no code and are not O^*.

Corollary B.11 (Soundness of the executable form). [machine-checked] *Let Z be as in Theorem 5.1. For each hash function H let O_H be a function from byte strings to 32-byte strings that induces H (Definition 5.4); such a family exists. Then the bound of Theorem 5.1 holds, with the same error $\varepsilon(q)$, for the experiment of Definition 2.7 in which the decision on (c, x, y, π) is $E_Z^{O_H}(c, x, y, \pi)$.*

Proof. By Theorem 5.5 the two experiments have the same event at every H. A family exists because byt is injective: let $O_H(\mathsf{byt}(\mathsf{bs}))$ be $H(\mathsf{bs})$, written as 32 bytes, and let O_H take any 32 bytes elsewhere. □

Remark B.12 (Malleable proofs). That an accepted proof cannot be altered into another accepted proof of the same instance is proved nowhere, and altered proofs were accepted in runs (Section 7.3). A false instance gains nothing by this: Theorem 5.1 bounds its acceptance whatever a proof holds.

B.2 Many proofs

Proof of Corollary 5.10. With the extractor of Theorem 5.1, for $i < N$ let $\mathcal{A}_{P,i}$ run the given prover and output entry i of its list, or a fixed triple; it makes the same queries. At every H and all random choices, the event of the corollary implies that of Definition 2.7 for some $\mathcal{A}_{P,i}$, which has probability at most $\varepsilon(q_C + q_P)$. □

Remark B.13 (Lists sent at once). In Corollary 5.10 the prover sees no decision of the verifier and no honest proof before it sends its list. For proofs that are sent one after another, each after the decision on the one before, no bound is stated. For lists sent at once, the bound $\varepsilon(q + Nq_H)$, without the factor N, is machine-checked if every run of V_Z makes at most q_H queries to H. Such a q_H is not known (Open problem 8.15): the bound q_V of Lemma A.35 counts queries to the separate oracles, each of which costs one query to H for each digest of its answer.

Corollary B.14 (Evaluation binding). [machine-checked] *Let Z and ε be as in Theorem 5.1, and let N, q_C, q_P, the committer and the prover be as in Corollary 5.10. With probability at most $N \cdot \varepsilon(q_C + q_P)$, V_Z accepts two entries i, j of the prover's list with $x_i = x_j$, $|y_i| = |y_j|$ and $y_i \neq y_j$.*

Proof. For fixed weights, prompt and length determine a greedy output. Of two accepted entries with equal prompts and lengths and different outputs, at most one lies in $\mathsf{Lang}(M^*)$, so the event of Corollary 5.10 occurs. □

Remark B.15 (Evaluation binding). Corollary B.14 has the form of evaluation binding for functional commitments [BNO21], for the function that maps a prompt and a length to an output and for two proofs in one list. It is neither extractability of the commitment nor the absence of two openings (Definition 2.10).

B.3 Other policies, parsers and the parameter checks

Theorem 5.11 is proved in step 3 of the proof of Theorem 5.1 (Appendix A.5), and Proposition 5.14 in Appendix A.3.

Corollary B.16 (Soundness with any parser). [machine-checked] *Let Z be as in Theorem 5.1. In V_Z (Construction 4.15), replace the parsing of step 2 by any deterministic oracle algorithm that makes at most q_{par} queries to H and maps x, y and a proof, from any set of proofs, to a tuple of the form that step 2 produces. Then the resulting argument is index-binding knowledge sound (Definition 2.7) for the relation of greedy outputs at μ_Z, under the same assumptions, with error*

$$\frac{16\,T_{\mathrm{par}}}{2^{128}} + \frac{(T_{\mathrm{par}} + 2)^2}{2^{256}}, \qquad T_{\mathrm{par}} = q + q_{\mathrm{par}} + 2^{23}.$$

At every admissible Z, the same holds for the relation of Theorem 5.11, with the extractor Ext_Z.

Proof. The proof of Theorem 5.1 uses of the parser only its number of queries (Lemma A.15 and Lemma A.36). □

Proof of Proposition 5.6. Admissibility and the policy of Z_0 follow from tests on its byte strings that Lean's kernel evaluates. $\mathsf{Chk}(Z_0) = 1$ is unfolded into closed facts, many of them proved by evaluation in the kernel (Remark 5.7, How the proposition is checked). Among admissible configurations Chk does not depend on the policy. □

Remark B.17 (Checkers). Lean's kernel accepts the proofs of the Lean theorems that the machine-checked statements of Section 5 and this appendix render or join, in a build of the released Lean files with no error and no `sorry`; for those named in `lean/CHECK-RESULTS.md` Lean prints no axiom beyond its three standard ones. No lemma about a step of the protocol and no conjecture on codes is assumed, and the random oracle and H' belong to the statements. An independent proof checker has accepted the soundness theorem on `verify`, `sound_of_verify_at` (the form of Theorem 5.1 that leaves the extractor open: Section 5, head), with every declaration beneath it, and nothing that does not lie beneath it, in a pass on that theorem's export alone: the pass was made over a copy of the compiled files of the released Lean files' build. Its passes by name on `sound`, `repliesFit` and `refusedHandsOutNoProof` were made over that build too; that the compiled files that these passes read were that build's is known from a report alone; Lean's own files were not compared with a list of checksums. The checker's other passes were made over Lean files that differ from the released ones. It has accepted neither Proposition 5.6 (Remark 5.7, How the proposition is checked) nor Theorem 5.17 nor Theorem 5.19. The Lean verifier's reference run (the 109 calls of Appendix H.3) has been made over the released Lean files; what these are, and what has run: `lean/CHECK-RESULTS.md`. One whole run of the library's full check (`lean/scripts/check.py`) over all but one of the released Lean files has the last line "All steps passed: tree, words, imports, build (kernel replay, audit). . . . ". That line also says that no theorem got a verdict on binders or on `Classical.choice`: every line of `lean/scripts/top-theorems.txt` says "general" (the whole line, and what the check does not cover: `lean/CHECK-RESULTS.md`). It is one run, on one machine. In its kernel replay Mathlib and the other packages are taken as they were compiled beforehand, and are not replayed (`lean/GAPS.md`, 81 (g)); which modules go by another call than `leanchecker`: Remark 5.7. One Lean file, with one definition and four theorems, which nothing else imports, was added after the runs reported here and checked on its own. Three of the check script's pinned values were changed and three of its stored lists were extended for it; nothing else in the script or the lists changed. The full check's run builds the three libraries from nothing in one call of Lake; the other results told here stand on a second build of the same files, from nothing, in several calls (`lean/CHECK-RESULTS.md`). The checker judges a text that this project's own exporter writes from the compiled library; the exporter is not in the repository, so a reader cannot repeat the pass; and no run shows the checker rejecting a wrong proof. The pinned version of Lean, 4.29.1, is older than the published fixes (from version 4.32.1 on) of soundness bugs in Lean's kernel and runtime and in the big-number library (GMP) that its Linux builds link. No proof in this library is known to use such a fault; the proofs have not been searched for one, and the check script's word list and audit would not detect one. This paper tells no check of the library with a later version of Lean (`lean/LIMITS.md`, L45).

Remark B.18 (Typed statement). Lemma 5.13 is the statement as typed in the Lean library. That it follows from the two-word case of the published theorem [BCHKS25, Theorem 4.6] rests on a reading; no theorem says it, and the lemma does not make that theorem, in its own form, formally verified. The Lean proof derives it from two theorems in files adapted from ArkLib [ArkLib] and Mathlib [Mathlib].

B.4 The commitment

Definition B.19 (Soundness after an honest commitment). Let $\Pi = (\mathsf{Rnd}, \mathsf{Commit}, P, V)$ and $\mathcal{I}$ be as in Definition 2.9, let $\mathcal{Q}$ be a predicate on triples (M, x, y) of an index, a prompt and an output, and let $\varepsilon\colon \mathbb{N} \to [0, \infty]$. For an index M and an oracle algorithm $\mathcal{A}_P$, the experiment is: sample the random oracle H; then

$$\mathsf{r} \leftarrow \mathsf{Rnd}^H, \qquad c = \mathsf{Commit}^H(M, \mathsf{r}), \qquad (x, y, \pi) \leftarrow \mathcal{A}_P^H(c, \mathsf{r}), \qquad \mathsf{acc} = V^H(c, x, y, \pi).$$

The argument Π is *sound after an honest commitment* for $\mathcal{Q}$ with error ε if, for every $M \in \mathcal{I}$, all $q_P \in \mathbb{N}$ and every q_P-query algorithm $\mathcal{A}_P$,

$$\Pr\big[\,\mathsf{acc} = 1 \text{ and not } \mathcal{Q}(M, x, y)\,\big] \;\le\; \varepsilon(q_P).$$

Remark B.20 (Composition). Suppose that the bound of Definition 2.7 holds for Π, a relation with languages $\mathsf{Lang}(\cdot)$, a given extractor Ext and an error ε; that Π has correctness of extraction for the same Ext with error $\varepsilon_{\mathrm{ext}}$; and that Rnd followed by Commit makes at most q_0 queries. Then Π is sound after an honest commitment (Definition B.19) for the predicate $(x, y) \in \mathsf{Lang}(M)$, with error

$$\varepsilon(q_0 + q_P) + \varepsilon_{\mathrm{ext}}.$$

The implication is machine-checked for every argument. Theorem 5.19 states this property of Π_Z^Γ.

Remark B.21 (Implications). For every argument and every extractor, two implications are machine-checked.

- Extractability of the commitment with error $\varepsilon_{\mathrm{op}}$ implies no two openings with error $2\varepsilon_{\mathrm{op}}$.
- If Rnd makes no query and outputs a function of one uniform number, and Commit makes at most q_0 queries on every input, it also implies correctness of extraction for the same extractor with error $\varepsilon_{\mathrm{op}}(q_0)$.

No two openings does not exclude two openings to one index with different randomness.

Remark B.22 (Proof outlines for the commitment). The outlines follow the Lean proofs.

- *Part 2 of Theorem 5.17 with γ (Proposition 5.18).* Where $\mathsf{Chk}(Z) = 1$, at every hash function the explicit committer outputs the commitment of a second committer, which is stated over the separate oracles of Construction D.11. This rests on five equations on single queries and two on the encoder and the OOD answers; the equation on leaves needs that both committers' leaf salts have one length, which $\mathsf{Chk}(Z) = 1$ gives. For the second committer, take the trace of $\mathcal{A}_C$ and then the traces of the opener and of the test that (M, r) opens c. With probability at most $\gamma(q_C + q_O)$, a hash event occurs on these traces, or the word under rt_0 changes when the other two traces are added. Otherwise c fixes the key, and Ext_Z reads off the trace of $\mathcal{A}_C$ the table that the test forms. The decoder returns the digit matrix of every honestly formed table (Definition 4.5).
- *Theorem 5.17.* Part 2 follows because $\gamma \le \beta$ wherever $\beta < 1$. Parts 1 and 3 follow from part 2 by Remark B.21, with $q_0 = 2^{35} + 2$.
- *Theorem 5.19.* Where $\mathsf{Chk}(Z) = 0$, V_Z rejects every input. Where $\mathsf{Chk}(Z) = 1$, part 1 is the implication of Remark B.20, applied to Theorem 5.11 with the extractor Ext_Z. Its other two inputs are part 1 of Theorem 5.17 and $q_0 = 2^{35} + 2$. Part 2 follows by Proposition 3.5.

Remark B.23 (Constants of the errors).

- The bound $2^{35} + 2$ on the explicit committer's queries is machine-checked, on every input. It is the argument of β in part 1 of Theorem 5.17 and a summand of q in Theorem 5.19: the honest committer's queries are counted among the adversary's.
- In β, the summand 2^{23} is the allowance for the verifier's queries in Theorem 5.1, although no verifier runs here.
- With $\gamma(2^{35} + 2)$ (Proposition 5.18) in the place of $\beta(2^{35} + 2)$, part 1 of Theorem 5.19 is likewise a single Lean theorem, joined with the theorem on admissibility as in Remark 5.20.

Proposition B.24 (Two degenerate committers). [machine-checked] *Let Z and Γ be as in Theorem 5.19.*

1. *If the explicit committer replaces every digit matrix by the all-zero matrix, none of the three properties of Theorem 5.17 holds, for Ext_Z and the errors stated there.*
2. *The* constant committer *always outputs 0^λ, without randomness or queries. With it in the place of the explicit committer, and any predicate on (M, x, y) in the place of "y meets pol_Z on x under M", part 1 of Theorem 5.19 holds with $q = q_P$ and without the term $\beta(2^{35} + 2)$.*

Remark B.25 (What the two committers show). All of the following is machine-checked.

- Part 2 of Proposition B.24: soundness after an honest commitment (Definition B.19) does not separate the explicit committer from the constant one. For the predicate that never holds, it states that V_Z accepts after the constant commitment with probability at most the error of Theorem 5.1.
- Part 1 holds in a stronger form: with the all-zero matrix, correctness of extraction fails for every extractor at every error below $1/2$, and no two openings for every error function whose value at $2^{35} + 2$ is below 1.
- With completeness, the property excludes that committer. If Π_Z^Γ with the all-zero matrix and a q-query prover is complete on the instances that can be served (Definition 2.12 and Definition 3.11) with error α, and sound after an honest commitment for greedy outputs with error ε, then $\alpha + \varepsilon(q) \ge 1$. Completeness is proved for no committer of Section 5.5.

C Proofs on zero knowledge

Appendix C.2 proves Theorem 6.3 and Proposition 6.5: Lemma C.8, Lemma C.9, and Lemma C.10 treat one table, and Lemma C.12 many tables together with runs of the IOPs. Appendix C.3 proves Theorem 6.6 and Corollary 6.7, and Appendix C.4 concerns the link proof.

C.1 The definition of index privacy

Remark C.1 (Environment and simulator). Definition 2.14 is adapted from [CFWpc]. Its reach and its limits are these.

- The environment sees the whole view of V^* and may decline, but it is a function of that view and has no coins of its own: no machine-checked statement extends a bound to environments with coins.
- Batches, with their revealed and hidden instances, are this paper's addition to the definition. The addition has had no independent check.

The simulator's running time is not bounded.

C.2 Committed tables

Definition C.2 (Disclosed view, honest-verifier simulator). Let P be a prover for an IOP (Definition 2.3) in which each oracle part is marked *masked* or not. Let ch denote the challenges of a run. The *abort set* of P at an input is a set of values of ch, the union of an input-independent *public part* and a *private part*.

The *disclosed view* at ch consists of the plain parts of P, its oracle parts that are not masked, and the symbols that the verifier reads in the masked ones and in the external oracle strings. The *output* of P at ch is $\perp$ if ch lies in the abort set, else the disclosed view.

An *honest-verifier simulator* is a randomized algorithm given the IOP instance, ch, and the symbols that the verifier reads in the external oracle strings. It is *exact* at an input and ch if its output is distributed as that of P.

The core of this notion, a view of the verifier and a simulator with limited access to the oracle strings, goes back to [BCFGRS17], as [CFW26, version of February 25, 2026, Definition 2.1] attributes it; here a prover may abort, and a simulator is exact at single values of ch.

Definition C.3 (Table system). A *table system* consists of a set of indices; an encoder from an index and randomness, uniform in a finite set, to a *table*, a string of columns; a finite set of OOD seeds with a map from index, randomness and seed to OOD answers; and three IOPs (Definition 2.3). Their external oracle strings are one table, two tables, and one table; their IOP instances are the table's OOD sample, both samples, and, in a batch IOP, pairs of a request and the sample. $\mathsf{FlashProve}_Z$ (Construction 4.13) is a table system whose indices are the matrices over $\mathbb{F}$ shaped as in Definition 4.1, encoded as in Definition 4.2 and Definition 4.3.

Definition C.4 (Four conditions on the tables). Let a table system (Definition C.3), a set $\mathcal{U}$ of OOD seeds and $\varepsilon_{\mathcal{U}} \in [0, \infty]$ be given.

(Bounded reads) The first verifier reads at most ν_{off} columns of its table, the second at most ν_{first} of its first table and ν_{copy} of its second, the third at most ν_{on} for a batch. For every seed in $\mathcal{U}$ and every set Q of at most t positions, the OOD answers with the columns at Q have the same distribution for all indices.

(First-table budget) $\nu_{\text{off}} + K\nu_{\text{first}} \le t$.

(Copy budget) $\nu_{\text{copy}} + N_{\text{c}}\,\nu_{\text{on}} \le t$.

(Good seeds) A seed derived from uniform digests lies outside $\mathcal{U}$ with probability at most $\varepsilon_{\mathcal{U}}$.

Definition C.5 (Table experiment). Let a table system (Definition C.3), provers $\mathbf{P}$ for its IOPs and an index A be given. In $\mathsf{Tab}^{\mathbf{P}}(A, \mathcal{A})$ an unbounded randomized algorithm $\mathcal{A}$ names an OOD seed ς_0; then $\mathbf{T}_0$ is sampled, and $\mathcal{A}$ receives its OOD answers. Next $\mathcal{A}$ adaptively asks for:

- *copy*: $\mathcal{A}$ names ς_j; then $\mathbf{T}_j$ is sampled with fresh randomness, and $\mathcal{A}$ receives its answers;
- *column*: $\mathcal{A}$ receives $\mathbf{T}_j[\phi]$ for j and ϕ of its choice;
- *run*: the first IOP on $\mathbf{T}_j$, the second on $(\mathbf{T}_0, \mathbf{T}_j)$ with $j \ge 1$, or the third on $\mathbf{T}_j$ and a batch of its choice is run with the honest verifier; $\mathcal{A}$ receives ch and the output of P_* (Definition C.2), in the third IOP for the revealed requests.

The output is that of $\mathcal{A}$. Tab_Z is the case of $\mathsf{FlashProve}_Z$ with constant-message provers.

Definition C.6 (Within the opening budget). Let $\nu_{\text{off}}, \nu_{\text{first}}, \nu_{\text{copy}}, \nu_{\text{on}}$ be as in (Bounded reads) of Definition C.4; for $\mathsf{FlashProve}_Z$ they are $62, 62, 62, 64$. An algorithm $\mathcal{A}$ in the table experiment (Definition C.5) is *within the opening budget* t if there are natural numbers K and $N_j^{\text{key}}, N_j^{\text{link}}, N_j^{\text{req}}, N_j^{\text{col}}$, for $0 \le j \le K$, such that, for every j,

$$\nu_{\text{off}}\, N_j^{\text{key}} \;+\; [\,j = 0\,]\;\nu_{\text{first}} \sum_{i \le K} N_i^{\text{link}} \;+\; \nu_{\text{copy}}\, N_j^{\text{link}} \;+\; \nu_{\text{on}}\, N_j^{\text{req}} \;+\; N_j^{\text{col}} \;\le\; t,$$

and in every execution $\mathcal{A}$ asks for at most K copies and, for each j, at most N_j^{key} runs of the first IOP on $\mathbf{T}_j$, N_j^{link} of the second on $(\mathbf{T}_0, \mathbf{T}_j)$, N_j^{req} of the third on $\mathbf{T}_j$, and N_j^{col} columns of $\mathbf{T}_j$.

Remark C.7 (The budget covers a table's whole use). The inequality of Definition C.6 counts every column of $\mathbf{T}_j$ that any run ever opens. The numbers 62 and 64 are upper bounds; where $\mathsf{Chk}(Z) = 0$ the verifiers of $\mathsf{IOP}_Z^{\mathrm{key}}$ and $\mathsf{IOP}_Z^{\mathrm{req}}$ read no column. With one offline proof, K link proofs and N_{c} online proofs on each copy it reads $62 + 62K \le t$ and $62 + 64\,N_{\mathrm{c}} \le t$, that is, $K \le 710{,}324$ and $N_{\mathrm{c}} \le 688{,}127$.

Lemma C.8 (Columns are uniform). [machine-checked] *Let $f_r \in \mathbb{F}^\ell$ for $r < 2^v$ be arbitrary messages, and let $Q \subseteq D$ with $|Q| \le t$. Over independent uniform $\tau_r \in \mathbb{F}^t$, the values $\mathsf{Enc}(f_r; \tau_r)(\phi)$, for $r < 2^v$ and $\phi \in Q$, are independent and uniform in $\mathbb{F}$.*

Proof. By Definition 4.2, $\mathsf{Enc}(f;\tau)(\phi) = \tau(\phi) + \phi^t f(\phi)$, where $\tau(X) = \sum_{j<t} \tau_j X^j$. The linear map $\tau \mapsto (\tau(\phi))_{\phi\in Q}$ from $\mathbb{F}^t$ to $\mathbb{F}^Q$ is given by a Vandermonde matrix on $|Q| \le t$ distinct points, so it is surjective. Hence $(\tau(\phi))_{\phi\in Q}$ is uniform, also after the fixed vector $(\phi^t f(\phi))_{\phi\in Q}$ is added. Rows have independent coefficients. □

Lemma C.9 (Pads mask the answers). [machine-checked] *Let $s = 1$. For every OOD seed (z_0, β_0) and all plain entries of a row, the map from the row's three pads to its OOD answer (Definition 4.3) is a bijection from $\mathbb{F}^3$ onto $\mathbb{E}$. Hence, over uniform pads, the OOD answers of a table are independent and uniform in $\mathbb{E}$, for every matrix.*

Proof. By Definition 4.3, $a_{r,0}$ is the sum of $\beta_0\,\mathsf{pad}_{r,0}$ and a term that depends on the plain entries and on z_0 alone. The three pads are the coordinates of $\mathsf{pad}_{r,0}$ in the basis $1, \vartheta, \vartheta^2$ of $\mathbb{E}$ over $\mathbb{F}$, and $\beta_0 \neq 0$. Hence pads map to $a_{r,0}$ bijectively. Rows have independent pads. □

Masking an OOD answer (Definition 4.3) with fresh random entries of the message follows the private zero-evaders of [CFW26, version of February 25, 2026, Section 9.2]: Lemma C.9 is in substance the privacy part of its Lemma 9.3, with its Definition 9.2, at one answer in $\mathbb{E}$ with three pads in $\mathbb{F}$.

Lemma C.10 (One table). [machine-checked] *Let $t = 44{,}040{,}191$ and $s = 1$, at the table sizes of the default configuration. For every OOD seed and every $Q \subseteq D$ with $|Q| \le t$, the distribution of the pair of the OOD answers and the columns $(\mathbf{T}[\phi])_{\phi\in Q}$ of a committed table $\mathbf{T}$ for A is the same for all matrices A over $\mathbb{F}$.*

Proof. Fix the pads, and with them all messages f_r and all answers. By Lemma C.8 the columns at Q are uniform, whatever the pads are. Hence the columns are independent of the pads, and so of the answers. By Lemma C.9 the answers are uniform; so is the pair, for every A. □

Lemma C.11 (Column counts). [machine-checked] *Let Z be admissible. The verifier of $\mathsf{IOP}_Z^{\mathrm{key}}$ reads at most 62 columns of its table, that of $\mathsf{IOP}_Z^{\mathrm{link}}$ at most 62 of each of its two tables, and that of $\mathsf{IOP}_Z^{\mathrm{req}}$ at most 64 for a batch. With Lemma C.10, $\mathsf{FlashProve}_Z$ satisfies (Bounded reads) of Definition C.4 at all OOD seeds.*

Lemma C.12 (Tables and runs before compilation). [machine-checked] *Let a table system satisfy (Bounded reads) at all OOD seeds (Definition C.4), and let $\mathcal{Y}$ be a set of pairs of an index and a request. Let $\mathbf{P}$ be provers whose external oracle strings are the tables of their input. An input is* valid *if its IOP instance holds the OOD samples of its tables and its requests form pairs of $\mathcal{Y}$ with its index. Let each Sim_* be exact, at every valid input, except with probability ζ_*^{sim} over uniform ch (Definition C.2). Fix an index A_0, and let $\mathsf{Tab}^{\mathsf{Sim}}$ be the table experiment (Definition C.5) with Sim_*, on the samples and opened columns, in place of each run of P_*. For every index A and every $\mathcal{A}$ within the opening budget (Definition C.6) whose requests form pairs of $\mathcal{Y}$ with A, at most $R_{\mathrm{rev}} \ge 1$ of a batch being revealed,*

$$\mathsf{SD}\big(\mathsf{Tab}^{\mathbf{P}}(A,\mathcal{A}),\ \mathsf{Tab}^{\mathsf{Sim}}(A_0,\mathcal{A})\big) \ \le\ \zeta_{\mathrm{key}}^{\mathrm{sim}} \sum_j N_j^{\mathrm{key}} + \zeta_{\mathrm{link}}^{\mathrm{sim}} \sum_j N_j^{\mathrm{link}} + R_{\mathrm{rev}}\,\zeta_{\mathrm{req}}^{\mathrm{sim}} \sum_j N_j^{\mathrm{req}}.$$

Proof. Step 1: simulated runs on the true tables. Replace each run of P_* by Sim_*, given the true OOD samples and the true columns that the verifier reads. One replacement changes the distribution by at most ζ_*^{sim}, or $R_{\mathrm{rev}}\,\zeta_{\mathrm{req}}^{\mathrm{sim}}$ for a batch; the sum is the stated bound.

Step 2: one reader. $\mathcal{A}$ with the simulators is one randomized algorithm that depends on the tables only through their OOD answers and columns that it reads adaptively, at most t of each table by (Bounded reads) and Definition C.6.

Step 3: all tables at the start. Sample one independent table for every pair (j, ς) at the start; when $\mathcal{A}$ names ς_j, use the table of (j, ς_j). Since $\mathcal{A}$ names one seed for each j, the distribution is unchanged.

Step 4: from fixed to adaptive positions. The probability of a transcript of the reader is the product of the probabilities of its own coins and, for each table, of the event that the answers and the columns at the transcript's positions take the transcript's values. These positions form a fixed set of size at most t, so by (Bounded reads) the probability is the same for A and A_0. □

Steps 1 and 2 of the proof above, which replace the runs by simulators and take $\mathcal{A}$ with these as one reader, follow the composition of honest-verifier zero knowledge in [CFW26, version of February 25, 2026, Definitions 4.4 and 4.7 and Theorem 4.5].

Definition C.13 (Table game). Let $\mathcal{U}$ be a set of OOD seeds, and let K, N_j^{key}, N_j^{link}, N_j^{req}, N_j^{col} be numbers that satisfy the inequality of Definition C.6. The *table game* is the table experiment (Definition C.5) that answers $\perp$ beyond these numbers and at seeds outside $\mathcal{U}$, with one request in a batch. In each run an *auxiliary value* is sampled from a fixed distribution, which may depend on the IOP, the table's number and the IOP instance; ch, or $\perp$, is a function of it. $\mathcal{A}$ receives the auxiliary value and, unless $\mathsf{ch} = \perp$, the output at ch.

Lemma C.14 (Tables and runs in the table game). [machine-checked] *In the setting of Lemma C.12, let (Bounded reads) hold for $\mathcal{U}$ in place of all OOD seeds. Fix an index A, and let each Sim_* be exact except with probability ζ_*^{sim} under the distribution of ch in the table game (Definition C.13), at the valid inputs with index A and seeds in $\mathcal{U}$. For every index A_0 and every $\mathcal{A}$ that asks for at most N runs of the third IOP, at requests that form pairs of $\mathcal{Y}$ with A, the outputs of the game at A with $\mathbf{P}$ and of the game at A_0 with Sim_* are within statistical distance*

$$\zeta_{\mathrm{key}}^{\mathrm{sim}} \sum_j N_j^{\mathrm{key}} + \zeta_{\mathrm{link}}^{\mathrm{sim}} \sum_j N_j^{\mathrm{link}} + N\, \zeta_{\mathrm{req}}^{\mathrm{sim}}.$$

Proof. As for Lemma C.12; Step 1 uses the hypothesis on Sim_* at A alone. □

Proof of Theorem 6.3. By Lemma C.11, (Bounded reads) holds at all seeds. The simulator that outputs the fixed messages and the symbols that it is given is exact at every ch. Let $\mathcal{Y}$ hold all pairs and $A_0 = A$. By Lemma C.12, $\mathsf{Tab}_Z(A, \mathcal{A})$ and $\mathsf{Tab}_Z(A', \mathcal{A})$ both have the distribution of $\mathsf{Tab}^{\mathsf{Sim}}(A_0, \mathcal{A})$. □

Proof of Proposition 6.5. Suppose that the distributions agree for some Q. The value that A' produces with all pads and random coefficients 0 then has positive probability for $A = 0$, at some pads and $(\tau_r)_r$. At ς° the entry $f_0[0]$ has coefficient $\mathrm{eq}((1, \dots, 1), 0) = 0$, and the other plain entries of f_0 vanish for both matrices. Hence both answers $a_{0,0}$ are images of the pads alone, and by Lemma C.9 the pads of f_0 are 0 for A too. The codeword of f_0 is then the evaluation of X^t for A' and of $\tau_0(X)$ for A. They agree on Q, so the nonzero polynomial $X^t - \tau_0(X)$ of degree t has $t+1$ roots, a contradiction. □

Remark C.15 (What tightness shows). Proposition 6.5 negates Lemma C.10 at $t+1$ columns. It follows, though not machine-checked, that the conclusion of Theorem 6.3 fails for the algorithm that names ς°, asks for the columns at Q and outputs what it receives. It bounds no distance and concerns one seed, at which $\mathrm{eq}((1, \dots, 1), j) = 0$ for every $j < \ell_0$. It is not shown that $\mathsf{wt}(A) \neq \mathsf{wt}(A')$.

Proposition C.16 (Linear equations beyond the opening budget). [argued on paper] *Let $\mathbf{T}$ be a committed table with row messages f_r (Definition 4.2), and let $Q \subseteq D$ be a set of $t + k$ positions with $1 \le k \le d\, 2^{n_0}$. There are a matrix $\mathsf{mat}_Q \in \mathbb{F}^{k \times \ell}$ and a linear map $\mathsf{lin}_Q \colon \mathbb{F}^Q \to \mathbb{F}^k$, both depending on Q alone, such that:*

1. *$\mathsf{mat}_Q\, f_r = \mathsf{lin}_Q\big((\mathbf{T}[\phi]_r)_{\phi \in Q}\big)$ for every row r and all random coefficients;*
2. *the first k columns of mat_Q, which belong to digits of A, are linearly independent;*
3. *the row space of mat_Q contains a subspace of dimension at least $k - 3s$ whose vectors vanish at the $3s$ pads.*

Hence the columns at Q determine k independent linear equations in the digits and pads of each row, and at least $k - 3s$ in its digits alone.

Proof of Proposition C.16. Row r of $\mathbf{T}$ is the evaluation on D of $g_r(X) = \tau_r(X) + X^t f_r(X)$, where $\deg \tau_r < t$ and $f_r(X) = \sum_{j<\ell} f_r[j]\, X^j$.

Step 1: the random coefficients cancel. Let

$$\Upsilon \;=\; \Big\{ v \in \mathbb{F}^Q \;:\; \sum_{\phi \in Q} v_\phi\, \phi^j = 0 \text{ for all } j < t \Big\}.$$

The $t \times (t+k)$ Vandermonde matrix on Q has rank t, so $\dim \Upsilon = k$. For $v \in \Upsilon$,

$$\sum_{\phi \in Q} v_\phi\, g_r(\phi) \;=\; \sum_{j<\ell} \mathsf{c}_j(v)\, f_r[j], \qquad \mathsf{c}_j(v) = \sum_{\phi \in Q} v_\phi\, \phi^{t+j}.$$

For a basis $v^{(1)}, \dots, v^{(k)}$ of Υ, let mat_Q have the rows $(\mathsf{c}_j(v^{(i)}))_{j<\ell}$, and let lin_Q map $(a_\phi)_{\phi \in Q}$ to $\big(\sum_\phi v_\phi^{(i)} a_\phi\big)_{i \le k}$. This proves part 1.

Step 2: independence. If $v \in \Upsilon$ and $\mathsf{c}_j(v) = 0$ for all $j < k$, then $\sum_\phi v_\phi\, \phi^j = 0$ for all $j < t + k$, and $v = 0$ since the square Vandermonde matrix on Q is invertible. Since $k \le d\, 2^{n_0}$, the entries $f_r[j]$ with $j < k$ are digits. This proves part 2.

Step 3: eliminating the pads. The map $v \mapsto (\mathsf{c}_j(v))_{\ell_0 \le j < \ell}$ from Υ to $\mathbb{F}^{3s}$ has a kernel Υ' of dimension at least $k - 3s$. This proves part 3. The entries of f_r between $d\, 2^{n_0}$ and ℓ_0 are 0, so the equation of $v \in \Upsilon'$ involves digits alone; by Step 2 these equations are independent. □

Remark C.17 (Few further columns). Proposition C.16 rests on the printed proof alone, which is not machine-checked and has had no independent check. For $1 \le k \le 3s$ each of the k equations may involve the pads. At ς° (Proposition 6.5) the OOD answers are the pads, so the k equations are independent equations in the digits alone, for every $k \ge 1$. It is not shown that this holds at every OOD seed.

C.3 After compilation

Theorem C.25 reduces index privacy of a whole-key argument (Definition C.20) to twenty conditions: four on the tables (Definition C.4), eight on nonces and positions (Definition C.22), three on formats (Definition C.23) and five on the provers (Definition C.21). Proposition C.26 gives the first fifteen for $\mathsf{FlashProve}_Z$ at admissible Z and bounded K and N_c, Lemma C.28 the five for constant-message provers, and Lemma C.27 bounds the error where $\zeta_*^{\mathrm{sim}} = \zeta_*^{\mathrm{ab}} = 0$. The idea of this step, from honest-verifier zero knowledge of a public-coin IOP to zero knowledge of the argument compiled from it, is public: compare [CF24, Lemma 2.20 (informal) and Lemma 9.9] and [CO25, Section 2.6, Corollary 1 (informal), with Theorem 7.1 for interactive proofs].

Definition C.18 (Compiled run of a prover). Let P_* be a prover for an IOP of a table system, with a *search limit* for each round with proof of work. A *compiled run* of P_* computes a proof string, over the oracles of Construction D.11.

- The chain salt is sampled; challenges are derived as the compiled verifier recomputes them.
- In a round with proof of work the nonces $0, 1, 2, \ldots$ up to the search limit are tried, one query each, until the challenge passes.
- Oracle parts are committed with the leaf salts of Definition C.19.

The result is $\perp$ if a search exhausts its limit or the challenges lie in the abort set.

Definition C.19 (Coins of a compiled run). Let $\ell_s \ge 0$. In a compiled run of P_* (Definition C.18):

- a leaf salt over an oracle part is a *public prefix*, a function of a value of the coins of P_*, the round, the component and the chain salt, followed by ℓ_s uniform bits;
- a leaf without a position has a fixed public body;
- the leaf number in the batch tree is fixed and public, and the path's digests are uniform.

A leaf salt of a committed table is a fixed public prefix followed by ℓ_s uniform bits.

Definition C.20 (Whole-key argument). Let Z be a configuration, and fix values $(\mathsf{dg}, \mathsf{par})$ in place of the two public readings of V_Z (Construction 4.15). Let $\mathbf{P} = (P_{\mathrm{key}}, P_{\mathrm{link}}, P_{\mathrm{req}})$ be provers for the three IOPs of $\mathsf{FlashProve}_Z$, with search limits and ℓ_s (Definition C.18 and Definition C.19), and $K \ge 1$. The *whole-key argument* $\Pi_Z^{\mathrm{wk}}[\mathbf{P}]$ has the matrices over $\mathbb{F}$ in the place of indices (Definition C.3), encoded requests ξ in the place of instances, and the separate oracles of Construction D.11, independent and random.

- The committer samples $\mathbf{T}_0, \ldots, \mathbf{T}_K$ with independent randomness, hashes each into a salted Merkle tree with root rt_j, computes $\varsigma_j = \mathsf{Seed}(\mathsf{rt}_j)$ and σ_j, and makes one compiled run of P_{key} and K of P_{link}. The commitment is $\big(\mathsf{rt}_0, \sigma_0, \pi_{\mathrm{key}}, (\mathsf{rt}_j, \sigma_j, \pi_{\mathrm{link}}^{(j)})_{1 \le j \le K}\big)$.
- The stateless prover samples j uniformly, makes a compiled run of P_{req} on $\mathbf{T}_j$ and one encoded request, and outputs (j, π_{req}). On larger batches it outputs $\perp$.
- The verifier consists of steps 6 to 9 of V_Z at $(\mathsf{dg}, \mathsf{par})$.

Definition C.21 (Five conditions on the provers). For $\mathbf{P}$ as in Definition C.20, let Sim_* be honest-verifier simulators (Definition C.2), $\mathcal{J}$ a set of matrices, $\mathcal{Y}$ a set of pairs (A, ξ), and $\zeta_*^{\mathrm{sim}}, \zeta_*^{\mathrm{ab}} \in [0, \infty]$. Probabilities are over uniform ch with proof-of-work bits 0. The conditions are required for all table randomness, OOD seeds and copy numbers.

(Encoded tables) For all matrices: the external oracle strings of P_* are the committed tables of its input.

(Exact simulation) For $A \in \mathcal{J}$, $(A, \xi) \in \mathcal{Y}$: except with probability ζ_*^{sim}, ch lies in the private part of the abort set or Sim_* is exact.

(Rare private aborts) For the same A, ξ: ch lies in the private part with probability at most ζ_*^{ab}.

(Readable prefixes) For $A \in \mathcal{J}$ and all ξ: unless P_* aborts, its plain parts determine the value of its coins in the public prefixes (Definition C.19).

(Uniform coins) All coins are uniform and independent, as in Definition C.19; salts have the compiled verifier's lengths.

Definition C.22 (Eight conditions on nonces and positions). Let a table system and data for the compiled verifiers of its three IOPs (Construction D.11) be given. A condition stated for each IOP counts three times.

(Nonce-free positions) For each IOP: the positions read are independent of the nonces.

(Nonces readable) For each IOP: a plain part determines its nonce, up to the search limit.

(Plain seeds) The two queries of Seed have the nonce 0.

(Local verifiers) Each decision depends on the oracle strings only at the positions read.

Definition C.23 (Three conditions on formats). In the setting of Definition C.22:

(Round formats) The rounds are own rounds, then shared rounds, then own rounds, beginning with an own round. In an own round the shared component is empty; in a shared round other than the first, the own components are.

(Coherent rounds) The plain part of a message is the pair of its components' plain parts, and each oracle position is answered from one component alone.

(Consistent layout) A uniform oracle answer gives a uniform challenge, of which the proof-of-work bits are a separate coordinate. Each position of an oracle part is one leaf of one tree of its component. A table fits its Merkle tree, position ϕ at leaf ϕ, and a batch its tree. Symbols in leaves can be read back. Openings and tree depths are within declared bounds. The code of the OOD answers into digests is injective and of fixed length.

Definition C.24 (Hashing error). In the setting of Definition C.20, the queries of one run of the committer and N of the prover are counted at *names* o: the oracles of Construction D.11 and, for each IOP and $i \geq 2$, a further name for attempt number i of all proof-of-work searches of that IOP.

- m_o is the number of queries at o that contain a secret string: all leaves of Merkle trees over oracle strings; attempt number 1 of each round's query to fs; the further attempts, up to the search limits; each compiled run's query to g, leaf and inner nodes in the batch tree.
- e_o is $2(K+1)$ at the oracle of Seed and 0 elsewhere.
- b_o is ℓ_s at such a leaf, the minimum of λ and the length of the chain salt at a query to fs, and λ elsewhere.

The *hashing error* is

$$\zeta_{\mathrm{hash}}(q,N) \;=\; q \cdot \max_o \frac{\mathsf{m}_o}{2^{\mathsf{b}_o}} \;+\; \sum_o \frac{\mathsf{m}_o\,(\mathsf{e}_o+\mathsf{m}_o)}{2^{\mathsf{b}_o}},$$

over names with $\mathsf{m}_o \neq 0$.

Theorem C.25 (Reduction of index privacy to twenty conditions). [machine-checked] *Let a table system (Definition C.3), data for its compiled verifiers, provers* $\mathbf{P}$*, simulators* Sim_**, and* $\mathcal{J}$*,* $\mathcal{Y}$*,* $\mathcal{U}$*,* K*,* N_c*,* ζ_*^{sim}*,* ζ_*^{ab}*,* $\varepsilon_\mathcal{U}$*,* ℓ_s *be given. Suppose that the fifteen conditions of Definition C.4, Definition C.22, and Definition C.23 hold, and the five of Definition C.21, of which (Exact simulation) and (Rare private aborts) are required at the OOD seeds of* $\mathcal{U}$ *alone. Let* $\mathcal{S}$ *assign to each index of* $\mathcal{J}$ *a set of instances with which it forms a pair of* $\mathcal{Y}$*. Then the whole-key argument formed from these data as in Definition C.20, with constant* pp*, has index privacy on* $\mathcal{S}$ *for batches of at most* 1 *(Definition 2.14), with error*

$$\zeta(q,N) \;+\; (K+1)\,\varepsilon_\mathcal{U},$$

where ζ *is the error of Theorem 6.6.*

Proof. Fix $A \in \mathcal{J}$, a valid environment and V^*.

Step 1: lazy sampling. Committer and prover answer their queries that contain a secret string with fresh uniform values, without consulting the table of oracle answers. Such an answer is entered into that table only where a proof string discloses the query: at opened leaves and at queries that are not leaves, never at an unopened leaf or after $\perp$. The distribution changes only if such a query coincides with another query. A query at the name o contains b_o uniform secret bits, and two attempts of one search differ in the nonce. A union bound over the q queries of V^* gives the first term of $\zeta_{\mathrm{hash}}(q,N)$ (Definition C.24), one over the pairs of queries of committer and prover at one name the second. Only (Uniform coins), (Consistent layout), (Plain seeds) and (Nonces readable) are used.

Step 2: a reduction to the table game. After Step 1 a table's leaf digests are fresh and independent of it. Root and OOD seed are therefore computed from sampled digests before the table is asked for, which is the order of the table experiment. Hence the experiment is obtained by a reduction from the table game (Definition C.13) with provers $\mathbf{P}$, one offline proof, one link proof for each copy and N_c online proofs on each copy. They differ if some copy is sampled more than N_c times, with probability at most $K \cdot \Pr[\mathrm{Bin}(N,1/K) > N_\mathrm{c}]$, or a seed lies outside $\mathcal{U}$, with probability at most $(K+1)\,\varepsilon_\mathcal{U}$ by (Good seeds).

Step 3: simulating the game. By the two budget conditions the game's numbers satisfy the inequality of Definition C.6. Challenges are derived from fresh answers; by (Consistent layout), the event that no search of a

compiled run fails and ch lies in a given set has at most that set's probability under uniform ch with proof-of-work bits 0. By (Rare private aborts) and (Exact simulation), a run differs from its simulation with probability at most $\zeta_*^{\mathrm{ab}} + \zeta_*^{\mathrm{sim}}$. By Lemma C.14, the game at A with $\mathbf{P}$ is replaced by the game at a fixed A_0 with Sim_*, at the sum of these terms over its $1 + K + N$ runs.

Step 4: the ideal experiment. The reduction around the simulated game depends on A_0 and the revealed instances alone; it is the simulator, and never answers one query in two ways. □

Proposition C.26 (The conditions that mention no prover hold). [machine-checked] *Let Z be admissible (Definition 3.8), let $(\mathsf{dg}, \mathsf{par})$ be arbitrary, $K \le 710{,}324$ and $N_{\mathrm{c}} \le 688{,}127$, and let the search limits be arbitrary. Then $\mathsf{FlashProve}_Z$ and the data of the compiled verifiers of V_Z at $(\mathsf{dg}, \mathsf{par})$ satisfy the conditions of Definition C.4, Definition C.22, and Definition C.23 with*

$$t = 44{,}040{,}191, \qquad (\nu_{\mathrm{off}}, \nu_{\mathrm{first}}, \nu_{\mathrm{copy}}, \nu_{\mathrm{on}}) = (62, 62, 62, 64), \qquad \varepsilon_{\mathcal{U}} = 0,$$

and with $\mathcal{U}$ the set of all OOD seeds.

Proof of Theorem 6.6. By Proposition C.26 its conditions hold with $\varepsilon_{\mathcal{U}} = 0$ and $\mathcal{U}$ the set of all seeds. The claim follows from Theorem C.25. □

Lemma C.27 (Bound on the error after compilation). [machine-checked] *Let Z be admissible and $(\mathsf{dg}, \mathsf{par})$ arbitrary. In Theorem 6.6 let $\zeta_*^{\mathrm{sim}} = \zeta_*^{\mathrm{ab}} = 0$, $N \le N_{\mathrm{c}}$ and $\ell_{\mathrm{s}} \ge 256$, and let the search limits of the three IOPs be at most $\mathsf{lim}_{\mathrm{key}}$, $\mathsf{lim}_{\mathrm{link}}$, $\mathsf{lim}_{\mathrm{req}}$. Then*

$$\zeta(q, N) \;\le\; \frac{q\, q_{\mathrm{hon}} + n_{\mathrm{nm}}\, q_{\mathrm{hon}} \big(2(K+1) + q_{\mathrm{hon}}\big)}{2^{256}},$$

where $q_{\mathrm{hon}} = (K+1)\, 2^{32} + 2^{37} + K\, 2^{25} + N\, 2^{39}$ and $n_{\mathrm{nm}} = 9 + \mathsf{lim}_{\mathrm{key}} + \mathsf{lim}_{\mathrm{link}} + \mathsf{lim}_{\mathrm{req}}$.

Proof. For $N \le N_{\mathrm{c}}$ the binomial term vanishes. At admissible Z every m_o is at most q_{hon}, $\mathsf{e}_o \le 2(K+1)$, $\mathsf{b}_o \ge 256$, and at most n_{nm} names have $\mathsf{m}_o \ne 0$; substitute in Definition C.24. □

Lemma C.28 (Constant-message provers satisfy the five conditions). [machine-checked] *For any Z and $(\mathsf{dg}, \mathsf{par})$, let $\mathbf{P}$ consist of constant-message provers (Definition 6.2) with arbitrary search limits. Let their coins be as in Definition C.19, with leaf number 0, with ℓ_{s} at most the length that the compiled verifier fixes for a leaf salt, and with all-zero public prefixes that complete this length. Let each Sim_* output the fixed messages together with the symbols that it is given. Then the five conditions of Definition C.21 hold with $\zeta_*^{\mathrm{sim}} = \zeta_*^{\mathrm{ab}} = 0$, for $\mathcal{J}$ the set of all matrices and every $\mathcal{Y}$.*

Proof of Corollary 6.7. Part 1. Where $\mathsf{Chk}(Z) = 1$ a leaf salt has 576 bits, so Lemma C.28 admits $\ell_{\mathrm{s}} = 320$. Theorem 6.6 applies with $\zeta_*^{\mathrm{sim}} = \zeta_*^{\mathrm{ab}} = 0$, and Lemma C.27 gives $\zeta \le \zeta_{\max}$. For $K = N = 1$ and $q \le 2^{64}$ the numerator is at most 2^{121}.

Part 2. A search fails only if $2^{40} + 1$ fresh answers miss a test of at most 32 bits, with probability at most $(1 - 2^{-32})^{2^{40}} \le 2^{-256}$. The three IOPs have at most 19, 6 and 46 rounds with proof of work. The committer runs before V^*, and its $19 + 6K$ searches fail with probability below the stated bound. The event that some $\pi_{\mathrm{req}} = \bot$ is a set of views, so its probability exceeds that in the ideal experiment by at most $\zeta_{\max}(q, N)$; there the Fiat–Shamir queries of a simulated run receive fresh answers. □

On the scope of Corollary 6.7:

- Beyond one copy and one request the bound is a formula, which no theorem evaluates at the default configuration (Section 6.3 has a value evaluated by hand).
- That the simulated view holds the roots, OOD answers and opened columns is read off the definitions, not proved.
- That each proof opens a column of a committed table is proved for the run of committer and prover alone and for one verifier that makes no query, not for every verifier.
- For two matrices of $\mathcal{J}$ and an environment valid at both, the real views are within $2\,\zeta(q, N)$, which is below 1 only if $\zeta(q, N) < 1/2$.

C.4 The link proof

Proof of Theorem 6.9, outline. Part 1. The abort set of P_{link} is empty by definition. Fix the input and ch. Prover and simulator have the same finite set of coins, vectors co of field elements; let $\mathsf{dis}_P(\mathsf{co})$ and $\mathsf{dis}_{\mathsf{Sim}}(\mathsf{co})$ be what each discloses at the coins co. There are maps sh, sh' on coins such that, for all co,

$$\mathsf{dis}_P(\mathsf{co}) = \mathsf{dis}_{\mathsf{Sim}}\big(\mathsf{co} + \mathsf{sh}(\mathsf{co})\big), \qquad \mathsf{sh}'\big(\mathsf{co} + \mathsf{sh}(\mathsf{co})\big) = \mathsf{sh}(\mathsf{co}).$$

For intuition, consider a masked value: a secret plus a uniform coin is distributed as the coin alone; here sh plays the role of the secret. By the second equation, two coins with the same image under $\mathsf{co} \mapsto \mathsf{co} + \mathsf{sh}(\mathsf{co})$ have the same sh, so they are equal. The map is injective, so a bijection of the finite set of coins, and by the first equation the two outputs are identically distributed under uniform coins. The equations use that no oracle part is opened at more positions than its rows have random coefficients, and that the claim left at the last sub-protocol is true in the honest run.

Part 2. The verifier accepts P_{link} whenever the OOD answers are those of the row polynomials, the tables are their evaluations, both encode the same plain entries, and the symbols have the format's form, as for two committed tables of one matrix. The other conditions of acceptance hold at the challenges that pass the proof-of-work checks. □

For its own constructions, [CFW26, version of February 25, 2026, Theorem 10.2] proves honest-verifier zero knowledge of the proximity test that $\mathsf{IOP}_Z^{\text{link}}$ runs with changes (Section 4.5), resting in substance on its Lemmas 6.4, 7.3 and 9.8, through its Theorems 9.10 and 7.1, by its composition theorem (its Theorem 4.5); Part 1 is proved for the whole of $\mathsf{IOP}_Z^{\text{link}}$ at once, by a bijection on coins.

Corollary C.29 (Committed tables and link proofs after compilation). [machine-checked] *Let* $(\mathsf{dg}, \mathsf{par}) = (0^\lambda, \emptyset)$, *and let the public prefixes of the leaf salts of* P_{link} *be zeros. Part 1 of Corollary 6.7, with its bound* 2^{-135}, *and for* $K = 1$ *part 2, hold with the honest prover* P_{link} *of Theorem 6.9 in place of the constant-message prover for* $\mathsf{IOP}_Z^{\text{link}}$.

Proof. In Lemma C.28 replace the prover for $\mathsf{IOP}_Z^{\text{link}}$ and its simulator by those of Theorem 6.9, whose part 1 gives $\zeta_{\text{link}}^{\text{sim}} = \zeta_{\text{link}}^{\text{ab}} = 0$. The other three conditions hold for a suitable choice of the compiled run's coins. Lemma C.27 gives the same bound, and part 2 uses only that no prover aborts. □

Remark C.30 (Reach of the corollary). Corollary C.29 holds at the one fixed value $\mathsf{dg} = 0^\lambda$ in place of the digest of the configuration, at which V_Z is not proved ever to be called. The public prefixes are zeros by choice. The other two provers send constant messages: nothing that these two would compute from the weights is covered. At these values and every admissible Z with $\mathsf{Chk}(Z) = 1$, the interactive verifier of $\mathsf{IOP}_Z^{\text{link}}$, not the compiled one, accepts P_{link} at the challenges that the oracle answers give, wherever $\pi_{\text{link}} \neq \bot$ (machine-checked).

D Sub-protocols, parameters and round errors

Definition D.1 (The default configuration, encoding). The default configuration Z_0 is the triple of byte strings that the Lean development writes out as a term. Its strings have the lengths 2629, 224975 and 2784 and the SHA-256 digests

```
ce9e126bb3ba7179319a31981a3fc36ca505d738348971f379dfc795f94a3fae
a99b964bbd3648d6e3b4007fc9370b5c70c7e9668848daa039820bc9e03e8f80
1c3191164c5113fd04d5108a469d7b3d17f4ccb3256e8a676d15bdc94435b919
```

The configuration $Z_0[\mu, \mathsf{pol}]$ (Definition 3.7), with $\mathsf{pol} = (C, \theta, \mathit{cap})$, consists of:

1. the parameter file of Z_0 with the values of ϵ_1, ϵ_2, ϵ_F, M_F, K_F, C, θ, *cap* replaced, in decimal without leading zeros, each ϵ in two lines as $\epsilon \bmod 2^{32}$ and $\lfloor \epsilon / 2^{32} \rfloor$;
2. the layout file of Z_0;
3. the scale file of 13 little-endian 32-bit words for each layer l,

$$k_q,\ k_k,\ k_v,\ k_o,\ k_g,\ k_u,\ k_d,\ \mathsf{M}_1,\ \mathsf{K}_1,\ \mathsf{M}_2,\ \mathsf{K}_2,\ k_s,\ \mathsf{w}_l,$$

where $\mathsf{w}_l = 0$ if $l \equiv 3 \pmod 4$ and $\mathsf{w}_l = 1$ otherwise, and, after the last layer's words, k_r, k_g', k_u' for each odd layer, in increasing order of the layers.

In μ_0 the first twelve words are $29, 29, 17, 17, 16, 20, 19, 44, 30, 44, 30, 8$ in every layer, $(k_r, k_g', k_u') = (17, 16, 20)$, $\epsilon_1 = \epsilon_2 = \epsilon_F$ and $(\mathsf{M}_F, \mathsf{K}_F) = (44, 30)$.

Remark D.2 (Written copy). The release ships Z_0 and Z_1 with its two stored proofs for the integer model at synthetic weights. No theorem relates the shipped files to the Lean terms; the release contains a program that compares them. The digests in Definition D.1 are the terms', computed outside Lean. The value of $\epsilon_1 = \epsilon_2 = \epsilon_F$ in μ_0 stands in the shipped `examples/full-size/config/policy.txt`.

Construction D.3 (The integer model). Put

$$(d_{\text{model}}, n_h, n_g, d_h, f, f_e, N_E) = (5120,\ 40,\ 8,\ 128,\ 16384,\ 8192,\ 128).$$

The weights M (Definition 3.1) are the following integer matrices: an embedding matrix ($|\mathcal{V}| \times d_{\text{model}}$), an output matrix ($d_{\text{model}} \times |\mathcal{V}|$) and, in each layer,

$$W_q\colon d_{\text{model}} \times n_h d_h, \qquad W_k, W_v\colon d_{\text{model}} \times n_g d_h, \qquad W_o\colon n_h d_h \times d_{\text{model}},$$
$$W_g, W_u\colon d_{\text{model}} \times d_{\text{hid}}, \qquad W_d\colon d_{\text{hid}} \times d_{\text{model}},$$

with $d_{\text{hid}} = f$ in an even layer; an odd layer has $d_{\text{hid}} = f_e$, a router matrix ($d_{\text{model}} \times N_E$) and N_E further triples of the shapes of W_g, W_u, W_d, the routed experts. Entries of the embedding matrix lie in $[-2155905152,\ 2139062143]$, all others in $[-2^{12}, 2^{12})$.

Let μ be valid (Definition 3.6) and $\bar{x}$ a context (Definition 3.1). At each token position of $\bar{x}$ the model starts from the token's row of the embedding matrix; each layer adds the output of a causal attention block and then of a gated feed-forward block, each applied to a normalized input; an odd layer evaluates W_g, W_u, W_d and the routed expert that the router ranks first. The logits $\text{logit}_M(\bar{x})$ are the normalized vector at the last token position times the output matrix. All steps are exact integer operations. Eight kinds of intermediate value have prescribed intervals; the model is defined on $\bar{x}$ if these hold at every token position.

Remark D.4 (Model specification). Construction D.3 is an outline of the Lean term that defines the integer model (Section 3.1). The release's `docs/model.md` gives every step, the eight intervals and the five constant lookup tables; cosine and sine share one of the four digests of tables in the parameter file. In Lean the model holds its five tables as constants: no theorem describes their content, and none ties them to these digests. On synthetic weights at small sizes the Lean function agrees with the release's reference program in every value that was compared, over Lean files and programs that the integer-model package's pages name (`integrations/vllm/integer_model/tests/lean_vectors/README.md`); no theorem relates the two.

Remark D.5 (No real weights shown). On trained weights (Remark 3.3, Trained weights):

1. *Counted outside Lean.* A reference program like the release's, not the released one, under a parameter file that the release does not contain, found no range condition failing on 544 of 544 windows, cut from 96 prompts (what the count does not show: `lean/LIMITS.md`, "At real weights: a count, no theorem"), at a public model's 4-bit weights.
2. *Run.* The C++ programs have proved requests on these weights; the timed runs of Section 7.2 do so, at a configuration fitted to the weights (Section 7.1). The release ships the fitting tools and no fitted configuration. A fitted configuration of timed exact-match runs on these weights passes the scope test Adm (Definition 3.8) by one evaluation (whether it is these runs' own files, byte for byte, is not shown), with Lean sources as in Section 7.1 (no theorem); every security line of its parameter file is the default's. The parameter checks (Definition 4.14) were not evaluated at it; where they fail, V_Z rejects every input. The configuration of the clipped-gap rows was not put to the scope test.
3. *Not tried.* The executable form of V_Z has been run on proofs for synthetic weights (Section 7.3), not on proofs for a trained model's weights.

The sources of the sub-protocols are in Table 11; the Proximity test's Fold, Code switch and Base case are, with changes (Section 4.5), Constructions 6.3, 9.7 and 7.2 of [CFW26, version of February 25, 2026]. Send has the masking of the answers from its Section 9.2, here with the pads in the message itself (in its Construction 9.7, in a mask oracle); the main course of Masked sumcheck, with committed masks whose values are not revealed but join the output claim, has the form of steps 1 to 4 of its Construction 6.3 and, for a polynomial summand, of steps 4 to 6 of its Construction 11.4. Steps 2 and 4 of Construction D.7 have the form of steps 1 to 4 of its Construction 9.7, which carries over to zero-knowledge encodings a known template for code switching, here with the table's random coefficients in the new message.

Construction D.6 (Sub-protocols). This construction gives the course of each sub-protocol (Remark D.10, Given in outline). An *oracle message* is a list of rows, each sent as a masked encoding (Definition 4.2) under a code that Z fixes; its *mask rows* are uniform. A *group* is a set of these under one code. A *linear constraint* is an equation $\langle \mathsf{co}, f \rangle = \mathsf{tg}$ on the entries f of a group, with co, tg computed by V. A challenge is a uniform tuple of blocks of λ bits, from which fixed maps compute what V *samples* (Lemma A.20).

Send. P sends an oracle message f; V samples OOD seeds $(z_i, \beta_i)_{i<s}$; P sends answers a_i. The constraints $\langle \mathrm{eq}(z_i, \cdot), f\rangle + \beta_i \,\mathsf{pad}_i = a_i$ join the group of f, with pad_i as in Definition 4.3.

Batching. V samples $\varkappa \in \mathbb{E}$. Claims or constraints $\mathsf{cl}_0, \mathsf{cl}_1, \ldots$ are replaced by $\sum_j \varkappa^j \,\mathsf{cl}_j$.

Masked sumcheck, for a claim that F has the sum S over the Boolean cube. P sends S', the sum of a mask polynomial F' with committed coefficients; V samples $\varkappa$; the running claim is $S' + \varkappa S$. For each variable, P sends the round polynomial of $F' + \varkappa F$ without the coefficient that the running claim determines, and V samples an element. The output is a claim on $F' + \varkappa F$ at the sampled point. Those of steps (d), (f) of Construction D.9, and all but one of a Lookup, end with two further rounds: one more variable, with a nonzero sample; then masked end values.

Fold, for one constraint on a group. A masked sumcheck with sampled point γ replaces it by one constraint on $\sum_r \mathrm{eq}(\gamma, r)\,\mathrm{row}_r$ and on mask rows.

Code switch. Send f' under another code, said to hold the folded row, with a nonce beside the answers; V samples positions and opens the old group there. Each opened column, folded at γ, gives a constraint on f'; these, the fold's and the answers' constraints form a new group.

Base case, for one constraint $\langle \mathsf{co}, f\rangle = \mathsf{tg}$ on rows f sent with masking rows f_*. P sends tg_*; V samples $\varkappa$; P sends $f_* + \varkappa f$ with its random coefficients, and a nonce; V samples positions, and checks $\langle \mathsf{co}, f_* + \varkappa f\rangle = \mathsf{tg}_* + \varkappa\,\mathsf{tg}$ and the encoding against the opened symbols.

Proximity test, on a group. Batching; for each stage but the last, Fold and Code switch, whose second challenge also batches the new group's constraints; Base case.

Lookup, for lookup keys ky_i and a table $(\mathsf{tb}_j)_j$. P has sent, in bits, the multiplicity mu_j of each tb_j. V samples ϖ. The claim $\prod_i(\varpi - \mathsf{ky}_i) = \prod_j(\varpi - \mathsf{tb}_j)^{\mathsf{mu}_j}$, with a public factor for padding, is reduced along the two trees of products, by masked sumchecks, to claims on the lookup keys and the bits; one more shows that the bits are bits.

Range test, for cells Y already sent. Send a vector U, said to hold the sums of cells that the segment's claims name. V samples Θ (Table 2). Send the digits of $Y\Theta$ with their multiplicities; a Lookup places them in their ranges. Claims relate the digits to Y and Θ, and U to Y.

Construction D.7 (Test on a committed table). *Input.* An external oracle string $\mathbf{T}^*$ with a sample $\sigma = (\varsigma, \mathbf{a})$; numbers ν and κ; and, for each request of a batch, a claim on the table (Construction 4.13) with $h \in \mathbb{E}^d$, $\psi \in \mathbb{E}^n$, $b \in \mathbb{E}$, and an oracle message $w \in \mathbb{E}^{2^n}$, said to be $h^\top A$, with the constraint $\langle \mathrm{eq}(\psi, \cdot), w\rangle + \eta = b$. Brackets hold what the honest prover sends.

Shared part, run once for the batch.

1. V samples $\psi^* = (\psi_0^*, \gamma) \in \mathbb{E}^{n_0} \times \mathbb{E}^v$.
2. Send u $\big[\, u = \sum_r \mathrm{eq}(\gamma, r)\,(f_r, \tau_r)$, with f_r, τ_r of Definition 4.2 $\big]$, with a mask message.
3. For $i < s$ the following constraint is added to the group of u:

$$\sum_{j<\ell_0} \mathrm{eq}(z_i, j)\, u[j] + \beta_i \sum_{k<3} \vartheta^k u[\ell_0 + 3i + k] = \sum_r \mathrm{eq}(\gamma, r)\, a_{r,i}.$$

4. V samples ν positions, with κ bits of proof of work. $\mathbf{T}^*$ is opened there. For each position ϕ the constraint that the encoding of u has the symbol $\sum_r \mathrm{eq}(\gamma, r)\,\mathbf{T}^*[\phi]_r$ at ϕ is added.
5. Send ω $\big[\, \omega = A\,\mathrm{eq}(\psi^*, \cdot) \in \mathbb{E}^d \,\big]$, with a mask message.
6. V samples $\chi \in \mathbb{E}^{\lceil \log_2 d\rceil}$.
7. P sends one masked value. With it the equation

$$\sum_{\iota<d} \mathrm{eq}(\chi, \iota)\,\omega[\iota] = \sum_{\iota<d,\ j<2^{n_0}} \mathrm{eq}(\chi, \iota)\,\mathrm{eq}(\psi_0^*, j)\, u[\,j + 2^{n_0}\iota\,]$$

   is split into one constraint on the group of u and one on the group of ω.
8. Proximity test on the group of u, then on the group of ω (Construction D.6).

Last part, run for each request.

9. ω joins the group of w, and the constraint $\langle \mathrm{eq}(\psi^*, \cdot), w\rangle = \langle h, \omega\rangle$ is added.
10. Proximity test on that group.

Proposition D.8 (Two rounds at the default configuration). [machine-checked] *Let the sizes of a committed table, e, ν_{on}, κ_{bat} and R be those of Z_0 (Table 2); in particular $v = 11$, $n = 18$, $\nu_{\text{on}} = R = 64$ and $\kappa_{\text{bat}} = 28$. Let $d_{\min}$, J and ls be as in Lemma A.24, with $e' = e$.*

1. *$|D|\,(|D| - d_{\min}) < (|D| - e)^2$ and $e < d_{\min}$.*
2. *$\lfloor J \rfloor \cdot 2^{132} \le |\mathbb{E}|$.*
3. *At a word without a collision, at most $(11\,\lfloor J \rfloor + 286)\,|\mathbb{E}|^{10}$ points γ are bad.*
4. *The bound of Lemma A.20 is at most 2^{-128} for step 1 of Construction D.7 with at most 64 requests.*
5. *It is at most 2^{-128} for step 4, with the count $(|D| - e - 1)^{64}$ and 28 bits of proof of work. With 27 bits it exceeds 2^{-128}.*

Construction D.9 (The rounds of the three IOPs). The sub-protocols are those of Construction D.6 and Construction D.7. Brackets hold what the honest prover sends.

Opening rounds. In $\mathsf{IOP}_Z^{\text{key}}$ and $\mathsf{IOP}_Z^{\text{req}}$ each part opens with a round whose message is a plain part alone. The first part opens with OOD answers and a number, the shared part with OOD answers and a free value, the last part with a free value. V rejects unless both lists of answers are the IOP instance's and, in $\mathsf{IOP}_Z^{\text{key}}$, the number is 0; in $\mathsf{IOP}_Z^{\text{req}}$, step 9 of Construction 4.15 reads it. The shared part closes with an empty round.

$\mathsf{IOP}_Z^{\text{key}}$.

1. *First part.* Send a mask message; Send the multiplicities. Two Lookups: for the entries of A, and for the sums $b_0 + 2^8 b_1$ of Definition 4.1. Proximity test on these messages' group. Batching and a masked sumcheck over the n variables of the matrix columns reduce the Lookups' claims to one claim (h, ψ, η, b). Send w.
2. *Shared and last part.* Construction D.7 with $\nu = \nu_{\text{off}}$, $\kappa = \kappa_{\text{off}}$ and a batch of one.

$\mathsf{IOP}_Z^{\text{link}}$.

1. V samples $\gamma \in \mathbb{E}^v$.
2. Send one oracle message with u_0, u_j [each table's fold at γ, as u in Construction D.7] and a mask.
3. For each table and each $i < s$, the constraint of step 3 of Construction D.7.
4. V samples $\nu_{\text{first}} + \nu_{\text{copy}}$ positions, with κ_{off} bits of proof of work. $\mathbf{T}_0^*$ is opened at the first ν_{first}, $\mathbf{T}_j^*$ at the others, with the constraints of step 4 of Construction D.7.
5. V samples $\chi \in \mathbb{E}^{\lceil \log_2 \ell \rceil}$. The constraint $\sum_{k < \ell_0} \mathrm{eq}(\chi, k)\,(u_0[k] - u_j[k]) = 0$ is added.
6. Proximity test on the one group.

$\mathsf{IOP}_Z^{\text{req}}$.

1. *First part.* For each segment $g < G$ (Definition 4.8) in turn:
   (a) in the head alone, P first sends a token in a plain part, and V stores it;
   (b) for $g = 0$, Send the segment's cells and a mask message for the claims on the table; for $g > 0$, Code switch, with κ_{req} bits of proof of work, from the group of segment $g - 1$ to a message that also holds the cells of segment g and the lower segments' cells that higher segments read;
   (c) Lookup for the segment's lookups;
   (d) the segment's own rounds: sampled points, Batching and masked sumchecks reduce its identities to claims on cells and on the table;
   (e) Range test on the cells that the segment reads;
   (f) in the head alone, Batching and a masked sumcheck on its claim on the table;
   (g) Batching of the segment's claims into one constraint on its group; Batching of the group's constraints; Fold.

   Then Code switch and Proximity test on the last group; Batching and a masked sumcheck, as in $\mathsf{IOP}_Z^{\text{key}}$, leave one claim (h, ψ, η, b); Send w.
2. *Shared and last part.* Construction D.7 with $\nu = \nu_{\text{on}}$, with $\kappa = \kappa_{\text{bat}}$ in the shared part and κ_{req} in the last.
3. V rejects unless the token of step (a) is the y^+ of ξ.
4. After every sub-protocol of the first part other than step (a), V rejects if its stored value of that token has changed.

Each verifier also rejects if a group is left untested.

Remark D.10 (Given in outline). This paper gives V_Z in outline. Its Lean code defines, and the paper does not print:

- the placement rule and wt (Definition 4.1);
- the arithmetization (Definition 4.8), with the rounds of step (d) of Construction D.9; Lemma 4.9 states its meaning; that the lemma applies at an admissible configuration joins Lean theorems by hand, a step that Lean's proof of Theorem 5.1 does not use, since that the three byte strings of Z decode without error, which admissibility implies, is one of that theorem's hypotheses in Lean;
- the closed list of the parameter checks (Definition 4.14), which are computed with a constant in the place of dg_Z; among them are a second derivation of the rounds, from the sizes alone, with one decoding radius for each shape of oracle message, and, in attention segments, a count of each round's bad challenges;
- the sub-protocols in detail (Construction D.6);
- the maps code, the strings of the queries and the bijection (Construction D.11).

The release's format specification states what is hashed and sent.

Construction D.11 (Oracles, seeds, and the compiled verifier of one IOP). *Oracles.* The algorithms of Π_Z query separate oracles: for leaves and inner nodes of Merkle trees over oracle strings; for those of the *batch tree*, with one leaf for each request of a batch; and, for each IOP, a Fiat–Shamir oracle fs and an oracle g. A bijection between bit strings and triples (oracle, query, number of an answer block) implements them by H. The public readings determine it.

Public readings. The pair $(\mathsf{dg}_Z, \mathsf{par})$, where par consists of the values of H at up to seven strings, formed from Z, dg_Z and H', that state the parameters of the three IOPs. Ext_Z chooses public readings whose digest is dg, and translates every query of tr by their bijection.

Unpacking. Step 2 of Construction 4.15 unpacks the byte strings of π at dg_Z, by a query of its own; step 3 queries $\mathsf{code}(Z)$ again. If π cannot be unpacked, V_Z continues with a fixed tuple whose fill is not a list of tokens of $\mathcal{V}$; step 4 then rejects. A byte string that is unpacked but gives no proof string of its IOP's shape gives $\perp$, and step 6 rejects.

Commitment to an oracle string. A Merkle root; the leaf at a position hashes the symbol there and a *leaf salt*.

Seed of a root. $\mathsf{Seed}(\mathsf{rt})$ is read off the second of two answers of one Fiat–Shamir oracle: the first query holds rt, dg_Z and the table's sizes, the second the first answer's chaining value.

Proof string. A *chain salt*; the roots of the prover's oracle parts; its plain parts (Definition 2.3); openings; the batch tree's root, a leaf number, a path and a leaf salt.

Compiled verifier, given a list cms of digests, an IOP instance and a proof string. For $\mathsf{IOP}_Z^{\mathrm{req}}$ the list holds the copy's root twice.

1. Recompute the challenges. Every round's query to fs holds the round's number, roots and plain part, nonce included. The first also holds cms, the chain salt and, for $\mathsf{IOP}_Z^{\mathrm{req}}$, the *early part* of ξ, which is ξ with y^+ replaced by 0; each further one the chaining value of the preceding answer. An answer gives the challenge, bits of proof of work included, and a chaining value.
2. At the first shared round, g maps the request's chaining value and own component to its *batch value*, and the query to fs holds cms, the batch tree's root and the shared component. The last part continues from the shared part's chaining value and the batch value.
3. Reject unless the salts have their fixed lengths and the openings are exactly those that the IOP's verifier reads.
4. Verify every opening against its root and, where the IOP has shared rounds, the batch value, with its leaf salt, against the batch tree's root.
5. Decide as the IOP's verifier does on the challenges, the plain parts and the opened symbols.

Construction D.12 (Committer and prover of Π_Z). Rnd_Z outputs the masks and salts. $\mathsf{Commit}_Z^H(M, \mathsf{r})$ forms $\mathbf{T}_0, \ldots, \mathbf{T}_K$ (Definition 4.6) for a digit matrix determined by M, computes rt_j, $\varsigma_j = \mathsf{Seed}(\mathsf{rt}_j)$ and $\mathbf{a}_j$, makes π_{key} and each π_{link} as the honest provers of Construction D.9 do at the recomputed challenges, and returns $H(\mathsf{code}(\mathsf{k}, \mathsf{dg}_Z))$. $P_Z^H(M, \mathsf{r}, x, y)$ makes π_{req} likewise, with empty fill; for a batch it chooses a copy within the opening budget (Definition C.6) and places the requests at distinct random leaves of the batch tree.

Remark D.13 (How the two hash functions differ). O^* (Assumption 4.10) and H' (Assumption 4.11) both iterate the compression function of BLAKE2b, which takes a counter of 128 bits. O^* evaluates BLAKE2b as specified, on labelled strings of fewer than 2^{32} bytes; the upper 64 bits of the counter are 0 in each of its calls. H' starts from the state of BLAKE2b with 64-byte output and sets these 64 bits to $\mathsf{num} + 2^{32}k$ in every call, where k is the number of 32-bit words of its input and num is one of the four numbers in use, none of them 0. The two functions hence differ in that half of the counter in every call. That this keeps them apart is part of Assumption 4.11: no formal proof states it, and the published analysis of BLAKE2b does not cover H'.

V_Z evaluates H' on public tables and on parts of the statement, x and y among them. Its digest of the encoded request is part of the first string that an online proof hashes (Construction D.11). The digest is redundant there: the

same string holds the window, with x and y, and the two token positions; the digest is compared with nothing; and the coding of that string is injective by the encoded request alone, whatever the values of H' (proof of Lemma A.17). Hence a collision of H' merges no two queries.

Remark D.14 (How the parameters are obtained). In Table 2 the parameter file states the sizes of the tables and the bits of proof of work. Every other value is calculated from that file and taken from no run. A verifier program calculates these values without checking a proof; δ and the bound on K are calculated from its values, and no program outputs them. Rates are rounded up, proximity parameters down. The reduced configurations of Section 7 are synthetic models far smaller than the default one.

- *Range test.* V samples a matrix Θ with 128 columns whose entries are 0 with probability 1/2 and ± 1 with probability 1/4 each. P sends each entry of the product as two digits; an entry that two digits can express lies in $[-2^{17}, 2^{17})$.
- *Copies and batches.* A copy serves at most 688,127 batches of up to 64 requests, and a commitment has at most 710,324 copies; both numbers are calculated. The committer fixes K when it commits. Every started batch counts towards N_j, whatever its size. These 688,127 batches are about 4.81 years (1,759 days) of batches of 24 requests back to back on one GPU that never pauses, calculated from the measured time of such a batch (Section 7), and less with smaller ones. The programs choose a batch's copy at random (from code review).

E Sources and credit

E.1 The sources of each part

Table 11 lists each part of the protocol with the section or numbered item that describes it, its sources and, for some parts, a pointer to the technical report, which proves it, cites it or takes it as given (its written proofs are not verified, those for soundness and for zero knowledge are not settled, and nearly every reading of them has found errors: Limitation G.7; `docs/limits.md`). Where a place in [CFW26] is named, it is that of the version of February 25, 2026; citations that name none refer to that work as its authors gave it to this project [CFWpc].

The report is written for other programs (Section 7.1). Among other differences, those programs depart from the report's protocol at three points, which lie in the compilation, and none of the report's written proofs covers them. How the report's theorems relate to the released programs: `docs/limits.md`.

Table 11: The parts of the protocol and their sources.

Part	Section; statement	Source
The definitions of the primitive: a commitment to the weights, then proofs of requests; completeness, index-binding knowledge soundness, index privacy	Section 2.4; [TP]	adapted from [CFWpc]; the report's experiment hands the extractor the OOD seed, unlike Definition 2.7, whose extractor is given the commitment and the committer's trace alone
The plan: an offline phase, a phase for each batch, copies with their link proofs	Definition 4.6 and Definition 4.7	adapted from [CFWpc]
The link proof	Definition 4.6	after [CFWpc]; [CFW26]
The weights as a matrix of digits; one test of the committed table, then tests of its folded rows	Definition 4.1 and Construction D.7	[CFWpc]; [GLSTW23, Section 4.1], whose scheme with a Reed–Solomon code is, by its own statement, implicit in Ligero [AHIV17]
Committed tables under a code that hides	Definition 4.2	[CFWpc]; [CY24, Section 32.5]; [CFW26]; [ACY22, Section 1.2], for data encoded apart from the instance and committed once for many statements, without hiding
The OOD sample of each committed table, taken when the table is committed, with distances beyond unique decoding in the tests of a committed table	Definition 4.3 and Definition 4.5	[CFWpc]; the mask of an answer: [CFW26, version of February 25, 2026, Section 9.2]; that only different messages collide: its Section 9.1

Part	Section; statement	Source
The step from two words to a fold of several rows	Lemma A.24; a lemma and a corollary of [TP]	by the report's list of statements: the lemma after [AER24] and [DG24, Theorem 3.6], in part the report's own; the corollary the report's own, with [BCGM25, Corollary 8.6 and Theorem 9.2] and [CFW26] as its sources; see also [BCGM25, Lemma 4.4]: a tensor product of two generators with MCA has MCA, with the sum of their errors
The repeated mask codes	a lemma and a corollary of [TP]	repeating a mask's code is in [CFW26], where the verifier enforces the repetition; the report departs from that [TP]: a mask row is sent copied, and the errors are those of the repeated code. By the report's list of statements: the lemma and the corollary are stated and proved in the report; they are not verified here, and no machine-checked statement of this paper takes them as a premise; the list names no source for the lemma, and for the corollary it names [CFW26], for the repetition of a mask's code alone; the corollary holds given MCA strictly below the Johnson radius (Lemma 5.13, machine-checked), by way of the lemma on interleaved rows [TP]; on that lemma see the row "No factor for the number of interleaved rows" below
One reading of the committed table per batch	Construction D.7	after [GLSTW23, Section 4.1] and [BFRW25, Section 2.1], the second of which, by its own statement, follows Ligero [AHIV17]; [CFW26]; [CFWpc]
The count of revealed positions: query bound and opening budget	Definition 4.2 and Definition C.6	[CFWpc]
The zero-knowledge proximity tests: the folding sumcheck, code switching, the final test	Construction D.6	[CFW26] with three changes [TP], of which this paper describes one
The zero-knowledge sumcheck for a polynomial summand	Construction D.6	adapted from a theorem of [CFW26]; the sumcheck [LFKN92]; its round polynomials [Gru24]; its hiding polynomial [CFW26]; that paper's version of February 25, 2026 (its Section 2.5) likens its own small masks to the small mask for each round in [XZZPS19], and finds a full mask, a random polynomial as in [CFS17], too costly the two lemmas on the hiding polynomial [TP]: both are adapted from [CFW26]
The lookup's two products (grand products)	Construction D.6	[GKR08]; [Tha13, Section 5.3.1]
Lookups into several tables combined as one	Construction D.6	adapted from [BCGJM18, Appendix B.5]; several tables in one: [GW20, Sections 4 and 4.1]; [BCGJM18, Appendix B.6]; the report states the lookup's interface as a sub-protocol of its own; the form in which it states sub-protocols is after [CFWpc]
Range checks	Construction D.6; [TP]	adapted from [LNS21, Lemma 2.5]; [GHL22, Section 3.4]
Congruences that hold over the integers	[TP]	[CFWpc]; [GHL22, Section 3]
The sumcheck for a matrix product, on committed weights	Construction D.9 and Construction D.7	[Tha13]; [LXZ21, Section 4.3]; [SLZ24]
Proving the model segment by segment	Definition 4.8 and Construction D.9	no source named; earlier systems that split a model into parts: Appendix E.2

Part	Section; statement	Source
Proximity below the Johnson radius	Lemma 5.13	in the report cited and taken as given: [BCHKS25, Theorem 4.6]; see also [Hab25]; in Lean machine-checked in the typed form of Lemma 5.13 (Appendix E.4)
Out-of-domain samples	Construction D.6; a definition, a lemma and a theorem of [TP]	the technique, from [BGKS20], as in [ACFY24b, Section 4.4], whose results and proofs are, by its own statement, adapted from [ACFY24a]; by the report's list of statements: the lemma is adapted from [CFW26]; the definition is after a suggestion [CFWpc], with its statements and proofs the report's own and parts from [CFW26] and [CFWpc]; the theorem is after [CFW26], in part the report's own following a suggestion [CFWpc], the verifier takes such a sample after each message that carries data, as in [BGKS20], [ACFY24a, Section 4.3], [ACFY24b, Section 4.4] and [CFW26]
No factor for the number of interleaved rows in the MCA error of a line	Lemma A.24	a case of a statement in the abstract of [Jo26], which has no such factor for generators with no more seeds than the field has elements; of that preprint only the abstract has been read; the case used here is machine-checked within Lemma A.24; see also [CFW26, Lemma 3.15]: for additive codes, interleaving costs at most a factor of the number of rows
Reed–Solomon encodings that hide: the lemma	Remark 4.4 and Lemma C.8; a lemma of [TP]	adapted from lemmas of [CFW26]
The compilation: Merkle trees, the Fiat–Shamir transformation, proof of work	Definition 2.4	the BCS transformation [BCS16] with proof of work, adapted from [CY24, Section 25.1] and [Sta21, Section 6.3]; the compiled argument, adapted from [CFWpc] and the COS transformation [COS20, Section 10], [CY24, Section 32.7]; [CFW26]; the lemma on proof of work, adapted from [Sta21, Theorem 6]; the reduction, [CY24, Sections 26.1 and 26.2]; [BCFW25, Theorem B.4]; proof of work on the rounds that sample positions, [CFWpc], with published uses in [Sta21, Section 3.11.3], [ACFY24a, Section 6.2] and [ACFY24b, Section 6.2]; the outlines of the report's proofs of index-binding knowledge soundness and of index privacy, and the first term of its bound on index privacy, after [CFWpc] committed tables, each committed once, are compiled as external oracles with the COS transformation, following [CFWpc] how proof of work counts: two lemmas of [TP], one of them adapted from [Sta21, Theorem 6]
A single bound on the error of every round	Proposition 5.14 and Definition 4.14	follows a suggestion [CFWpc]; as in [ACFY24b, Section 6] and [ACFY24a]
The field	Table 2	the choice of the field takes up a suggestion [CFWpc]
Logit gap, cost, acceptance policy; tokens that the provider can place	Definition 3.4	as in the technical report's appendix on acceptance policies [TP]; on Token-DiFR: Appendix E.2

E.2 The forward pass, other systems, batches and policies

The forward pass. Softmax, attention, and rounding by a lookup of the remainder follow [SLZ24, QSL25, CWS24].

zkComposer [SGO26] cuts the computation into parts, by layers or by input positions, and proves each part on its own. What passes from one part to the next is committed to, and that ties the parts' proofs together. They can be made in parallel or, to save memory, one at a time. DeepProve [GHL26, Section 4.5] likewise partitions the model into chunks of consecutive layers, for several machines to prove. NanoZK [Wan26] proves each layer apart and links the proofs by a chain of hash commitments.

Our segments are not proofs of their own: one proof runs through all of them in order, and a closed segment's folded row and the rows that a later segment still needs are sent on, encoded anew, with the claims on them (Construction D.9).

Batches. Our batches rest on the observation of SLP [LY26] that the weights, not the tokens, dominate the cost. In SLP, one proof covers many requests; here, each request gets a separate proof. A Merkle tree over several verifiers' transcripts, whose root serves as the challenge for all of them, appears in [YLFKM21]; [GGQZ25, Section 4.2] uses one for proofs of neural networks and credits [ZXHSZ22].

Many proofs. The hiding theorems of [SLZ24, GGQZ25] cover one prompt or one query. For many proofs under one commitment, [GGQZ25, Section 4.3], whose own aim is to hide the architecture too, notes that a one-time mask fails, and masks the committed polynomial afresh for each proof through the commitment's homomorphism.

As we understand it, a design like ours, in which each proof reveals positions of a table committed once, needs an opening budget; a homomorphic commitment can be masked afresh for each proof and needs none. A hash-based commitment needs no number-theoretic assumption, where zkLLM's Hyrax commitment [WTSTW18] rests on the discrete logarithm problem; neither needs a trusted setup.

Policies. At temperature 0, the logit margin of Token-DiFR [DiFR, Section 4.2] equals our logit gap up to one unit of the integer logits, which our rule for ties can add (Definition 3.4). The equality is one of formulas on the same logits: Token-DiFR's are a reference model's floating-point logits, and ours are the committed integer model's integer logits.

Reported numbers. Table 8 gives four earlier systems' results as their authors report them.

E.3 Standard terms

Table 12 sets terms of this paper beside those of the literature. It concerns the words alone: the sources of the definitions are those of Table 11.

Table 12: Terms of this paper and of the literature.

This paper	The cited work	Source
commit-and-prove argument	commit-and-prove SNARK	[CFQ19, Definition 3.1]
a commitment to the weights, then proofs of outputs	function-hiding functional commitment, a close primitive	[BNO21, Definition 5]
offline phase, online phase	the same, of a preprocessing argument	[COS20, Sections 1.1 and 9]
index privacy	the same, of an algebraic holographic proof	[BNO21, Definition 6]
trace	query-answer trace	[CGSY23, Definition 3.3]
round-by-round soundness, doomed	round-by-round soundness, state function, doomed set	[CCHLRR18, Definition 5.3]; [Hol19, Definition 2.5]; [CFW26, version of February 25, 2026, Definitions 3.6 and 3.7]
leakage function	the same, of structured encryption	[CK11, Section 4.2]

E.4 The Lean library

Codes. The proof of Theorem 5.1 calls theorems on Reed–Solomon codes, above all mutual correlated agreement below the Johnson radius (Lemma 5.13, the statement that Lean's theorems use in the place of [BCHKS25, Theorem 4.6]). As typed in the library they are proved there, not assumed, from files adapted from ArkLib and Mathlib [ArkLib, Mathlib] (`lean/ThirdParty/`).

A public Lean library, ArkLib [ArkLib] (Apache-2.0), has a theorem of this kind: mutual correlated agreement of Reed–Solomon codes on affine lines, for every radius below the Johnson radius (the probability form, which we have not checked); the source's own form of the bound is not proved there. The proof runs through adapted proofs of two theorems of ArkLib's: a counting form of such a statement and a bound on its count. Files of our own (`lean/Cb/Bridge/`) prove that, under the statement's premises, the estimate in the second equals the bound in the statement, and derive the

statement from the two. Two of these files (`CodeListRSFromLines.lean`, `CodeListRSProved.lean`) write out ArkLib's estimate and restate what its two theorems give together. Their heads say that the two cannot count as independent of one more file of ArkLib's, `Johnson/Probability.lean`, the file of the probability form, which the next paragraph names.

The commit of [ArkLib] is cited as named in the copy that we used; we have not looked that commit up on ArkLib's public repository. ArkLib's declarations in question are ReedSolomon.exists_johnson_line_exactCorrelatedPair and ReedSolomon.HiddenDerivative.johnsonExceptionCount_le_comparisonEstimate, and ReedSolomon.mcaError_affineLine_johnson_le (in ArkLib/Data/CodingTheory/ReedSolomon/MutualCorrelatedAgreement/Johnson/Probability.lean), the probability form, which we have not checked; two files of our own do not count as written independently of that file (the paragraph above). Files adapted from it come with this repository's Lean library (lean/ThirdParty/). Comments in six of these files credit further sources, among them the formalization kz99/rs-ld-mca ("adapted, with permission", in ArkLib's words; that work, its licence and the permission were not seen) and the public project proximity-prize; lean/ThirdParty/README.md lists them.

Numbered items. Table 13 lists the numbered definitions and statements that follow Lean files which `lean/CREDITS.md` lists under four of its headings, and, in four further rows, those over files marked as written from [CFW26], files that follow or build on further work of the three researchers whom the acknowledgments name first [CFWpc], or files not counted as written independently of that work or of ArkLib. That page also points to the published works that the files cite and names, for adapted code, Mathlib [Mathlib], Lean 4's core library and the BLAKE2 reference source code package [BLAKE2]. Under two of the four headings that page lists files in which a comment names a public work: fifteen files name the Lean library VCV-io [VCVio], and one, `lean/IntegerModel/Model.lean`, names the file `modeling_llama4.py` of Hugging Face transformers, version 5.17.0 [transformers].

Among these works: a file that Definition A.1, Definition A.18, and Lemma A.19 follow takes the idea of the doomed set from [Hol19] and [CCHLRR18] and names [BGKTTZ23] and [BGTZ23] as related work; a file that Lemma A.24 follows cites [DP23]. In Table 13, as in Table 11, [CFW26] without a place is that work as its authors gave it to this project (Appendix E.1).

Table 13: Numbered items that follow credited Lean files.

The Lean files	Numbered items
Listed under "ArkLib" [ArkLib]	Lemmas 5.13, A.24
Listed under "VCV-io"	Definitions 2.6, 2.7; Theorem 5.1
Listed under "The model" (values that depend on Meta's models)	Definitions 3.1, 3.2, 3.6, 3.7, 3.8, 5.16, D.1; Constructions 4.15, D.3; Assumption 4.11; Theorems 5.1, 5.17, 5.19; Propositions 3.9, 5.6, 5.18, 6.17, B.1, B.3, B.24; Corollaries 5.8, B.9, B.16
Listed under "Hugging Face transformers"	Definitions 3.1, 3.2, 3.6; Construction D.3
Marked as written from [CFW26]	Definitions 2.2, 4.3, 4.5, 4.6, 5.4, A.1, A.2, A.6, A.7, C.3, C.4, C.5, C.6, C.13; Constructions 4.13, B.8; Theorem 6.3; Proposition 5.14; Lemmas 5.13, A.21, A.24, A.25, A.30, A.33, C.9, C.10, C.12
Follow or build on further work of the three researchers whom the acknowledgments name first [CFWpc]	Definitions 2.3, 2.6, 2.7, 2.13, 2.14, A.1, A.2, A.6, A.7, A.18, C.22, C.23; Construction 4.13; Theorem 5.1; Proposition C.26; Lemmas A.11, A.30, C.11; Corollaries 5.10, B.14
Do not count as written independently of that work	Definitions 2.3, A.1, A.7, A.18, C.2, C.4, C.5, C.6, C.18, C.20, C.22; Construction 4.13; Theorems 6.3, 6.6, 6.9; Proposition C.26; Lemmas C.11, C.12, C.27; Corollaries 6.7, B.11, C.29
Do not count as written independently of ArkLib [ArkLib]	Construction 4.13; Theorem 5.1 (through its statement); Lemmas 5.13, A.24, A.29

The last four rows do not come from `lean/CREDITS.md` or from its table of files. The row on [CFW26] rests on a mark that stood at the head of Lean files while the library was written. The mark named no title; we read it as meaning that work. It was set on groups of files, in some directories on every file, and for a time on every new file, and it passed to files that a program made from marked ones; so it may name more files than were written from that work, and it does not show, file by file, what a file's writer had at hand.

The row on [CFWpc] is over files that comments in the Lean files tie to that work, and over files that state, prove or use the definitions of the Lean files that follow or render it. The row after it is over files that do not count as written independently of that work. The last row is over the files adapted from ArkLib, which that page lists too, and

over files whose writers had seen its source or had it available while they wrote: the heads of some say so, and for the others the ground is the record of who wrote them, which is not in this repository.

Some places in further Lean files are like places in public Lean libraries (ArkLib, VCV-io, Mathlib and others); this table does not list items for that likeness. Among the items that follow such files are Definitions 2.1, 2.13 and 2.14. The lists behind the rows on [CFW26] and [CFWpc] are known to lack files; no other list of files behind this table, on that page or here, is claimed to be exhaustive.

The Lean statement of Theorem 5.1 is over a description (Construction 4.13) that uses definitions from twelve files whose writers had seen the source of ArkLib or had it available while they wrote: three files under `lean/Johnson/Bridge/`, `lean/Johnson/Desc/Lens.lean`, `lean/Johnson/Desc/Ok/FormRules.lean`, `lean/Johnson/Desc/First/Parts.lean` and six files under `lean/Johnson/Desc/First/TailClaims/`. The heads of the three files under `lean/Johnson/Bridge/` and of `lean/Johnson/Desc/Lens.lean` say so; for the other eight the ground is the record of who wrote them, which is not in this repository. The twelve may not be all. This paper does not decide whether these files call for further credit.

F Notation and Lean names

Table 14 lists the notation. The list below gives, for the numbered statements and the section that it names, the Lean names that each renders. Elsewhere in this paper a Lean name stands only where a sentence speaks of that declaration itself.

Table 14: Notation.

Symbol	Meaning
H, H', λ	hash function (random oracle); second hash function (fixed); output length of H
Z; $\mathsf{Adm}(Z)$; $\mathsf{Chk}(Z)$	configuration; scope test; parameter checks
M; A; wt	index (weights); digit matrix, $d \times 2^n$, $n = n_0 + v$; map from matrices in range to weights
x, y; $\mathcal{V}$; L; ξ	prompt, output; vocabulary; length of the window; encoded request
Serv_μ	instances that can be served, with their outputs
c, π; k; dg_Z, par	commitment, proof; key $(\mathsf{rt}_0, \sigma_0)$; digest of Z, second public reading
π_{key}, $\mathsf{IOP}_Z^{\mathrm{key}}$, ν_{off}, κ_{off}	one object, the *offline proof*, on the first table $\mathbf{T}_0$: proof string, IOP, columns opened, bits of proof of work; stored in the key file
π_{link}, $\mathsf{IOP}_Z^{\mathrm{link}}$, ν_{first}, ν_{copy}	the *link proof*, on $\mathbf{T}_0$ and a copy $\mathbf{T}_j$; stored in the file of copy j
π_{req}, $\mathsf{IOP}_Z^{\mathrm{req}}$, ν_{on}, κ_{req}, κ_{bat}	the *online proof*, of an encoded request ξ on a copy; stored in a proof file
code; Seed	codes as bit strings; OOD seed of a root
build, fill	map from (x, y, fill) to an encoded request; filling tokens
$\mathbb{F}$, p; $\mathbb{E} = \mathbb{F}[\vartheta]$	prime field and its order; extension field of the verifier's samples
$\mathrm{eq}(z, j)$	$\prod_k (z_k j_k + (1 - z_k)(1 - j_k))$, with j_k bit k of j, the lowest first
D; ℓ, ℓ_0; t; s	positions; entries of a row's message, plain entries; opening budget; answers a row
$\mathsf{Enc}(f; \tau)$; $\mathbf{T}$, $\mathbf{T}[\phi]$	masked encoding; committed table with 2^v rows, its column at $\phi \in D$
ς, $\mathbf{a}$, σ	OOD seed, OOD answers, OOD sample $(\varsigma, \mathbf{a})$
e, δ, ρ	radius in positions; $\delta = e/\lvert D\rvert$; rate $(\ell + t)/\lvert D\rvert$
Λ, Coll, Dec	list of a word; collision at a seed; decoder
$\mathbf{T}_0$; $\mathbf{T}_j$; rt_j	first table; copy j, $1 \le j \le K$; Merkle root of $\mathbf{T}_j$
$\nu_{\mathrm{off}}, \nu_{\mathrm{first}}, \nu_{\mathrm{copy}}, \nu_{\mathrm{on}}$	maximum number of opened columns: offline proof; link proof in $\mathbf{T}_0$, in its copy; batch
K; N_j; R; G	number of copies; number of batches on copy j; largest batch; segments
$\mathsf{IOP}_Z^{\mathrm{key}}, \mathsf{IOP}_Z^{\mathrm{link}}, \mathsf{IOP}_Z^{\mathrm{req}}$	the three IOPs of $\mathsf{FlashProve}_Z$
Req_Z	request relation
(h, ψ, η, b); w	claim on the table; $w = h^\top A$
$\psi^* = (\psi_0^*, \gamma)$; u; ω; χ	shared point; rows folded at γ; $\omega = A\,\mathrm{eq}(\psi^*, \cdot)$; second sampled point
m, κ_m, ε_m; $\varepsilon_{\mathrm{rd}}$	round, its bits of proof of work, its error; bound on every round's error
$\kappa_{\mathrm{req}}, \kappa_{\mathrm{bat}}, \kappa_{\mathrm{off}}$	values of κ_m: a request's own rounds; shared rounds; offline and link proofs
Π_Z, V_Z, Ext_Z	the argument, its verifier, its extractor
q; $\varepsilon(q)$	hash queries of committer and prover together; soundness error

Section 2

2.1 Random oracle, query bound, trace: `VerifiableInference.Hash`, `VerifiableInference.uniformHash`, `VerifiableInference.RandomizedAlgorithm`, `VerifiableInference.DeterministicAlgorithm`, `VerifiableInference.RandomizedAlgorithm.QueryBound`, `VerifiableInference.QueryLog`, `VerifiableInference.RandomizedAlgorithm.withQueryLog`.

2.3 Interactive oracle proof, batch: `Iop.InteractiveOracleProof`, `Iop.InteractiveOracleProof.IsLocal`, `Iop.BatchInteractiveOracleProof`.

2.5 Proof of work (grinding): `Iop.InteractiveOracleProof.grind`, `Iop.GrW`, `Iop.GrW.ok`.

2.6 Indexed relation, commit-and-prove argument: `VerifiableInference.ArgumentSystem`, `VerifiableInference.ArgumentSystem.uploadRelation`, `VerifiableInference.ArgumentSystem.uploadArgument`, `ZKCP.IndexedRelation`, `ZKCP.IndexedRelation.L`, `ZKCP.Arg`, `ZKCP.Gen.Arg`.

2.7 Index-binding knowledge soundness: `VerifiableInference.Sound`, `ZKCP.HasIndexBinding`, `ZKCP.bindingExp`, `ZKCP.bindingBad`, `VerifiableInference.sound_iff`.

2.9 Correctness of extraction: `VerifiableInference.ExtractsCommittedWeights`.

2.10 Extractability of the commitment, no two openings: `VerifiableInference.BindsWith`, `VerifiableInference.NoTwoOpenings`.

2.12 Completeness on a set of triples: `VerifiableInference.CompleteOn`, `VerifiableInference.Complete`.

2.13 Real and ideal experiment: `ZKCP.Gen.Arg`, `ZKCP.Gen.realExp`, `ZKCP.Gen.idealExp`, `ZKCP.Gen.ViewEnv`, `ZKCP.Gen.Simulator`, `ZKCP.Gen.View`, `VerifiableInference.realExperiment`, `VerifiableInference.simulatedExperiment`.

2.14 Index privacy: `ZKCP.Gen.PrivacyOn`, `ZKCP.Gen.PrivacyFor`, `ZKCP.Gen.HasIndexPrivacyWith`.

Section 3

3.1 Integer model: `VerifiableInference.ModelWeights`, `VerifiableInference.settingsAreValid`, `VerifiableInference.modelNextToken`, `Policy.modelLogits`.

3.2 Greedy token, greedy output: `VerifiableInference.modelNextToken`, `VerifiableInference.ArgumentSystem.IsModelOutput`, `Policy.modelNextToken_eq_firstArgmax`.

3.4 Logit gap, acceptance policy, exact match, clipped gap: `Policy.logitGap`, `Policy.tokenCost`, `Policy.outputCost`, `Policy.countedTokens`, `Policy.HasScores`, `Policy.MeetsPolicy`, `Inst.Model.AcceptancePolicy`, `Inst.Model.AcceptancePolicy.exact`.

3.5 Exact match and greedy outputs: `Policy.meetsPolicy_exact_iff`, `Policy.meetsPolicy_of_isModelOutput`.

3.6 Model parameters: `IntegerTransformer.Settings`, `VerifiableInference.settingsInRange`, `VerifiableInference.settingsAreValid`, `Inst.Model.Lawful`.

3.7 Configuration, default configuration: `VerifierAndGoals.PublicFiles`, `VerifierAndGoals.storedPublicFiles`, `VerifierAndGoals.zGap`, `VerifierAndGoals.storedLines`, `VerifierAndGoals.policyFile`.

3.8 Admissible configuration: `VerifierAndGoals.InScope`, `VerifierAndGoals.readFiles`, `VerifierAndGoals.settingsOfRead`, `VerifierAndGoals.FreeLines`, `VerifierAndGoals.policyFile`.

3.9 All-zero weights: `Policy.meetsPolicy_zeroWeights`.

3.11 Instances that can be served: `VerifiableInference.CanServe`, `IntegerModel.zeroFilledWindow`.

3.12 Non-vacuity of the instances that can be served: `VerifiableInference.validSettings_exist`, `VerifiableInference.canServe_exists`, `VerifiableInference.canServe_nowhere_atHaltingWeights`, `VerifiableInference.referenceSettings_valid`.

3.13 Which instances can be served: `VerifiableInference.Compilation.complete_at_the_stored_files_at_weights_within_800`, `VerifiableInference.canServe_iff_of_passesSomeWorstCaseCheck`, `VerifiableInference.passesSomeWorstCaseCheck_atReferenceSettings`, `VerifierAndGoals.AtFiles.settingsOfRead_named`.

Section 4

4.1 Digit matrix: `Cm.Idx`, `Inst.Model.Layout.WFW`, `Cm.Idx.weights`, `VerifiableInference.OnPublicFiles.Flat.InRange`.

4.2 Masked encoding, committed table, opening budget: `OnBytes.Flat.codingCells`, `OnBytes.Flat.cellEntryAt`, `OnBytes.Flat.codingHonest`.

4.3 Out-of-domain seed, answers, sample: `Iop.HolographicProtocol.Rec`, `Hiop.ConfigurationWithOutOfDomainSample.smp`, `OnBytes.Slice.recZ`.

4.5 List, collision, decoder: `OnBytes.Slice.DecR.dcd`, `OnBytes.Slice.DecR.coll`, `OnBytes.Slice.DecR.configurationOfStoredTable`, `Hiop.ConfigurationWithOutOfDomainSample.D`, `OnBytes.Flat.cellsOf`.

4.6 First table, copy, link proof: `Iop.HolographicProtocol.link`, `VerifiableInference.OnPublicFiles.CompilerSoundnessConditionStatements.entrySound`.

4.7 Encoded request, offline and online proof, files: `IntegerModel.requestOf`, `VerifiableInference.OnPublicFiles.Flat.buildReq`, `VerifiableInference.OnPublicFiles.Flat.Req`.

4.8 Segments, cells, request relation: `VerifiableInference.OnPublicFiles.Flat.RelLAt`, `Policy.LedgerRel.Rno`, `Policy.LedgerRel.Cond`, `OnBytes.Flat.claimsReqAt`.

4.9 Meaning of the request relation: `Policy.ledger_relation_at_protocol_prime`, `Policy.IsTrueUnder`, `Policy.ledger_sub`.

4.10 Random oracle: `VerifiableInference.Hash`, `VerifiableInference.uniformHash`, `VerifiableInference.Prob`, `Fmt.oracle`, `Fmt.size_oracle`.

4.11 Separate second hash: `VerifierAndGoals.atSecondHashFunction`, `OnBytes.secondHashFunction`.

4.13 The interactive protocol $\mathsf{FlashProve}_Z$: `VerifierAndGoals.theDescription`, `Iop.HolographicProtocol`, `VerifiableInference.OnPublicFiles.CompilerSoundnessConditionStatements.keySound`, `VerifiableInference.OnPublicFiles.CompilerSoundnessConditionStatements.entrySound`, `VerifiableInference.OnPublicFiles.CompilerSoundnessConditionStatements.proofSound`.

4.14 Parameter checks: `FlashProve.protocolDataWellFormed`, `FlashProve.protocolDataWellFormedAt`, `FlashProve.dataWellFormedAtFullSizeDepth`, `VerifiableInference.OnPublicFiles.sizeNumbersFitAtFullSize`, `LPFD.lawsAsk`, `VerifiableInference.Desc.FileDescription.planOk`.

4.15 The argument Π_Z: `VerifierAndGoals.systemOfRecord`, `VerifierAndGoals.atSecondHashFunction`, `VerifierAndGoals.theDescription`, `VerifierAndGoals.verify`, `VerifierAndGoals.verifyAt`,

VerifiableInference.OnPublicFiles.innerCompiledVerifier, TypedOracleCompiler.Compile.Batch.Cms.verifyAtFullKey, ZKCP.verifyDigestKey, ZKCP.Arg.buildInstance, FlashProve.theExtractor.

4.17 True request, constraints, honest bound: VerifiableInference.IsTrueRequest.

4.18 True requests and greedy outputs: IntegerModel.isModelOutput_of_isTrueRequest, IntegerModel.isTrueRequest_of_canServe, VerifiableInference.canServe_weightsOfIndex_iff, VerifiableInference.canServe_iff, VerifiableInference.modelNextToken_eq_nextToken, VerifiableInference.lawful_scalesOf, VerifiableInference.settingsOf_scalesOf.

Section 5

VerifierAndGoals.AtTheName.read_of_scope: each statement below whose Lean form asks that the three byte strings decode is joined with it.

5.1 Soundness: VerifierAndGoals.AtTheName.sound_of_verify_at, VerifierAndGoals.AtTheName.nearestSound_of_verify, VerifierAndGoals.AtTheName.indexBinding_of_verify_at, VerifierAndGoals.AtTheName.indexBindingWith_of_verify_at.

5.2 What the theorem gives: VerifierAndGoals.AtTheName.indexBindingWith_of_verify_at.

5.4 Executable form: PV.Ride.theForm, FlashProve.FormBytes, VerifiableInference.OnPublicFiles.hashOf.

5.5 The executable form computes the verifier: PV.Ride.theForm_eq_verify, PV.Ride.theForm_total, PV.Ride.theForm_at_hash, PV.Ride.verdictThe_eq, PV.Ride.verdictThe_some.

5.6 The parameter checks pass at the default configuration: VerifierAndGoals.filesRead, VerifierAndGoals.inScope_zAt, AtFullSide.Top.theLive_zFull, AtFullSide.Top.theLive_zGap, AtFullSide.Top.theLive_zAt.

5.8 At the default configuration: VerifierAndGoals.AtTheName.sound_of_verify_at, VerifierAndGoals.AtTheName.indexBindingWith_of_verify_at, VerifierAndGoals.filesRead, AtFullSide.Top.theLive_zFull, PV.Ride.theForm_eq_verify, AtFull.Top.sound_zFull.

5.9 What a client gets: PV.Ride.theForm_accepts_only, PV.Ride.sound_of_the_form_events.

5.10 Many proofs: VerifierAndGoals.AtTheName.soundManyProofs_union_of_verify_at, VerifierAndGoals.AtTheName.sessionAugmented_union_of_verify_at.

5.11 Soundness under any policy: VerifierAndGoals.AtTheName.soundUnderPolicy_of_verify_at, VerifierAndGoals.AtTheName.indexBindingUnderPolicy_of_verify_at.

5.13 Mutual correlated agreement strictly below the Johnson radius: Cb.BCHKS46At, Cb.Bridge.bchks46At, Cb.Gl.bchks46At.

5.14 Round errors: AtPublicFiles.AtFiles.soundAsIOPAtTheProtocol, VerifierAndGoals.AtTheName.keySound_at_live_holds, FlashProve.entrySound, VerifierAndGoals.AtTheName.proofSoundAt_of_verify.

5.16 Explicit committer: Bind.Cpp.theCpp, Bind.Cpp.theTwin, Bind.Cpp.cppSide, Bind.Cpp.cppCommit, Bind.Twin.twinID, Bind.Twin.commitW.

5.17 Extraction and extractability for the explicit committer: Bind.Cpp.extractsCommitted, Bind.Cpp.bindsAt, Bind.Cpp.noTwoOpenings_live, VerifierAndGoals.bindingErrorInQueries, Bind.Cpp.twinBudget.

5.18 Sharper errors: Bind.Cpp.extractsCommitted_square_salt, Bind.Cpp.bindsWith_of_tieAt, Bind.Cpp.tieAt, Bind.Cpp.saltAt_of_live, H2K.Sizes.inScope_readFiles, Bind.Cpp.extracts_salt, Bind.Cpp.extracts_square_salt, VerifierAndGoals.Fit.epsBind.

5.19 Soundness after a commitment by the explicit committer: Bind.Cpp.soundAtCommitted_own, Bind.Cpp.soundAtCommitted_exact_nonempty.

5.20 Sizes of the errors: Bind.Cpp.honest_count, Bind.Cpp.bindsAtZ_cpp.

5.21 Scope of the statements on the commitment: AtFullSide.Tie.extractsCommitted_zFull, AtFullSide.Tie.bindsAt_zFull, AtFullSide.Tie.noTwoOpenings_zFull, AtFullSide.Tie.extractsCommitted_zAt, AtFullSide.Tie.bindsAt_zAt, AtFullSide.Tie.noTwoOpenings_zAt, Bind.Cpp.rejects_of_dead.

Section 6

6.2 Constant-message prover: OnBytes.WholeKey.silent, OnBytes.Flat.Files.fixedMessageProvers, AtPublicFiles.Premises.silentAtC, OnBytes.WholeKey.extEnc_silent.

6.3 Committed tables before compilation: AtPublicFiles.AtFiles.tables_own, AtPublicFiles.AtFiles.tables_own_named, AtPublicFiles.Reads.hidE_filesAt.

6.5 The opening budget is tight for one table: AtPublicFiles.Table.every_set_objAt, AtPublicFiles.Table.every_set_filesAt, AtPublicFiles.Table.not_hidE_filesAt, AtPublicFiles.AtFiles.not_hidE_own.

6.6 Reduction of index privacy to five conditions: AtPublicFiles.Premises.privacyFromFive_description, OnBytes.WholeKey.wholeKeyPrivacy, OnBytes.WholeKey.zetaW.

6.7 Committed tables after compilation: AtPublicFiles.ZetaAt.privacyWithBound_description, AtPublicFiles.Shown.keyInGame_description, AtPublicFiles.Shown.sessionsInGame_description, AtPublicFiles.ZetaAt.zeta_number_caps, AtPublicFiles.Shown.allAtOnce_description, AtPublicFiles.ZetaAt.privacyWithNumber_description, AtPublicFiles.ZetaAt.sMT_description.

6.9 Link proof before compilation: OnBytes.LinkSim.AtHome.homeExactAt, OnBytes.LinkSim.AtHome.home_withheld, OnBytes.LinkSim.AtHome.home_beside, OnBytes.LinkSim.AtHome.homeSide_accepted, FlashProve.linkFits_at, Johnson.Prover.Link.link_yes.

6.11 Configuration in use: VerifiableInference.PublicSetup, VerifiableInference.Compilation.FilesInUse, VerifiableInference.Compilation.FilesTaken, AtPublicFiles.AtFiles.read_of_inScope.

6.12 Served instance, label, bit: VerifiableInference.AtEveryRequest.WellFormedRequest, VerifiableInference.RequestForm, VerifiableInference.AtEveryRequest.requestLabel, VerifiableInference.AtEveryRequest.Label, VerifiableInference.AtEveryRequest.requestOutcome, VerifiableInference.AtEveryRequest.Outcome, VerifiableInference.AtEveryRequest.modelResult, VerifiableInference.AtEveryRequest.served_iff_canServe, VerifiableInference.AtEveryRequest.wellFormed_and_result_iff, VerifiableInference.AtEveryRequest.requestLabel_eq_served_iff, VerifiableInference.AtEveryRequest.requestLabel_eq_failed_iff, VerifiableInference.AtEveryRequest.requestLabel_eq_otherTokens_iff, VerifiableInference.AtEveryRequest.bitOf_requestLabel, VerifiableInference.canServe_iff_requestResult, VerifiableInference.canServe_and_modelOutput_iff, VerifiableInference.requestForm_iff_requestOfTokens, VerifierAndGoals.valid_of_read.

6.14 Lean provers: VerifiableInference.Compilation.

theProvers, AtPublicFiles.AtFiles.HonestTerm.honestProvers, AtPublicFiles.AtFiles.HonestTerm.thePartsAt, AtPublicFiles.AtFiles.ServedRequest, VerifiableInference.Compilation.proversAtFiles, VerifiableInference.Compilation.completeOnServedRequests_refuse_iff, VerifiableInference.Compilation.theProvers_eq, FlashProve.backThe, OnBytes.FrontRoad.inRange_backThe, FlashProve.sec_the.

6.15 Compiled Lean argument: VerifiableInference.flashProve, VerifiableInference.Compilation.honestProversAndSettings, VerifiableInference.Compilation.finalSystem, VerifiableInference.Compilation.flashProveAtFiles, VerifiableInference.Compilation.compile, VerifiableInference.Compilation.compiledSide, VerifiableInference.Compilation.compile_flashProve_verify_eq, VerifiableInference.Compilation.finalExtractor, VerifiableInference.Compilation.noProofForm, VerifiableInference.Compilation.compiled_provers_are_the_triple, VerifiableInference.Compilation.compiled_protocol_is_the_bound, OnBytes.FrontRoad.familyLaws_the.

6.16 Arguments with the same verifier: VerifiableInference.sound, VerifiableInference.Compilation.finalSound, VerifiableInference.Compilation.compile_flashProve_sound_iff.

6.17 The hypotheses are met: VerifiableInference.storedSetup, VerifiableInference.Compilation.filesInUse_stored, VerifierAndGoals.storedPublicFiles, VerifiableInference.Compilation.someRequestIsServed_holds, VerifiableInference.Compilation.zkOnServedRequests_inScope.

6.19 Whole run: Iop.HolographicProtocol.wholeRunShape, Iop.HolographicProtocol.WholeRun, Iop.HolographicProtocol.WholeRun.Accepted, Iop.HolographicProtocol.WholeRun.Concludes.

6.20 Soundness of a whole run before compilation: AtPublicFiles.AtFiles.probabilityInScope, AtPublicFiles.AtFiles.probabilityInScope79, AtPublicFiles.AtFiles.bound_le_79, AtPublicFiles.AtFiles.soundAsIOPAtTheProtocol, AtPublicFiles.AtFiles.roundsInScope, AtPublicFiles.AtFiles.relationAtFiles, AtPublicFiles.AtFiles.ConclusionFails, AtPublicFiles.AtFiles.clearingIsTheWires.

6.21 Reading the bound on a whole run: AtPublicFiles.AtFiles.soundAsIOPAtTheProtocol, AtPublicFiles.AtFiles.wholeRunAtFiles, AtPublicFiles.AtFiles.mostWorkInScope, AtPublicFiles.AtFiles.roundsInScope.

6.22 Completeness of provers for the three IOPs: VerifiableInference.Compilation.CompleteIOP, VerifiableInference.Compilation.CompleteInParts, Iop.FailsRarely, Iop.FailsAt, Iop.chanceAtPassing, Iop.PartsWork.Faithful, AtPublicFiles.AtFiles.workFaithful_atFiles.

6.23 Completeness before compilation: AtFullSide.Honest.complete_in_scope, AtPublicFiles.AtFiles.HonestTerm.CompleteAtFiles, AtPublicFiles.AtFiles.CompleteOnServedRequests, AtPublicFiles.AtFiles.completenessTarget.

6.24 Exact simulation up to private aborts: VerifiableInference.Compilation.RunByRun, VerifiableInference.Compilation.ExactOutsideStops, VerifiableInference.Compilation.PrivateStopsAtMost, VerifiableInference.Compilation.RunSimulators, VerifiableInference.Compilation.HonestVerifierZeroKnowledge.

6.25 Table experiment with served bits: VerifiableInference.Compilation.WholeLife, VerifiableInference.Compilation.commitmentLife, VerifiableInference.Compilation.everyRequest, VerifiableInference.Compilation.realSide, Hiop.Seeded.EnvF.Adm, Hiop.Seeded.HostF.view.

6.26 Simulated table experiment: VerifiableInference.Compilation.simulatedRuns, VerifiableInference.Compilation.simulatorsByTable, Hiop.Seeded.simWholeF, Hiop.Seeded.ViewNearF.

6.27 Honest-verifier zero knowledge before compilation: VerifiableInference.Compilation.zkOnServedRequests_inScope, VerifiableInference.Compilation.ZeroKnowledgeOnServedRequests, VerifiableInference.Compilation.life_number_at_most, VerifiableInference.Compilation.life_err_at_zero_zero, VerifiableInference.Compilation.budgetAtFiles.

6.29 Completeness after compilation: VerifiableInference.completeOnCanServe, VerifiableInference.completeAtWellFormedRequests, VerifiableInference.Compilation.theFinalStatementHolds, VerifiableInference.Compilation.FinalComplete, VerifiableInference.AtEveryRequest.ProofAcceptedWhere, VerifiableInference.Compilation.finalComplete_iff_completeOn, VerifiableInference.Compilation.finalAtT1_filled_withRoom.

6.30 Reading completeness: VerifiableInference.Compilation.rejectsNoProof_value.

6.31 Soundness with extractability, openings and extraction: VerifiableInference.soundAndBinding, VerifiableInference.Compilation.finalSoundAndBinding_value, VerifiableInference.Compilation.bindingOf_value, VerifiableInference.Compilation.finalExtracts_value, VerifiableInference.Compilation.BindingOf, VerifiableInference.Compilation.finalBindingError, VerifiableInference.Compilation.finalOpeningError, VerifierAndGoals.Fit.epsBind_min_le_bitsBinding, VerifiableInference.Compilation.finalSoundnessError_le_finalBindingError, VerifiableInference.Compilation.bitsBinding_le_finalBindingError, VerifierAndGoals.Fit.BindsWith.min_one, VerifiableInference.BindsWith.mono.

6.33 Empty proofs, queries and opened columns: VerifiableInference.refusedHandsOutNoProof, VerifiableInference.repliesFit, VerifiableInference.rejectsEmptyProof, VerifiableInference.Compilation.finalHandsOutNoProof_value, VerifiableInference.Compilation.handsOutNoProof_value, VerifiableInference.Compilation.FinalRefusedHandsOutNoProof, VerifiableInference.Compilation.finalRefusedHandsOutNoProof_value, VerifiableInference.Compilation.rejectsNoProof_value, VerifiableInference.Compilation.FinalHandsOutNoProof, VerifiableInference.Compilation.RepliesFit, VerifiableInference.Compilation.RepliesFitAtFiles, VerifiableInference.Compilation.TableBudgetFits, VerifiableInference.AtEveryRequest.WithinBudget, VerifiableInference.Compilation.theFinalStatementHolds.

6.34 Zero knowledge with request leakage: VerifiableInference.AtEveryRequest.ZeroKnowledgeWhere, VerifiableInference.AtEveryRequest.ZeroKnowledgeBelowWhere, VerifiableInference.AtEveryRequest.ZeroKnowledgeShownBelowWhere, VerifiableInference.

AtEveryRequest.Simulator, VerifiableInference.AtEveryRequest.simulatedExperiment, VerifiableInference.realExperiment, VerifiableInference.labelled, VerifiableInference.ZeroKnowledgeOn, VerifiableInference.Simulator, VerifiableInference.simulatedExperiment.

6.35 Zero knowledge on served instances: VerifiableInference.zeroKnowledgeOnCanServe, VerifiableInference.ZeroKnowledgeOn, VerifiableInference.PublicSetup, VerifiableInference.flashProve, VerifiableInference.Compilation.compile, VerifiableInference.CanServe, VerifiableInference.Compilation.finalZeroKnowledgeError, VerifiableInference.Compilation.finalVerifierQueries, VerifiableInference.Compilation.finalReplyCount.

6.37 Zero knowledge at every instance: VerifiableInference.zeroKnowledgeAtEveryRequest, VerifiableInference.zeroKnowledgeAtEveryRequest_of_lines, VerifiableInference.zeroKnowledgeBelow_of_onCanServe, VerifiableInference.AtEveryRequest.zeroKnowledgeFollowsFromOn, VerifiableInference.AtEveryRequest.ZeroKnowledgeFollowsFromOn, VerifiableInference.AtEveryRequest.served_iff_canServe, VerifiableInference.refusedHandsOutNoProof, VerifiableInference.Compilation.finalServedIsModelOutput, VerifiableInference.Compilation.zeroKnowledgeOn_of_domain, VerifiableInference.Compilation.theFinalStatementOneBitHolds_at, VerifiableInference.Compilation.finalZeroKnowledgeOneBit_at, VerifiableInference.Compilation.stepErrorAtTheStops, VerifiableInference.Compilation.theFinalStatementOneBitHolds, VerifiableInference.Compilation.theFinalStatementBothHold, VerifiableInference.Compilation.FinalZeroKnowledgeOneBit.

6.38 Zero knowledge with the label shown: VerifiableInference.zeroKnowledgeWithLabel, VerifiableInference.Compilation.theFinalStatementHolds_at, VerifiableInference.Compilation.theFinalStatementOneBitHolds_at, VerifiableInference.Compilation.finalZeroKnowledge_of_oneBit, VerifiableInference.Compilation.theFinalStatementHolds, VerifiableInference.Compilation.finalStatement_of_oneBit, VerifiableInference.Compilation.FinalZeroKnowledge.

6.39 Reading zero knowledge after compilation: VerifiableInference.AtEveryRequest.toldWeightsSaysNothing, VerifiableInference.Compilation.served_at_T1, VerifiableInference.Compilation.termLeft_at.

6.46 The two together: VerifiableInference.Compilation.threeAtFiles_in_use, AtPublicFiles.OneProtocol.ThreeAtOneProtocol, AtPublicFiles.OneProtocol.ThreeAtFiles.

6.47 Definitions of the properties: VerifiableInference.CompleteOn, VerifiableInference.ZeroKnowledgeOn, VerifiableInference.Complete, VerifiableInference.ZeroKnowledge.

Section 6.5 Completeness and zero knowledge of the Lean provers: VerifiableInference.Compilation.theProvers, AtPublicFiles.AtFiles.HonestTerm.honestProvers, AtPublicFiles.AtFiles.HonestTerm.thePartsAt, AtPublicFiles.AtFiles.soundAsIOPAtTheProtocol, AtPublicFiles.AtFiles.probabilityInScope, AtFullSide.Honest.complete_in_scope, VerifiableInference.Compilation.someRequestIsServed_holds, VerifiableInference.Compilation.zkOnServedRequests_inScope, VerifiableInference.Compilation.finalAtT1_filled, VerifiableInference.Compilation.theFinalStatementHolds, VerifiableInference.Compilation.theFinalStatementOneBitHolds, VerifiableInference.Compilation.finalStatement_of_oneBit, VerifiableInference.Compilation.theFinalStatementBothHold, VerifiableInference.Compilation.rejectsNoProof_value, VerifiableInference.Compilation.finalExtracts_value, VerifiableInference.Compilation.life_number_at_most, AtPublicFiles.AtFiles.probabilityInScope79, VerifiableInference.Compilation.threeAtFiles_in_use, VerifiableInference.Compilation.filesInUse_stored, VerifiableInference.Compilation.someRequestIsServedInUse.

6.49 Completeness in Lean: FlashProve.completeOn_last, FlashProve.LastArgs, FlashProve.LastFits, FlashProve.verify_record, FlashProve.caps_home, FlashProve.number_home, FlashProve.bytes_forty, FlashProve.lastFits_idle, FlashProve.three_le_idleAlpha, AtFullSide.Home.lastFits_idle_named, Meaning.served_met, VerifierAndGoals.settingsOfRead_full, VerifiableInference.canServe_nowhere_atHaltingWeights.

Section 8

8.15 Many proofs without the factor N: VerifierAndGoals.AtTheName.soundManyProofs_of_verify_at.

Appendix A

A.1 Round-by-round soundness: Iop.InteractiveOracleProof.Sound, Iop.Doomed, Iop.BatchInteractiveOracleProof.SndB, Iop.JointLaw.

A.2 Early part, knowledge soundness: Iop.BatchInteractiveOracleProof.SndBL, Iop.BatchInteractiveOracleProof.guardP, Iop.BatchInteractiveOracleProof.SndBP, Iop.BatchInteractiveOracleProof.SndBKT.

A.3 The argument over separate oracles: VerifiableInference.OnPublicFiles.schemeOverTypedOracles, TypedOracleCompiler.Compile.Batch.Cms.verifyAtKeyRoot, TypedOracleCompiler.Compile.Batch.Cms.verifyAtFullKey, TypedOracleCompiler.Compile.Batch.Cms.indexExtractorAtRootAndOutOfDomainSample, TypedOracleCompiler.Compile.Batch.Cms.tVC, TypedOracleCompiler.Compile.Batch.Cms.detBound_VRA.

A.4 Word under a root, rooted query, seed-first trace: TypedOracleCompiler.Compile.Batch.extEnc, TypedOracleCompiler.Compile.Batch.Announced, TypedOracleCompiler.Compile.Batch.rooted, TypedOracleCompiler.Compile.Batch.Cms.SeedsFirstAll, TypedOracleCompiler.Compile.Batch.Cms.preAll, OnBytes.Slice.between.

A.5 The event Bad: TypedOracleCompiler.Compile.Batch.Cms.BadA, TypedOracleCompiler.Compile.Batch.EM, TypedOracleCompiler.Compile.Batch.EF, TypedOracleCompiler.Compile.Batch.stepS, TypedOracleCompiler.Compile.Batch.Lane.step, TypedOracleCompiler.Compile.Batch.Cms.stepSeed.

A.6 Conditions on soundness: VerifiableInference.OnPublicFiles.CompilerSoundnessConditionStatements.keySound, VerifiableInference.OnPublicFiles.CompilerSoundnessConditionStatements.entrySound, VerifiableInference.OnPublicFiles.CompilerSoundnessConditionStatements.proofSound, Iop.

BatchInteractiveOracleProof.SndBL, Iop.BatchInteractiveOracleProof.SndBKT.

A.7 Conditions of the compiler: VerifiableInference.OnPublicFiles.CompilerSoundnessConditions, VerifiableInference.OnPublicFiles.CompilerSoundnessConditionStatements.seedLaw, VerifiableInference.OnPublicFiles.CompilerSoundnessConditionStatements.wires, VerifiableInference.OnPublicFiles.CompilerSoundnessConditionStatements.commitment.

A.8 Bad is rare: TypedOracleCompiler.Compile.Batch.Cms.cmsA_bcs, TypedOracleCompiler.Compile.Batch.hashEvents, TypedOracleCompiler.Compile.Batch.Cms.pr_step4_le, TypedOracleCompiler.Compile.Batch.pr_stepS_le.

A.9 Outside Bad: TypedOracleCompiler.Compile.Batch.Cms.BcsAtA, TypedOracleCompiler.Compile.Batch.Cms.cmsA_bcs.

A.10 The three relations in sequence: TypedOracleCompiler.Compile.Batch.Cms.coreA.

A.11 Index-binding knowledge soundness over separate oracles: TypedOracleCompiler.Compile.Batch.Cms.bindingGAt, TypedOracleCompiler.Compile.Batch.Cms.cmsA_binding_at, TypedOracleCompiler.Compile.Batch.Cms.cmsRA_binding_at, VerifiableInference.OnPublicFiles.CompilerSoundnessConditions.binds, TypedOracleCompiler.Compile.Batch.Cms.soundnessBoundOverTypedOracles.

A.12 One hash function: TypedOracleCompiler.Compile.Batch.Cms.bindingR_one, Rom.binding_overOne, Rom.OracleCoding.

A.13 Public readings: TypedOracleCompiler.Compile.Batch.Cms.pub_lift, VerifiableInference.OnPublicFiles.told_binds_core, VerifiableInference.OnPublicFiles.LayoutsByPublicReading, VerifiableInference.OnPublicFiles.LayoutsByPublicReading.ext.

A.14 The commitment as the hash of the key: ZKCP.HasIndexBindingWith.digestKey, VerifiableInference.Rec.soundForWith_systemE, VerifiableInference.OnPublicFiles.epsCode.

A.15 The compilation lemma: VerifiableInference.OnPublicFiles.verifierAcceptingNoMoreThanBehindReadingAndParser_soundFor, VerifierAndGoals.Fit.soundForWith_system_of_toldO, VerifiableInference.OnPublicFiles.innerCompiledVerifier_soundFor_byPublicReading, VerifiableInference.soundForWith_via, VerifiableInference.SoundForWith.bindWeights.

A.16 Which form of the lemma: VerifiableInference.OnPublicFiles.innerCompiledVerifier_sound, VerifiableInference.OnPublicFiles.verifierAcceptingNoMoreThanBehindReadingAndParser_soundFor.

A.17 The side conditions hold: FlashProve.compilerSoundnessConditions_layoutsByPublicReading, FlashProve.publicReadingFitAtDescriptionOfFields, VerifierAndGoals.AtTheName.soundUnderPolicyAtZ_numbers, FlashProve.layoutsByPublicReading_spare, FlashProve.layoutsByPublicReading_word, FlashProve.layoutsByPublicReading_code, FlashProve.readingsQueryBound_descriptionOfFields, FlashProve.layoutsByPublicReading_noCoin, FlashProve.layoutsByPublicReading_once.

A.18 Doomed, reduction, leads: Iop.Doomed, Iop.Err, Iop.Err.Below, Iop.Reduction.Leads.

A.19 State functions; composition: TypedOracleCompiler.Compile.Batch.stOf, TypedOracleCompiler.Compile.Batch.stOf_start, TypedOracleCompiler.Compile.Batch.stOf_prover, TypedOracleCompiler.Compile.Batch.stOf_ver, TypedOracleCompiler.Compile.Batch.stOf_full, Iop.Doomed.append, Iop.Reduction.Leads.seq, VerifiableInference.Desc.Plain.fold_chain.

A.20 Proof of work; challenges from hash blocks: Iop.GChal.cnt_pass, OnBytes.Flat.CarryS.doomed, OnBytes.Flat.Maps.below_err, Iop.kapSl.

A.21 From three parts to a batch IOP: TypedOracleCompiler.Joined.proofSound_plainJB, VerifierAndGoals.AtTheName.proofSound_of_rows, VerifierAndGoals.AtTheName.RowsOfProofFile, Iop.Reduction.LeadsB, Iop.Reduction.LeadsBN, Iop.BatchInteractiveOracleProof.SndBKT.

A.22 Where a parameter check fails: FlashProve.keySound_theLay, FlashProve.entrySound_theLay, FlashProve.proofSound_theLay.

A.23 List size strictly below the Johnson radius: Cb.johnson_tables, Cb.johnson_tables_cnt, Cb.listMax, Cb.InJohnson.

A.24 Folding a table: Cb.Bridge.foldErr_ix, Cb.LdIx.Bad, Cb.rows_cost_nothingAt, Cb.rows_free_abs, Cb.rows_avoid, Code.cnt_bad_eqk_mul_sum, OnBytes.Slice.list_johnson.

A.25 Collisions are rare: OnBytes.Flat.entryFull_seedLawK, OnBytes.Slice.recBound, OnBytes.Slice.fr_bound, OnBytes.Slice.coll_count_of_list, Cb.ldFc_overF, OnBytes.Slice.recZ_okOn.

A.26 Round counts: Cb.ldSendFam_cnt, Cb.ldSendFamErr, VerifiableInference.Desc.K9L.num_k9, Cb.psc_know, Cb.ld_sum_know, Cb.Pv.SendLoc.errs, Cb.cnt_ldSwBad1, Cb.cnt_ldSwBad2, Cb.ldSwitch_know, Cb.ldBase_good, TwRound.lb_entN, Cb.ldRng_err, Cb.ldRng_goodT.

A.27 Conditions and claims of $\mathsf{IOP}_Z^{\mathrm{req}}$: VerifiableInference.OnPublicFiles.meansSome, Meaning.firstMeans, VerifiableInference.Desc.First.start, VerifiableInference.Desc.First.openSets, VerifiableInference.Desc.Shared.startS, VerifiableInference.Desc.Shared.endS, VerifiableInference.Desc.Shared.promsAt.

A.28 The first part of $\mathsf{IOP}_Z^{\mathrm{req}}$: VerifiableInference.OnPublicFiles.rows_at_descB, VerifierAndGoals.AtTheName.kindsLaws_at_head, VerifiableInference.Desc.Sound.Piece.GoodIn, Iop.Err.cap, VerifiableInference.Desc.Sound.chainOfPromisesR, P5J.good_of_kinds, VerifiableInference.Desc.First.K2Caller.k2_err_cap, OnBytes.Road.K4.Plan4.herr_cap, VerifiableInference.Desc.First.Att.P.K05_law_one_guard, VerifiableInference.Desc.Kinds.Head.Law.Bears.k06BearsCap, VerifiableInference.Desc.Kinds.Head.Law.Came.k7_own, VerifiableInference.Desc.First.MergeLaw.merge_goodIn_cap, P5J.headA_at_name, P5J.conds_of_laid, P5J.owed_of_live, H2K12.lastTypedAt_at_head.

A.29 The shared part and the last part: P5J.five_at_free2_laid2, Desc.K14.Law.k14_at_free_head, Desc.K14.Law.point_leadsB_at_k14, Desc.K14.Law.ownPt_below, VerifiableInference.Desc.Sound.eachLawsBNOwn_plain, P5J.AtMgA.leads3_of_leftAt, P5J.AtMgA.lastLeft_at_head_alone, P5J.headA_at_name, P5J.conds_of_laid, P5J.owed_of_live.

A.30 The two other IOPs: VerifierAndGoals.AtTheName.keySound_at_live_holds, LPFD.The21, LPFD.keyDebtCap_of_21, LPFD.keySound_of_debt, FlashProve.hole_entrySound, Desc.K15.H_fullV, VerifiableInference.OnPublicFiles.entrySoundOfFullD_of_joh.

A.31 From the open claims to the request relation: VerifiableInference.OnPublicFiles.tie_of_some,

VerifiableInference.OnPublicFiles.TieOwesSome, P5J.meansOfStart, Meaning.layersSay, VerifiableInference.Desc.Kinds.Head.Law.head_cuts_of_guard, P5J.reply_leaves_at_mg, P5J.owed_of_live.

A.33 From the request relation to the policy: OnBytes.Road.L5.sound_ledger, VerifierAndGoals.AtTheName.meaning, VerifierAndGoals.AtTheName.carries, Policy.LedgerRel.Rno, Policy.ledger_noWrap, Cm.TI.eq_of_dvd, Cm.TP.Lay.stream_eqK.

A.34 The extractor: VerifierAndGoals.AtTheName.hext, FlashProve.theExtractor.

A.35 Numbers of queries: VerifierAndGoals.AtTheName.count_the, FlashProve.publicReadingFitAtDescriptionOfFields, VerifierAndGoals.theDescription_within, FlashProve.readingsQueryBound_descriptionOfFields.

A.36 The arithmetic of the bound: VerifierAndGoals.frame_le_bits, VerifierAndGoals.soundnessErrorInQueries, VerifierAndGoals.AtTheName.errorAtBits.

Appendix B

B.1 Policies that every eligible output meets: Policy.meetsPolicy_of_close_scales, Policy.exists_policy_of_close_scales, VerifierAndGoals.AtFiles.every_output_at_meets_iff.

B.3 The example clipped-gap policy: Policy.gaps_of_meetsPolicy_qExample_short, Policy.card_gap_ge_mul_le_qExample, Policy.meetsPolicy_qExample_of_one_free.

B.7 Which compiler: VerifierAndGoals.AtTheName.sound_of_verify_at_through_the_IOP, VerifiableInference.OnPublicFiles.innerCompiledVerifier, TypedOracleCompiler.Compile.Batch.verifyOneProtocol, VerifiableInference.Compilation.compile.

B.8 Algorithm of the executable form: Run.toBytes, Run.bytesToDig, Fmt.bitAt, PV.Ride.verdictThe, PV.Ride.verdictFrom, PV.Ride.rideOf, PV.Ride.letsThe, FlashProve.tiePremB, FlashProve.decodeD, PV.Head.cThru, PV.Head.sevenThru, PV.RunPure.Full.stepsFullAllAt, VerifiableInference.OnPublicFiles.SlotsBehindCutD.fastSteps.

B.9 Program verdict: PV.Ride.verdictThe_is_theDescription, PV.Program.claimOf, PV.Program.verdictOf, PV.Program.verdictOf_is_verify, PV.Program.verdictOf_eq, PV.Program.verdictOf_none, PV.Program.verdictOf_map.

B.11 Soundness of the executable form: PV.Ride.sound_of_the_form_events, PV.Ride.sound_of_the_form_events_oOf, PV.Ride.soundOfDecision_form_iff, PV.Ride.sound_of_the_form_at, Run.size_oOf.

B.13 Lists sent at once: VerifierAndGoals.AtTheName.soundManyProofs_of_verify_at, VerifierAndGoals.AtTheName.sessionAugmented_of_verify_at.

B.14 Evaluation binding: VerifierAndGoals.AtTheName.noTwoAnswers_of_verify_at.

B.16 Soundness with any parser: VerifierAndGoals.AtTheName.sound_of_verify_parsers_at, VerifierAndGoals.AtTheName.soundUnderPolicy_of_verify_at.

B.19 Soundness after an honest commitment: VerifiableInference.SoundAfterHonestCommitmentFor.

B.20 Composition: VerifiableInference.SoundForWith.soundAfterHonestCommitmentFor, Bind.Cpp.soundAtCommitted.

B.21 Implications: VerifiableInference.BindsWith.noTwoOpenings, VerifiableInference.BindsWith.extractsDrawn.

B.22 Proof outlines for the commitment: Bind.Cpp.tieAt, Bind.Cpp.bindsWith_of_tieAt, FlashProve.bindsWith_plain_law, VerifiableInference.OnPublicFiles.told_binds_written, VerifiableInference.OnPublicFiles.told_bindsWith_on, Bind.Cpp.bindsAtZ_cpp, Bind.Cpp.soundAtCommitted.

B.23 Constants of the errors: Bind.Cpp.honest_count, Bind.Cpp.commit_count, Bind.Cpp.soundAtCommitted_square.

B.24 Two degenerate committers: Bind.Cpp.blindTwin_fails, Bind.Cpp.blindTwin, Bind.Cpp.hollow_meets_join, Bind.Cpp.commit_D_0, Bind.Cpp.sample_D_0.

B.25 What the two committers show: Bind.Cpp.hollow_meets_join_never, Bind.Cpp.blind_extracts, Bind.Cpp.blind_twoOpenings, Bind.Cpp.blindTwin_exact_join_and_complete.

Appendix C

C.2 Disclosed view, honest-verifier simulator: Iop.Honest, Iop.DView, Iop.InteractiveOracleProof.dview, Iop.Honest.output, Iop.Honest.withheld, Iop.Honest.HVZK.

C.3 Table system: Iop.HolographicProtocol, AtPublicFiles.AtFiles.objOwn, AtPublicFiles.AtFiles.FullSizeMatrix.

C.4 Four conditions on the tables: Iop.HolographicProtocol.ReadsR, Hiop.Seeded.ReadsG, AtPublicFiles.Premises.NoProverSide, OnBytes.WholeKey.PremW.

C.5 Table experiment: Hiop.Seeded.EnvS, Hiop.Seeded.HostS.view, Hiop.Seeded.realHostS, AtPublicFiles.AtFiles.objOwn.

C.6 Within the opening budget: Hiop.BoundsR.Fits, Hiop.Seeded.EnvS.Adm.

C.8 Columns are uniform: AtPublicFiles.Table.table_law, AtPublicFiles.Table.strong_codingHonest.

C.9 Pads mask the answers: AtPublicFiles.Table.answer_bijective, AtPublicFiles.Table.masked_cells, Hiop.rec_answers_uniform, AtPublicFiles.Table.rowsOf_laws.

C.10 One table: AtPublicFiles.Reads.hidE_filesAt, AtPublicFiles.Table.hides_full, Hiop.rec_hides_of_rows, Hiop.rec_hides_law.

C.11 Column counts: AtPublicFiles.Reads.reads_filesAt_scope.

C.12 Tables and runs before compilation: Hiop.Seeded.whole_all_seeds, Hiop.Seeded.form_laws, Hiop.BoundsR.err.

C.13 Table game: Hiop.Seeded.gameReal, Hiop.Seeded.gameSim, Hiop.Seeded.Kit, Hiop.Seeded.runAt, Hiop.BoundsR.Fits.

C.14 Tables and runs in the table game: Hiop.Seeded.lifeSeededAt, Hiop.Seeded.errN, Hiop.Seeded.SimsG.GoodAt.

C.18 Compiled run of a prover: OnBytes.WholeKey.shown, OnBytes.WholeKey.HonestSide, TypedOracleCompiler.Compile.Batch.Cms.proveTA.

C.19 Coins of a compiled run: OnBytes.WholeKey.dress, OnBytes.WholeKey.RunLaw, OnBytes.WholeKey.SaltsLaw, OnBytes.WholeKey.CoinNums.

C.20 Whole-key argument: OnBytes.WholeKey.wholeKeySchemeOverTypedOracles, OnBytes.WholeKey.argW, OnBytes.WholeKey.commitAt, OnBytes.WholeKey.proveW, OnBytes.WholeKey.wholeKeySchemeOverTypedOracles_V.

C.21 Five conditions on the provers: AtPublicFiles.Premises.FiveLeft, OnBytes.WholeKey.ExtEnc, OnBytes.WholeKey.Sims.Good, OnBytes.WholeKey.StopsSmall, OnBytes.WholeKey.RunVals, OnBytes.WholeKey.CoinsFit.

C.22 Eight conditions on nonces and positions: `AtPublicFiles.Premises.NoProverSide`, `OnBytes.WholeKey.ReadsFree`, `OnBytes.WholeKey.NoncesBack`, `OnBytes.WholeKey.SeedPlain`, `Iop.InteractiveOracleProof.IsLocal`.

C.23 Three conditions on formats: `AtPublicFiles.Premises.NoProverSide`, `Iop.RoundSplits.Fmt`, `Iop.RAnn.Coherent`, `TypedOracleCompiler.Iop.QueryLayout.OKG`, `TypedOracleCompiler.Iop.WireRB.OKG`, `TypedOracleCompiler.Compile.Batch.Cms.OKA`.

C.24 Hashing error: `Rom.costTotal`, `OnBytes.WholeKey.mW`, `OnBytes.WholeKey.eW`, `OnBytes.WholeKey.bitsW`, `OnBytes.WholeKey.osW`.

C.25 Reduction of index privacy to twenty conditions: `OnBytes.WholeKey.wholeKeyPrivacy`, `OnBytes.WholeKey.Spec.wholeKeyPrivacy`, `OnBytes.WholeKey.PremW`, `OnBytes.WholeKey.zetaW`.

C.26 The conditions that mention no prover hold: `AtPublicFiles.Premises.noProverSide_description`, `AtPublicFiles.Reads.reads_filesAt_scope`, `AtPublicFiles.Reads.hidE_filesAt`.

C.27 Bound on the error after compilation: `AtPublicFiles.LinkLaid.zeta_of_twelve`, `AtPublicFiles.ZetaAt.zetaW_le_caps`, `AtPublicFiles.ZetaAt.twelve_description`, `AtPublicFiles.ZetaAt.boundZ`.

C.28 Constant-message provers satisfy the five conditions: `AtPublicFiles.Premises.fiveLeft_fixed`, `AtPublicFiles.Premises.hasSilentMemberC_description`, `OnBytes.WholeKey.silent_good`, `OnBytes.WholeKey.extEnc_silent`.

C.29 Committed tables and link proofs after compilation: `AtPublicFiles.LinkLaid.linkLaidWithBound`, `AtPublicFiles.LinkLaid.linkLaidEight`, `AtPublicFiles.LinkLaid.premW_laid`, `AtPublicFiles.LinkLaid.linkLaidAtOnce`, `AtPublicFiles.LinkLaid.bitsE_listed`.

C.30 Reach of the corollary: `AtPublicFiles.LinkLaid.laid_hashed_accepted`, `AtPublicFiles.LinkLaid.laid_accepted`.

Appendix D

D.1 The default configuration, encoding: `VerifierAndGoals.storedPublicFiles`, `VerifierAndGoals.policyFile`, `VerifierAndGoals.FreeLines`, `Inst.Model.KLFile`, `OnBytes.Flat.klFileOf`.

D.3 The integer model: `IntegerTransformer.model`, `IntegerTransformer.trace`, `VerifiableInference.hasModelSizes`, `VerifiableInference.lookupTables`, `VerifiableInference.weightsAreValid`, `VerifiableInference.ModelWeights`, `Policy.modelLogits`.

D.6 Sub-protocols: `VerifiableInference.Desc.KindName`, `VerifiableInference.Desc.Block`, `TypedOracleCompiler.Run.PQ.roundS`, `TypedOracleCompiler.Iop.GrW`.

D.7 Test on a committed table: `Desc.K14.rounds`, `Desc.K14.readsAtLoc`, `VerifiableInference.Desc.KindName.K14`.

D.8 Two rounds at the default configuration: `OnBytes.Params.record_radius_inJohnson`, `Cb.Gl.Bld.J.cnt_32_446693374_2852142980_132`, `Desc.K14.Law.bad_le_efReal`, `Desc.K14.Law.real_test64`, `Desc.K14.Law.ownPt_below`, `Desc.K14.Law.k14_rounds_real`, `Desc.K14.Law.teeth`.

D.9 The rounds of the three IOPs: `VerifiableInference.Desc.Key.descriptionFromSource`, `Desc.K15.linkDescV`, `ProofFile.onlineDescription`, `VerifiableInference.Desc.First.firstChain`, `VerifiableInference.Desc.KindName`.

D.11 Oracles, seeds, and the compiled verifier of one IOP: `TypedOracleCompiler.Compile.Batch.typedOracleFamily`, `Rom.OracleCoding`, `TypedOracleCompiler.Compile.Batch.OneProtocolProof`, `TypedOracleCompiler.Compile.Batch.recomputeChain`, `TypedOracleCompiler.Compile.Batch.reqCheck`, `TypedOracleCompiler.Compile.Batch.proofHasStrictFormat`, `TypedOracleCompiler.Compile.Batch.checkPrunedOpenings`, `TypedOracleCompiler.Compile.Batch.verifyOneProtocol`, `TypedOracleCompiler.Compile.Batch.Cms.outOfDomainSeedIs`.

G Further limitations

These limitations continue Section 8.1. The *judgement* and an *honest run* are defined in Definition H.1.

The implementation.

Limitation G.1 (Implementation unverified). The C++ and CUDA programs and the Python check script are tested, not formally verified, as is usual: no theorem concerns them, for soundness, weight hiding or completeness, and no security level is claimed for them. They are only partly tested, with bad results among the tests. The released programs run a scored output head under every policy; that it is the head of Definition 3.4 is tested, not proved (Limitation 8.2, Protocol match unproved), and at one setting that no admissible configuration has its cost differs (`docs/limits-all-rows.tsv`, SO-6.3.1.1). Between Theorem 5.1 and the C++ verifier's answer on a real file lie several steps, none closed by a theorem (`lean/GAPS.md`, `docs/limits.md`).

Remark G.2 (Stricter counts). For the IOPs of Construction 4.13, at every admissible Z with $\mathsf{Chk}(Z) = 1$, the error 2^{-128} of the Range test's round is machine-checked (Proposition 5.14). The figures that the C++ verifier prints for single rounds follow another accounting, which no theorem relates to these IOPs (Limitation 8.2, Protocol match unproved).

In that accounting the Range test's round, which has no proof of work, and the positions in a link proof use a smaller error than a stricter count would give, and a batch's shared rounds are counted once. That the smaller errors suffice there is not proved; which count of an own round applies to a single request is not known. By a union bound over all columns of the matrix of cells that is tested, not of Θ, the Range test's round would have 113.44 bits at the default configuration, calculated and taken from no run, and so miss the target of 128 (Table 15); with a factor for the number of requests, some shared rounds miss it too.

Table 15: Figures in bits. *Programs*: the programs' own accounting at the default configuration, calculated and taken from no run, and no security level. 113.44 and "6 fewer" are printed by the released verifier's planning call. "About 122.5", with the rounds' proof of work counted, is reckoned by hand from a list of rounds stored in the repository, which the released verifier's planning call prints again line for line (the repository's test `verifier.kit_rounds`). No figure is a level of the released programs.

Figure	Source	What it counts
113.44	programs	the Range test's round, by a union bound over all columns of the tested matrix
about 122.2	this paper	the Range test's round, $128 - \log_2 54$, if a factor for a list of 54 codewords were owed
about 122.5	programs	25 rounds that sample a segment's positions, with their proof of work counted and with that factor
6 fewer	programs	an own round, if it is counted jointly for 64 requests

The relation.

Limitation G.3 (Greedy outputs and a fixed window only). Sampled outputs are not covered. By design a proof covers token IDs, not text, in a window whose length the configuration fixes, 1,024 tokens at the default configuration: every request fills a whole window and costs a full window of proving. An output cut short verifies as a shorter output, and under a clipped-gap policy a provider can still end one early (`docs/verifying.md`).

Limitation G.4 (Prefixes count). Nothing is stated about where the model would stop: any prefix of its output is a greedy output, so the client must check that the output is as long as expected, since Theorem 5.1 states nothing on length. The empty prefix is one too. By step 4 of Construction 4.15, V_Z rejects a request whose claimed output is empty, since build is undefined there (Definition 4.7); no theorem states this by itself; Serv_μ leaves such requests out (Definition 3.11). Every non-empty prefix of the model's own output meets every admissible policy.

Security model.

Limitation G.5 (Fiat–Shamir instantiation). That BLAKE2b, on which the programs build the hash function that Theorem 5.1 samples, may be treated as a random oracle, sampled after the configuration is fixed, is assumed; no proof can close this, as is usual. The bound states nothing of the concrete functions. One constant beneath the Lean verifier is a value of the programs' own hash function; that nothing beneath the verifier reads it rests on a reading of the code, not on a theorem (`lean/THEOREM.md`). Fiat–Shamir attacks on protocols of this kind have not been assessed, among them attacks on GKR-style protocols that use a concrete hash function [KRS25] and on protocols whose instances a program generates [Fen26] (`docs/limits.md`, `lean/LIMITS.md`).

Limitation G.6 (Classical adversaries). As is usual, in all machine-checked statements the committer and the prover query the oracle classically, so nothing is claimed against a quantum adversary.

Evidence.

Limitation G.7 (No independent review). The tests, runs, comparisons and readings were made and reviewed within this project, so reviews are not independent, and no written proof is verified or independently reviewed. The technical report's written proofs for soundness and for zero knowledge are not settled, and readings of them within this project have found errors (Appendix E.1; `docs/limits.md`). Each count, outcome, figure or condition of Section 7 and Appendix H is its producers' account unless another checker is named. To our knowledge no person has checked the written proofs, the code, or the logs of any run; of the Lean statements, six definitions of properties were shaped with a person, which is not a check of them (Section "Who wrote and checked this"); the Lean statements apart from the six top theorems and the final statement's six fields no person has checked, and the definitions that lie beneath these other statements alone no person has read in full; Lean's kernel checked the proofs.

Limitation G.8 (Few runs). The figures of Table 3 rest on three runs each, each row's made close together in time, whose rates differ, row by row, by 1.5 to 1.6 % in the three rows at exact match and by 1.5 to 4.7 % over all six (the most over the least), with no cause shown. The first row's ± 0.92 is the standard deviation of three runs made side by side: three runs of the same kind on other programs, on another day on other machines, had a standard deviation of 3.9; whether the programs, the machines, the day or the rule of the steady rate made that difference is not shown. Each row's spread reflects the machine and the seed, that of the like three runs also the order of the kinds of run on

a machine; it does not cover other prompts, days, kinds of machine or programs. Many measurements not on the released programs, and most counts of tests, are single runs (`docs/limits.md`).

Limitation G.9 (Quality unmeasured). The model of the theorems is the integer model (Definition 3.1), a program in integer arithmetic with the shape of Llama 4 Maverick, not the floating-point model; how far the two differ is stated by no theorem. That the released programs prove this same program is tested where compared, not proved (Remark D.4, Model specification); the model is defined at some inputs, weights and policies for which they make no proof. Its task quality has not been measured: no task benchmark was run (`docs/limits.md`).

Visible by design.

Remark G.10 (What a client of the programs learns). By design, clients and the public still see some things. From an honest, fault-free run of the implementation, with the provider's operating conditions kept and a randomly sampled secret seed, a client learns at least:

1. whether each request was served or failed;
2. the architecture, shape, ranges and fitted values;
3. each next token;
4. batch membership and timing;
5. failures and stops;
6. roughly how much was served;
7. the OOD answers.

The leakage (Definition 2.13) holds items 3 and 6, item 1 for served requests, and which revealed requests share a batch. Item 2 is part of Z. Failed requests, timing and item 5 are outside the experiments. Item 7 is for the simulator to produce: an OOD answer is computed from the weights and a pad, and with a fresh uniform pad, at a seed chosen before the table is sampled, its distribution does not depend on the weights (Theorem 6.3). No statement covers the OOD answers as the programs make them (Remark 6.4, No prover message is covered).

That nothing else is visible is not established: the release's longer list (its page "What is not hidden") is not shown to be complete, or to match the released programs.

Limitation G.11 (Outcome leakage). By design a client learns whether each request is served or fails, and nothing bounds what many answers reveal together. Whether an instance is served, that is, whether (M, x, y) lies in Serv_μ (Definition 3.11), depends on the weights and is not hidden: in the results on the Lean provers the simulator is given it, one bit per request, before compilation and in the statement at every request after it (Theorem 6.37; in a corollary, within one label per request: served, failed or other tokens), and where a request is not served the prover shows nothing before compilation and returns the empty proof after it; in the statement on served requests (Theorem 6.35) it is given the requests and beside these nothing of the weights, and only served requests are asked (Section 6.5). A bit says whether the claimed output is the model's result on that request, so over many requests the bits tell what the model answers. That a request is refused is unchecked, as the label is: an empty proof proves nothing, and a prover can return it at a request that it could serve; that a service says "failed" rightly is not proved either. For the implementation one bit is not a cap: no statement bounds how much is revealed, and rarer causes of a failure can also depend on hidden values (`docs/what-is-not-hidden.md`).

Operating the released programs: eleven limitations.

Limitation G.12 (Further commitments). The judgement of Section 7.5 takes in a further commitment to the same weights, which renewal or a second GPU requires, on thin ground. Among its five conditions: each commitment must sample its own secret seed, on a machine that was not cloned or restored, take nothing from another, and keep the same configuration files (all five: Appendix H.5). No test has searched two commitments together for leaks. With hidden weights, nothing that a client can check shows that two commitments fix the same weights (`docs/limits.md`).

Limitation G.13 (Mask faults unnoticed). The first of the two main risks to weight hiding in the programs is a mask silently lost to a bug or a hardware fault; the second is Limitation 8.4 (Opening budget). A lost mask can go unnoticed: a file with an unmasked value can still verify, and no verifier can see it. A hardware fault or a compiler fault lies outside an honest run (Definition H.1), and so outside the judgement of Section 7.5; a bug in the sources lies inside both. Four compiler faults are known, at least three of which can act silently. One of them was seen to give wrong values in the test of the prover's own product routine, never in a mask or a file, though no build looks for a wrong value there. It showed under an older version of the GPU compiler than the build requires, which built the GPU

programs of many of the tests and runs that the release's pages report as not on the released programs; whether the required version has it is not known (`docs/how-hiding-is-checked.md`, `docs/limits.md`). No one has looked for a way for a client to cause such a fault.

Limitation G.14 (Published values are permanent). One lost mask can give away at least one exact equation in hidden values, in a published file, to everyone and for good. Without its mask an OOD answer gives one exact linear equation on a whole row of weights, permanently public, and so does every OOD answer if the secret seed gets out at any time.

Remark G.15 (Pseudorandom masks). All machine-checked statements of Section 6 suppose fresh randomness. The released programs derive their masks and other random values, the pads of the OOD answers among them, from a secret seed by a keyed function, the cipher ChaCha20 [RFC8439], which no machine-checked statement covers. For them index privacy is computational at best (Limitation G.14, Published values are permanent).

Limitation G.16 (Operating conditions). Weight hiding in the programs rests on operating conditions of the provider, most of which no program can check. Meeting them does not by itself show that the weights are hidden, and breaking one can leak weights permanently and silently. All of them apply: the release's thirteen conditions and their further requirements. One of them is a check that the provider must make on every batch. The prover checks none of its proofs with the verifier, and a proof spoiled by a bug or a hardware fault can fail to verify and hold unmasked values, by readings of the code; so no response may leave the provider before the C++ verifier has accepted every proof of its batch. Of what ships, by readings of the code, the middleware of the vLLM integration, from which clients fetch their responses, hands on no response that has a proof before the C++ verifier has accepted every proof of its batch; `driver/measure_rate.py` reports no rate, and an example script shows no answer of the model's, before the C++ verifier has accepted the proofs; a front end of a provider's own must make that check too. At a rejection the middleware overwrites the batch's proofs, answers its requests as failed, and stops handing out proofs under that batch's acceptance policy until the operator lifts this hold; where a batch cannot be judged, it stops likewise, at the second try in a row or at the tenth at the latest, and deletes nothing. That check misses a lost mask in a proof that verifies (Limitation G.13). Narrow exceptions to what the middleware holds back and to what it does at a rejection, none open to a client, are told in `integrations/LIMITS.md`. Nothing is hidden from anyone who can read the provider's disks, memory or GPU, and the shipped vLLM integration breaks at least one condition; in it vLLM and the prover run under one account, which can read and change the weight files, the secret key directory and the request directory (`docs/limits.md`). By a reading of the code, whoever can write the secret key directory, or name a directory of their own as the directory of weights, may learn about the weights from how the prover's starts end; the commitment covers no line of the text file in the secret key directory, the line on the weights among them.

Limitation G.17 (Tensors of eight or sixteen bits). Nothing stated here on weight hiding in the programs, the judgement of Section 7.5 included, covers a weight file in which any tensor is stored in 8 or 16 bits. The programs accept such files without a warning; it is for the provider to know how each tensor is stored. The examples' public 4-bit conversion has no such tensor (`docs/limits.md`).

Limitation G.18 (One trial with such a file). No test of hiding on the released programs has used a weight file with a tensor of 8 or 16 bits (Limitation G.17, Tensors of eight or sixteen bits); in the one trial not on the released programs, a party that could write such a file but not read it learned the magnitudes of two weights of their choice (`docs/limits.md`).

Limitation G.19 (Overlapping batches). By default, by a reading of the code, the prover takes a batch while one other, which it has proved, has not been released. In the vLLM integration's middleware every stored response waits until somebody asks for its batch, up to 1,000 responses by default; a rejection there withholds one batch, and a hold (Limitation G.16) keeps back all of them and deletes none. No proved statement covers batches that overlap in time, so none reaches a request that is sent again to the service, at once or later. No attack through the overlap is known; the only search for one was a brief reading of the code, with nothing run (`docs/limits.md`).

Limitation G.20 (Other requests in a batch). For the programs, hiding a batch's other requests is a design goal: not proved, never compared in a run of the prover, and outside the judgement of Section 7.5, which concerns the weights. Any client can submit a request that cannot be proved; for the implementation no statement on hiding then covers the batch's other proofs (`docs/what-is-not-hidden.md`, `docs/limits.md`).

Limitation G.21 (Clipped gap reveals logits). Under a clipped-gap policy, which ordinary serving uses, each answer also reveals by design something about the model's logits, and so about its weights, even if the masks hold; no statement bounds how much. Many requests can measure hidden logit gaps, up to the clip: outputs cut one token apart at a limit are such questions. Nothing bounds what many answers, time or a stop reveal (`docs/what-is-not-hidden.md`).

Limitation G.22 (Unmarked public-seed mode). A commitment does not say in which mode it was made: the release's mode for a public seed in place of the secret seed, in which nothing is hidden, neither the weights nor a batch's other requests, leaves no mark in what is published, so neither a client nor the verifier can detect it (`docs/what-is-not-hidden.md`).

Limitation G.23 (Second verifier text). The Lean library holds the text of a second verifier, written after the C++ verifier's source text; the release does not build it as a program. It is not V_Z. Definitions from that text also stand in V_Z's own term (`lean/GAPS.md`, gap 58); the second verifier as a whole does not. Some runs that this paper tells set it, built as a program, beside the C++ verifier (Appendix H.1): the two are not independent, so their agreement adds no evidence, and they are known to differ (`lean/OnBytes/LIMITS.md`, `docs/limits.md`).

Pages of the release. Table 16 names, for each limitation, pages of the release that hold finer facts.

Table 16: Limitations and pages of the release that hold finer facts.

Limitation	Pages
Limitation 8.1 (Lean verifier only)	`docs/limits.md`
Limitation 8.2 (Protocol match unproved)	`lean/GAPS.md`
Limitation 8.3 (Zero knowledge of definitions in Lean)	`docs/limits.md`, `docs/what-is-not-hidden.md`
Limitation 8.4 (Opening budget)	`docs/limits.md`, `docs/limits-all-rows.tsv`, `docs/protocol.md`
Limitation 8.5 (Provider-chosen configuration)	`docs/verifying.md`, `docs/limits-all-rows.tsv`, `lean/GAPS.md`
Limitation 8.6 (Scales can disable blocks)	`lean/GAPS.md`, `docs/limits-further.md`, `docs/limits-all-rows.tsv`
Limitation 8.7 (Unattributed commitment)	`docs/limits.md`, `docs/limits-further.md`
Limitation 8.8 (Clipped gap is weak)	`docs/limits.md`, `docs/limits-all-rows.tsv`, `integrations/vllm/integer_model/OPERATING.md`, `lean/LIMITS.md`
Limitation 8.9 (Completeness on served requests only)	`lean/THEOREM.md`, `docs/limits.md`
Limitation 8.10 (Released commitment)	`docs/limits.md`, `docs/limits-all-rows.tsv`, `guide/what-is-proved.md`
Limitation G.1 (Implementation unverified)	`lean/GAPS.md`, `docs/limits.md`, `docs/verifying.md`, `lean/TERMS.md`
Limitation G.3 (Greedy outputs and a fixed window only)	`docs/verifying.md`
Limitation G.4 (Prefixes count)	`lean/THEOREM.md`, `lean/GAPS.md`
Limitation G.5 (Fiat–Shamir instantiation)	`docs/limits.md`, `lean/LIMITS.md`
Limitation G.6 (Classical adversaries)	`docs/limits.md`
Limitation G.7 (No independent review)	`docs/limits.md`, `lean/CHECK-RESULTS.md`, `docs/limits-all-rows.tsv`, `docs/figures.tsv`
Limitation G.8 (Few runs)	`docs/limits.md`, `docs/limits-all-rows.tsv`, `integrations/LIMITS.md`
Limitation G.9 (Quality unmeasured)	`docs/limits.md`, `docs/results.md`
Limitation G.11 (Outcome leakage)	`docs/what-is-not-hidden.md`
Limitation G.12 (Further commitments)	`docs/limits.md`
Limitation G.13 (Mask faults unnoticed)	`docs/how-hiding-is-checked.md`, `docs/limits.md`, `docs/limits-all-rows.tsv`, `integrations/LIMITS.md`
Limitation G.14 (Published values are permanent)	`docs/limits.md`
Limitation G.16 (Operating conditions)	`docs/limits.md`, `docs/operators-duties.md`, `docs/what-is-not-hidden.md`, `integrations/LIMITS.md`
Limitation G.17 (Tensors of eight or sixteen bits)	`docs/limits.md`
Limitation G.18 (One trial with such a file)	`docs/limits.md`
Limitation G.19 (Overlapping batches)	`docs/limits.md`, `docs/limits-all-rows.tsv`
Limitation G.20 (Other requests in a batch)	`docs/what-is-not-hidden.md`, `docs/limits.md`
Limitation G.21 (Clipped gap reveals logits)	`docs/what-is-not-hidden.md`
Limitation G.22 (Unmarked public-seed mode)	`docs/what-is-not-hidden.md`
Limitation G.23 (Second verifier text)	`lean/OnBytes/LIMITS.md`, `docs/limits.md`
"Who wrote and checked this"	`lean/README.md`, `lean/CREDITS.md`, `lean/LIMITS.md`, `docs/limits.md`

H The artifact and the evaluation in detail

Under the name of each table of Section 7, this appendix gives the conditions, the provenance and the limits of its results.

H.1 Artifact and programs

In short. The runs that this subsection reports as runs on the released programs were made on programs built from sources that differ from these in comments, names and the wording of messages alone; of the programs that these sources themselves build it reports three small runs and one at the default configuration's size ("Which programs ran", below); the sources of `cp_verify` do not differ. In the test runs that cover all 173 public tests, most of which start none of the three programs, 170 passed, one passed in part, two were not run and none failed.

On synthetic weights, the released `cp_verify` and the Lean verifier were given the same sets of files at the default configuration's size, as reported: both accepted all 47 honest proofs, and neither accepted one of the 57 proof files from malicious provers; at 3 of the 57 the Lean command's parser refused the file, which is not a verdict. The Lean verifier also accepted 41 further proofs, on synthetic weights, as reported of the timed runs. One proof made at the default configuration's size, on synthetic weights, by the programs that these sources build, as reported, was accepted by `cp_verify` and by the Lean verifier.

`examples/maverick/run.sh` ran at the default configuration's size on the public weights, without building, downloading or installing: its five stored values came out the same, byte for byte, and `cp_verify` ended with `YES`. In the timed runs on those weights, at the default configuration as fitted to them, all 1,851 requests were served, and all 1,851 proofs, each of the one token after the prompt, were accepted by the runs' own calls of `cp_verify`. The first and the fifth of the five tests of weight hiding ended as written down beforehand, the first also at the default configuration, once on synthetic weights and once, as fitted to them, on the public weights.

What is left. The 57 proof files from malicious provers were made beforehand by test programs that this repository cannot build, all at one commitment and for three requests: no malicious prover was run against the released programs, and no rejection at the lookup checks themselves has been seen. At the reduced configuration `cp_verify` accepted 228 replayed calls on altered stored tables: that is right if each table still decodes to the honest weights alone, which rests on counts of the changed columns: no decoder was run. Neither test of weight hiding shows that the masks hide, and the other three of the five were made on other programs; no run shows what the programs leave in host or GPU memory, and no audit with the provider's secrets has run on a commitment to a real model's weight files.

Two runs at 64 requests on synthetic weights failed, and no cause is shown. The tests and scripts of those test runs differ from this repository's, a few in code; of the public tests that need a GPU or the CUDA compiler only two have been run on this repository's own files. The programs that ran and the runs' records are not in this repository, so a reader cannot check how they are tied to these sources. Nothing in this subsection is a proof about the compiled programs.

Which programs ran. The version of the release is the one line of the file `VERSION` at the repository's top; this PDF ships with it.

The release contains no model weights and no proof about the public weights.

The *released programs* are `cp_commit`, `cp_prove` and `cp_verify` (the committer, the prover and the C++ verifier) as the release's sources build them (the production build). The programs that made the runs that this paper tells as runs on the released programs, or on a test program built from these sources, were built from sources that differ from these in comments, in the names of identifiers and files, in seven quoted texts that hold such a name and in the wording of eight messages (seven that `cp_commit` or `cp_prove` prints, one that a build script prints), and in nothing else.

This paper tells a program built from those as built from these sources, a run on it as a run on the released programs or on a test program built from these sources, and a reading of those as a reading of these sources; how far that tie is shown, and where it ends, is told under "What a reader can check", below. This rule speaks of `cp_commit`, `cp_prove` and `cp_verify` and of their test programs. How the tests' own files, the serving scripts and the other scripts of those runs differ from this repository's (they differ in code too) is told under "What these runs do not show", at "The public tests".

Of the `cp_commit` and `cp_prove` that this repository's sources build, this paper tells four runs, three of them here alone: the first and the fifth test of weight hiding, once each, at the reduced configuration, on synthetic weights, each ending as written down beforehand; and a comparison with the middle build told under "What a reader can check", on synthetic weights: under one test seed the test programs of both sides gave the same files, byte for byte, at the reduced configuration, and so did both production builds of `cp_commit`, called with `-public-weights`, at the reduced configuration. The fourth made one commitment and one proof at the default configuration's size, on synthetic weights: its item under the calls of the Lean verifier, below, tells it. The counts of runs elsewhere in this paper leave these four out.

This paper tells of them no timed run, no other run at the default configuration's size, no run of `examples/maverick/run.sh` and no public test that starts one of the two. The times, speeds and peaks of memory that this paper gives as the released programs' were measured on the programs of the runs, and are told of the released programs on the comparison alone.

No program file is in this repository: a reader builds them, and no run told in this paper was made on a reader's or a provider's own build. The runs told in this paper are of these kinds, told below in turn: runs on the released programs (the timed runs are among these); runs on other programs, test programs among them; and, told apart, the calls of the Lean verifier. Rows are those of `docs/limits-all-rows.tsv`.

On the released programs is said only of a run in the list below; *in the timed runs*, of those of them that are told under that heading; *not on the released programs* (†), or *on other programs*, of every other run. *Test programs* are programs in which a fault, a test switch or a test seed is built in (those of the second and the third test of hiding bear ‡); a test program may or may not be built from the released sources. Where this paper says test programs, the reduced configuration and the default configuration, the release's pages say "on test builds", "small size" and "full size"; of a run on the public weights this paper says the default configuration's size, or that configuration as fitted to them.

What ran on the released programs. The programs that ran were built from the sources told above, and are tied to those by two digests: the verifier's build word, and the digest in each prover's build record (`cp_commit.build`, `cp_prove.build`). The verifier's build word is this repository's too; the provers' digest is not: comments, names and the wording of messages turn it. Those records are not in this repository, so a reader cannot check that tie. Where a run's record holds no digest, its programs' checksums are those of the build whose record holds both. Pages lie outside both digests, and so do nearly all tests and most scripts. Of the 469 files under the provers' digest 33 are not sources, six scripts and a Makefile among them (what a reader can check: below).

A test program leaves no build record. That the test programs of these runs came from these sources rests only on the logs of the test runs that made them, which are not in this repository; each checked a checksum of the sources before it built.

Weights are synthetic, and each run was made once, unless a line says otherwise. The list tells what ran on them. Of the public tests only some rows start one of the three: how many, and how many run test programs instead, is told under "What these runs do not show". The Lean verifier is none of them: its calls have a head of their own, below. A run that neither the list nor "The public tests, in full" names is to be taken as not made on the released programs, but for one test that is not in this repository, which by a reading of its commands ran them and passed (row RES-39), and the runs of the check by the vLLM integration's middleware told under the timed runs, and the four runs told under "Which programs ran".

- *The public tests* (`SUITE.md`: 173 rows), on one H100 a test run, or a CPU machine; reduced configurations, synthetic weights, but for ten rows that give `cp_verify` a parameter file or a stored example of the default configuration's size, also on synthetic weights, and a few that run a single kernel, sub-proof or matrix at a shape of the default configuration, also on synthetic data (among them `gpu.base_kernels_test`); no row runs `cp_commit` or `cp_prove` at the default configuration's size.

  On programs that each test run built from these sources, 170 of the 173 rows passed, one passed in part, two were not run and none failed. Apart from those test runs, on this repository's files alone (without `lean/`; some pages, tables, `paper/paper.pdf`, one list and licence and notice files differ), 90 of the 105 rows of the kind `cpu` passed and two passed in part; the other 13 need a set-up row that needs a GPU, and the runner's `cpu` mode leaves such rows out. Two of the rows that need a GPU or the CUDA compiler have been run on this repository's own files, once, as far as their files were compared, and passed. In full: below the list.

- *`cp_verify`*, on the four stored examples, two at the reduced configuration and two at the default configuration's size; 20 altered sets of the two at the reduced configuration, ten of each; 57 proof files from malicious provers at the default configuration's size, made by test programs that this repository cannot build, and replayed on the released `cp_verify`, beside 47 honest proofs at the default configuration's size and 5 controls; 15 bad parameter files; 13 small proofs on synthetic weights, from a test program that cheats, which are not in this repository: 2 of true claims and 11 false proofs; 2,720 kept calls (a kept call is a command line and its files, stored with the ending that other programs gave it, and replayed), among them 228 calls, at the small example's setting, on altered stored tables; planning calls at 15 configurations, of which one was read against the figures of the release's pages (12 of the 15 are in this repository), and three more planning calls told here; and 21 made-up settings, each a set of files in which one file is changed.

  All four stored examples accepted (two with `YES`, two with `WITHIN LIMITS`), and all 20 altered sets rejected. All 47 honest proofs accepted. All 57 proof files from malicious provers rejected, and all 5 controls (4 `NO: reply-differs`, 1 `NO: model-statement`). All 15 bad parameter files refused (10 `NO: parameter-range (policy)`, 4 `NO: parameter-group-shape (policy)`, 1 `NO: parameter-table-shape (policy)`). Of the 13 small proofs the 2 of true claims were accepted (`YES`) and the 11 false proofs rejected (`NO: proof-refused`), as expected.

  Of the 2,720 kept calls 2,672 ended as recorded, in exit code and last line, the 228 among them, which `cp_verify` accepted. Each of the 228 is on an altered table that, by separate programs' counts of the changed columns, lies far inside the distance at which it decodes to the honest weights alone, so that the accepted request and output

are the honest run's own. That rests on those counts: no decoder was run, nothing of it is machine-checked, and BLAKE2b-256 is taken as collision-free.

The counts measure from the table of the run with nothing altered; that this table holds the honest codewords was checked at all positions under one of the four test seeds only, under which 24 of the 57 sets of files behind the 228 stand (these 57 are other files than the 57 proof files from malicious provers). In 72 others of these calls `cp_verify` passed a check of one part on a key file or a copy's file whose own table was altered; a passed check of one part is not an acceptance (row SO-15.7).

The other 48 read parameter files that the released programs refuse, and end with `NO: parameter-range (policy)`: they show nothing beyond that refusal. No call was more accepting than recorded. All 21 made-up settings were refused, the one with `answers` 2 with `NO: parameter-range (policy)`. The planning calls at the 15 configurations all ended with exit code 4 (`PLAN ONLY`): a planning call judges no proof. The three others told here ended with exit codes 1, 1 and 2, as written down beforehand; on which files they were made this paper does not tell. The one that was read against the release's pages printed the figures that they give as its own.

- *Tests of weight hiding* (Section 7.5 and Appendix H.5). The first and the fifth of the five tests (row LIM-63.2, lines 1 and 5), under exact match, on all three of the released programs, each commitment under a seed that `cp_commit` drew. Both at reduced configurations, the first three times and the fifth once. The first also at the default configuration, once on synthetic weights and once, as fitted to them, on the public weights. Both audits also under a clipped-gap policy, at the reduced configuration, on two proofs. One of the two audits, the audit of a commitment, also at the default configuration's size, in the timed runs.

  Each ended as written down beforehand. The first: all published OOD answers were the row's weighted sum plus the public factor times the row's own pad; no pad came twice, and none was 0. The fifth: both audits (checks, with the provider's secrets, of a commitment and of a kept batch) accepted the honest files, and with a false file in turn for each of three kept files the answer was no each time. The figures: row LIM-63.2, lines 1 and 5. Under the clipped-gap policy both audits accepted the honest files, and `cp_verify` accepted both proofs as within the policy's limits.

- *The example script*, `examples/maverick/run.sh`, at the default configuration's size, on the public weights; all the stages of `run.sh all`, in their order, on one H100; the build stage was skipped: the programs were built beforehand from these sources; the download stage fetched nothing: the five weight files were laid there beforehand at their checksums; and the toolkit was installed beforehand by that page's typed lines, so building, downloading and installing the tools were not exercised.

  Exit code 0. The five values in `examples/maverick/expected.txt` came out the same, byte for byte, and the commitment ID was the stored one; `cp_verify` ended with `YES`. By the run's own record, which two readers read apart, it changed no file of the repository.

- *The timed runs*, told under their own heading, below.

The public tests, in full. This part gives in full what the list's item on the public tests gives in short.

The four test runs. For 172 of the 173 rows, on programs that each test run built from these sources: 169 passed, one passed in part (`tools.confine`: of its check "a device node" the part on the character device 0:0 was not run on that machine), two were not run (`verifier.clang_build`: no `clang++` on the machine; `tools.lean_verify.full_size`, the one row that would give a proof to a Lean verifier: the Lean library was not built), none failed; all on synthetic weights, in four test runs, which ran the tests by name.

The 173rd row, `verifier.rejected_proofs`, ran in a test run of its own on a verifier built from these sources, on a CPU machine, and passed. Of the 172 rows nine start `cp_commit` or `cp_prove` in the production build and 20 others start the production `cp_verify` alone; the rest run test programs, copies, the tests' own programs or scripts, only build, or only read files. The four test runs ran in a tree that holds files that this repository does not hold, and that differs from it in some tests and scripts (described under "What these runs do not show", at "The public tests").

The rows that need a GPU or the CUDA compiler. Nearly every public row that needs a GPU or the CUDA compiler compiles the provers' sources, and so reads files that differ as "Which programs ran" describes; beyond that: `gpu.forest_lookup_test` and `build.forest_lookup_test` (their test, in code), `gpu.coins_own_gpu_test` and `build.coins_own_gpu_test` (one printed message of their test), `gpu.quick_start` (the page that it reads), and `gpu.serving_first_run` and `gpu.coin_contexts.services.not_full` (they run `driver/spool.py`, in which comments and docstrings alone differ); each of them ran, in the four test runs, under a runner that differs as described under "What these runs do not show", at "The public tests".

The tests' own files that differ in comments, names and printed texts alone are described there too. `build.forest_lookup_test` and `gpu.forest_lookup_test` have been run once on this repository's own files, as far as their files were compared, by their rows' own commands, apart from the four test runs, and passed; from which build the checking program of `gpu.forest_lookup_test` came is not shown. No other row that needs a GPU or the CUDA

compiler has been run on this repository's own files. The builds for the comparison described under "What a reader can check" were made apart from these rows.

This repository's files alone. Apart from the four test runs, most of the rows to which `SUITE.md` gives the kind `cpu` or `long` (107 of the 173: the rows that themselves need neither a GPU nor the CUDA compiler) were run on a machine with no GPU, each row confined by the runner. On this repository's files alone, without `lean/`, the runner was asked in one round for all 105 rows of the kind `cpu`: 92 were run to their end, and 90 of them passed. That round ran on files that differ from this repository's in 47 paths: 38 pages, 4 tables, `paper/paper.pdf`, the list `tools/lean-verify/native-modules.txt` and 3 licence and notice files. No source, test or script is among them.

The 90 are all the rows of the kind `cpu` except `weight_sums`, `tools.confine` and the 13 described below; `verifier.clang_build`, which the four test runs could not run, is one of them. `weight_sums` and `tools.confine` passed in part. Of `weight_sums` 13 steps were skipped without two Python packages; run with both installed it passed, with no step skipped. Of `tools.confine` one check was not run, "as root, another account's file in /dev/shm", since the test already ran inside a row that the runner had confined as root. Sixteen rows were also run on files that differ from this repository's in 30 paths, of the same kinds, and all passed; `integer_model.cpu` (below) is one of them. Which rows read a path that differs has not been traced. Neither the round nor these sixteen ran on this repository's files as they are.

Rows left out or not run. On this repository's files alone the runner left out 13 rows of the kind `cpu` whose name begins `verifier.` (those that need `setup.target.other_parameters_worlds` or its `.work`): each needs a set-up row that needs the CUDA compiler and a GPU. The rows that were run to their end and these 13 make up all 105 rows of the kind `cpu`. Of the two rows of the kind `long`, `integer_model.cpu` passed on this repository's files alone, in the environment that `SUITE.md` names for it, and `tools.lean_verify.full_size` was not run: that tree held no `lean/`. Its command, run once apart from the test runner, is described below, with the calls of the Lean verifier.

A row that can fail on a busy machine. `driver.measure_rate` has failed twice, once on this repository's files alone and once in another tree with the same `driver/`, each time on a machine under load: one of its timing tests found no sample from its stand-in for `nvidia-smi`. Run again it passed both times, and in the round described above it passed at once. That the load is the cause is not shown.

What no row runs. No row runs `tools/accepts/accepts.sh`, the real `tools/lean-verify/LeanVerifyWalk.lean` (`tools.lean_verify.script` runs a stand-in for it), `integrations/vllm/integer_model/tests/lean_vectors/run.py`, or `make_vectors.py` and `CompareVectors.lean` beside it, whose runs the integer-model package's pages describe.

What these runs do not show.

- *The public tests.* Most rows run the tests' own programs or scripts, or only build, and start none of the three released programs. By a reading of the commands of the 172 rows, not a trace, nine rows start `cp_commit` or `cp_prove` in the production build, and 32 others start `cp_verify`: ten of these a copy of it that the folder of a test program's build holds, and two a sanitizer build of it.

  Of the public tests eight rows run `cp_commit` or `cp_prove` as test programs built from these sources. Those are not the released programs, and such rows are told as of test programs. Two of the eight are also among the nine. Thirteen further rows read or judge files that two of these eight made. In each GPU test run the two digests were found among the records of its production and verifier builds; test programs are tied to no digest. One production build is tied to none: `gpu.quick_start` builds its own programs in a copy of the tree.

  Each test run ran named rows (`--only`), so the commands that `SUITE.md` prints were not used. The tree that the rows ran in holds 473 files that this repository does not hold, nearly all of them tests, and no `lean/`. Of this repository's 1,218 files outside `lean/`, `paper/paper.pdf` apart, 880 stand the same, byte for byte, in that tree.

  The others are 97 pages and tables, three of them tables that tests read; nine files that stand at another path; 129 files of the provers' sources, which differ as "Which programs ran" tells; 12 files of the provers' tests, in comments, names and printed texts as reported, and one of them, `prover/tests/forest_lookup_test.cu`, in code; 52 other tests; the runner, `run_suite.py`, in a comment; eight files of the serving scripts and their tests; one file of `tools/lean-verify/`, in which a comment alone differs; one list, `tools/lean-verify/native-modules.txt`, which names the Lean modules that the command runs compiled: it was drawn anew; of the rows that were run none takes a module's name from it (`tools.lean_verify.script` lays a stand-in list of its own); one script each of `tools/accepts/` and `examples/rejected-proofs/`; eight tests and four sources of the integer-model package; four files under `docs/format/` and `tools/lean-verify/`, in comments, docstrings and two printed headings; three licence and notice files, one of which that tree lacks; and seven others: three `figures.txt`, `examples/maverick/run.sh`, `examples/quickstart-gpu.sh`, `scales/make_calibration_set.py` and `toy/mask_rules.py`.

  By a comparison of tokens, of the 52 other tests 35 differ in comments and docstrings alone, one in a name too, 11 in string literals too, two are text or data, and three differ in code (`integrations/tests/test_ask_and_check.py`,

`test_end_to_end.py` and `test_spool.py` beside it); of the eight files of the serving scripts and their tests two differ in comments and docstrings alone and six in code (`integrations/proof_middleware.py` and five tests or helpers; the runs of the middleware's check, told under the timed runs, ran another text of that script, and in what it differs is told there); the scripts of `tools/accepts/` and `examples/rejected-proofs/` differ in code; of the integer-model package's files and the seven others none differs in code.

A string literal is not always only printed: some are values that a test compares, or lines that are written into a file. Where a test or a script differs in code, what passed in those test runs is the text that ran there, not this repository's. `prover/tests/forest_lookup_test.cu` is the test of the rows `build.forest_lookup_test` and `gpu.forest_lookup_test`. Apart from those test runs, `tools/accepts/`, the released script and its table of calls, by their checksums, ran at the reduced configuration alone over the released Lean files: all 4 calls of Lean ended as expected, and all 17 controls were refused. It runs single parts' checks, not the Lean verifier.

A row that reads pages read that tree's. A row that reads the stored examples read them as they lie in this repository. Three rows ran a second time in one test run, and passed both times. Two rows passed with some of their cases skipped, as their entries in `SUITE.md` allow (`integrations.unit`, `integer_model.cpu`). The two rows `setup.target.other_parameters_worlds` and `setup.target.other_parameters_worlds.work` made 17 sets of files under other parameters (the tests call them worlds); the set-up leaves 3 out. Of each row its verdict line was read, not all that it printed.

The row `gpu.frame_readings`, one of the 169, compares the prover's and the verifier's readings of the public files. In the four test runs it ran over more folders than `SUITE.md` names; over the eight that it names it passed, at reduced configurations, in a test run of its own, on another machine, on the tree that the rows ran in. At the default configuration's size, on four folders (two different configurations), on synthetic weights, it ran in a test run of its own, in a form that `SUITE.md` does not have, on one H100, and ended as expected. What failed in other runs of these tests: row T45.

- *`cp_verify`.* The kept calls say nothing of a byte that is hashed on a path that none of them takes: a change there would not show. At the kept refusals the replay is blind: those calls end just as a changed digest would end them. Of the runs on the 57 files, each run's exit status, the number of its lines on standard output and its last two lines, cut at 120 characters, were kept; a refused run prints no build word.

  Twenty-two of the 57 proof files from malicious provers were tried once each, and all of the twenty-two claim the true output. Of the 57, 24 claim a false output and 33 the true output with a lie inside, as reported. All 57 proof files from malicious provers stand at one commitment and at exact match, for three requests. The stored files were checked, not made, and which programs made them is not traced. The altered sets are honest files with a few bytes changed, not the work of a malicious prover.

  How narrow the tests with proof files from malicious provers are, and that the falsifications of those twenty-two were not aimed at the lookup checks: `guide/known-limits.md`. Of the other 35, at least three, by their makers' labels (the 24 made for false outputs have records that name no kind), aimed at a lookup (a key in no row; a count at another row) or at a digit's range; these three were rejected at the check that the proof's masked message fits the linear claim, before any check of a lookup or a range was reached, so no rejection at the lookup checks themselves has been seen (as reported; row T1.1).

- *Tests of weight hiding.* Neither test shows that the masks hide (each test's bounds: row LIM-63.2, lines 1 and 5). No run or comparison told in this subsection shows what the programs leave in host or GPU memory (rows H39.1 and HW-748). On the released programs, of the audits with the provider's secrets, the audit of a commitment alone ran at the default configuration's size: in the timed runs, behind each of four honest commitments, one of them under a clipped-gap policy: by the runs' own account it accepted all four. At the default configuration's size the fifth's two audits have run together only on other programs (below). An audit's yes to honest files is no negative test. At the reduced configuration the false files stood for kept files: no altered public file was put to the audits.

  With the provider's secrets no audit has run on a commitment to a real model's weight files: by a reading of the code, these programs do not let it run there (row HW-747). Under the clipped-gap policy the audits at the reduced configuration show that the masks are there as derived: that is no search for a leak; in that run the batch's two requests have one and the same output, and no false proof was put to `cp_verify` (row H13.8). Of the five tests behind the judgement on weight hiding these two ran on the released programs (`docs/limits.md`). The other three, and most of the readings behind that judgement, were made on other programs, so it reaches the released programs on less evidence, and it does not reach code that differs from the released programs'.

- *The example script.* One run, of the one example request of `examples/maverick/README.md`, on one card, with no batch. It neither built the programs nor downloaded the weights. The commitment is the published one, made with `-public-weights` and this repository's public seed: nothing under a secret seed was tried. A program typed that page's lines, and a reader's machine may differ in CPUs, memory and disk. The script that ran differs from this repository's in comments, in its usage text and in some messages, and in nothing else.

The calls of the Lean verifier. The command `lean-verify`, which evaluates the Lean verifier, is not one of the released programs (Appendix H.3; `lean/THEOREM.md`, "What has been run"). It ran over the released Lean files. It was called on proofs at the default configuration's size, which are inputs made beforehand; 109 calls, on synthetic weights: 47 honest proofs (42, as reported, made by programs built from other sources; the 2 stored examples at the default configuration's size, whose maker is not traced; 3 by test programs), 57 proof files from malicious provers, which test programs made, and 5 controls, each an honest proof beside a claim that is not its own.

All 47 honest proofs accepted, 45 at exact match and 2 under the stored clipped-gap policy, where an accept says less. None of the 57 proof files from malicious provers accepted: 54 rejected, and at 3 the command's parser refused the file before Lean was called, which is no verdict and no rejection. All 5 controls rejected.

- *`lean-verify`, 41 calls more*: proofs at the default configuration's size; 14 requests, 7 of them also among those of the 47, at 3 commitments; 38 proofs for one next token, 3 for 1,001 tokens.

  41 proofs, as reported of the timed runs, all on synthetic weights and none of them among the 42 told above, have been put to the Lean verifier, over the released Lean files; it accepted each (`lean/CHECK-RESULTS.md`).
- *`lean-verify`, nine calls more*: a small stored example beside altered parameter files.

  Each ended at the parser with no verdict, as written down beforehand.
- *The command of the public test `tools.lean_verify.full_size`, run once apart from the test runner, beside the 109 calls*: the proof of the stored example `examples/full-size/` (on synthetic weights), which is one of the 2 stored examples among the 47; over the released Lean files.

  Exit code 0 and "VERDICT: ACCEPT (exact match)" (`lean/CHECK-RESULTS.md`). Through the runner that row has never reached Lean in a run that this paper tells; whether it can end in an accept there, under the runner's own HOME and confinement, is untested (`SUITE.md` names the obstacles).
- *`cp_commit`, `cp_prove`, `cp_verify` and `lean-verify`, one behind the other*: one commitment, under a seed that `cp_commit` drew, and one proof at the default configuration's size (the second call of `cp_prove`; the first was refused for a fault in the testers' command), for 1,001 tokens, on synthetic weights, made, as reported, by the `cp_commit` and `cp_prove` that this repository's sources build; judged over the released Lean files.

  `cp_verify` ended with `YES`, and the Lean verifier accepted the proof: exit code 0 and "VERDICT: ACCEPT (exact match)", as written down beforehand. It is one proof of one request: it says nothing of other requests, of proof files from malicious provers or of a real model's weights.

What these calls do not show. These calls were made over the released Lean files. This part speaks of the 109 calls, the 41, the nine and the run of the public test's command; the call on the one proof at the default configuration's size in the item on `cp_commit`, `cp_prove`, `cp_verify` and `lean-verify` above is described at its end. The command and the parser were held by checksums before and after the calls; every file of the command and of the parser outside `lean/` that the traced calls beside them opened has this repository's checksum (none of the run's own calls was traced: `lean/GAPS.md`, 14, "Which programs (k)").

The outcomes were known beforehand, from other calls, over other Lean files, and those written down beforehand were these. The program `cp_verify` was not called in these 109 calls: the script that made the calls calls the command alone, and traces of one small call and of one call at the default configuration's size, neither of them among the 109, show no start of that program. Its source file is compiled into the parser's unit. The same 109 sets of files were also put to the released `cp_verify`, as reported: it accepted the 47 honest proofs and rejected the 57 proof files from malicious provers and the 5 controls.

The sets were paired afterwards, by the label, the policy and the first 16 hex digits of the proof file's checksum; the other files of a set were not paired. The 42 told above are, as reported, 41 of the 363 at exact match and the one under a clipped-gap policy that `guide/what-is-proved.md` tells. They were taken by rules fixed beforehand and have little variety: as reported, the 42 are of 38 requests and 2 shapes (38 proofs for one next token, 4 for 1,001 tokens), at 4 commitments.

All proofs of these calls at the default configuration's size are at one model size: each set's three configuration files are those of one of the two stored examples at that size, which part in three lines of the parameter file alone. Who made a file, of the 109 or of the 41, on which weights, by which rules it was taken, and that it is honest or from a malicious prover, is as reported: it rests only on the word of those who made and chose them. None of the 57 proof files from malicious provers and none of the 5 controls was put to the command under the clipped-gap policy.

All calls ran on one machine, beside other work. By the command's text the natively compiled part and the parser's unit are each compiled by one compiler; what these two compilers print as their versions was read off that machine after the calls. That the command and the parser read the files that the traced calls opened, and no others, is not shown. At those of the 109 calls and of the 41 more that reached Lean the command loaded the compiled files of the released Lean library.

By the word of those who made them and of those who keep them, a build of these Lean files made them from nothing, in several calls of Lake; that they were compiled from these files is Lake's own finding, not the run's. The

checksums of the library's own compiled files, taken before and after the 109 calls, the 41 more and the nine more, are the same. The compiled files of the packages that the library imports, which that build did not make, were not held by checksum at the calls.

This paper tells no comparison of two builds of these Lean files. The natively compiled part the command compiled itself, in a call of its own before the 109, but for three objects of a package that lay in the tree before that call. The natively compiled objects' checksums, taken before and after the calls, are the same; each call links the library anew from them, and that library's checksum was printed only at the call that compiled them. In full: `lean/GAPS.md`, 14.

`lean-verify` evaluates the Lean verifier, natively compiled in part: that is a run, not a proof. Lean reads each file's parsed form, made by the command's parser, which no theorem covers. The parser's compiled code had the same checksums at each of the 109 calls that reached it and at a control compiled apart; at the run of the public test's command (its item above), at another path, its checksum is another, and the same once names are struck.

The call on the one proof in the item on `cp_commit`, `cp_prove`, `cp_verify` and `lean-verify` above was held less closely. As reported, the compiled Lean files of its folder had the checksum of the build described above before and after that folder's first call, which was another call; no checksum was taken after this one. That the Lean files, the command and the parser of that folder are the released ones rests on the testers' word alone. It ran on another machine.

The timed runs. These are the runs behind the figures of Section 7.2 and Appendix H.2 that carry no mark, made on the released programs. They were made within three hours of one day, on H100 GPUs: six kinds of run at the default configuration's size, made 3 times each, each kind's three on three machines (at the default configuration, on the public weights as fitted to them: on the public weights (1) a commitment with one request, and services (2) at 24, (3) at 64, and (4) at 8 and 1 requests a batch; on synthetic weights (5) a commitment, its audit and services at 24; (6) at that configuration with a clipped-gap policy: on the public weights, sessions with every output token checked and like sessions that check none).

Each of the nine machines that served on the public files at the default configuration as fitted to them made its own commitment first; three of the nine commitments were named beforehand as the runs behind the figures. The audit in these runs is a check of a commitment with the provider's secrets. One run each was made of the making of an example under a clipped-gap policy, with an audit of its commitment, and of one service and one call at 64 on synthetic weights, both on one commitment made for them on that machine, which was not audited; those last two failed.

On synthetic weights every proof was accepted, and every altered copy and call rejected. On the public weights, in the four kinds of run at the default configuration as fitted to them, all 1,851 requests were served, and all 1,851 proofs were accepted by the runs' own calls of `cp_verify` behind the timed stretches (1,845 in the runs behind the figures; six more, one on each of six further commitments); in the three runs of the services at 24 requests a batch, the only runs on these weights with that test, all 249 altered copies and calls were rejected.

`cp_verify` rejected every one (exit code 1, a last line NO); six of the 240 fell at the test on a file's first eight bytes, three of the calls fell at the comparison of the proof's next token with the caller's and three at that of the commitment's ID, and for 234 copies and three calls the records keep only a name that covers tests of form and of substance alike (for two of the 234 a name that places the fall in a chain inside a sub-proof, and no nearer), so at which test each fell is not known.

Each of these 1,851 proofs is of the one token after the prompt, under exact match. So are the 720 proofs of the services on synthetic weights; four more there, one on each of four commitments, cover 1,001 positions each: 1,000 tokens of the window are checked and the token behind it is stated; so does the clipped-gap example's one proof, under that policy.

In the sixth kind of run all 720 requests were served and all 720 proofs accepted by the runs' own calls of `cp_verify` behind the timed stretches: the 360 of the sessions with whole outputs checked check every output token of their request, and passed within the clipped-gap policy's limits, not by exact match; their control's 360 check the last position alone, and passed by exact match. The requests of both were 48 stored windows of 1,024 tokens, some sent more than once.

The requests were, on the public weights, in the four kinds of run at the default configuration as fitted to them, by each run's own line, a set of 96 windows of 1,024 tokens, and at 24 and 64 a batch some were sent more than once; on synthetic weights 96 windows of made-up tokens (row RES-6). The altered copies are honest files with a few bytes changed, not the work of a malicious prover. The audits that were part of these runs are told under the tests of weight hiding (Appendix H.5).

Whether a difference from a figure of another run comes from the programs is not shown: such runs were on other machines. The scripts `driver/spool.py` and `integrations/spool.py` that ran differ from this repository's in comments and docstrings alone. The check by the vLLM integration's middleware also ran in runs of its own, in none of the timed runs, with the released `cp_verify`, on stored proofs at the default configuration's size on synthetic weights, as reported, called directly, with no inference engine and no HTTP server around it.

By itself, on a batch of 24, all 24 were released, which is no negative test: no false proof was put to it. Beside a serving `cp_prove`, in one run, it read stored proofs, not that prover's own (row RES-32). By the checksums that

these runs' records print, the scripts that ran differ from this repository's in comments, docstrings and the texts of messages of `integrations/proof_middleware.py` alone; for one run the constant `CHECK_THREADS` was also set to 4 (row RES-32; a run behind none of this paper's figures).

What came out badly, or otherwise than expected.

- On the released programs: 48 of the 2,720 kept calls of `cp_verify` end otherwise than recorded, all with exit code 1 and `NO: parameter-range (policy)`: they read parameter files that the released programs refuse.
- In the timed runs:
  - Two runs at 64 requests failed, both on synthetic weights: a service of five batches of 64 in its first batch, and one call of the prover on 64 requests at once. None of their 320 and 64 requests got a proof, the GPU's memory was nearly full in both (81,053 of 81,559 MiB at the largest sample, which is a reading of the samples, not a line of the program's), and no cause is shown; one run each, so no failure rate comes of it. No batch of 64 has been proved on synthetic weights: every attempt on record failed, eight in all, six of them on other programs. On the public weights such services ended well (rows T50 and RES-11).
  - On the public weights the services at 64 requests a batch came to their end with 930 MiB of the card's 81,559 MiB free at the fullest (80,629 MiB in use); in like runs on other programs: 866 MiB (row RES-11).
  - With the start-up counted, one of the three sessions with whole outputs checked, at 24 requests a batch under a clipped-gap policy, stays under 100 tokens per second (99.33; the others 103.94 and 103.79), and so does one of the three sessions of their control (98.12; 102.47 and 102.54): both on one machine, the slower in both sets. No cause is shown. Both ran at the default configuration as fitted to the public weights, but for the three numbers of the clipped-gap policy in the parameter file, on stored outputs with no inference engine; the rates with whole outputs checked, accepted under that weaker policy, cannot be set against the project's target of 100 tokens per second or the rates at exact match (row RES-12).
- On other programs†, in another set of runs that were timed, three of each of five kinds of run at the default configuration (on the public weights: as fitted to them), no run reaches 100 tokens per second with the start-up counted: 96.32, 97.21 and 96.43 on the public 4-bit weights (start-ups of 78.2 to 81.0 seconds), 97.90 ± 0.74 on synthetic weights (97.10 to 98.56); in one of the runs on the public weights other work of the project stood on the host through one service's whole start-up, and the CPU pressure rose to 15.8 % of a session's length (row RES-33).
- On other programs†, in the first test's run at the default configuration's size on the public weights, which was not made to be timed, `cp_commit` took 3,143.5 seconds, about 38 % longer than a like run on other programs (2,277.0 seconds): a test script that is not in this repository bound the run to CPUs of the other NUMA node than its GPU's. In the same test on the released programs, not made to be timed either, bound to CPUs of the GPU's own node, on another machine of the same kind, it took 2,328.2 seconds; programs and machine differ too, so whether the binding is the cause has not been tried.

  Another run like the one of 3,143.5 seconds, also on other programs, took 3,169.9 seconds, against 2,369.76 to 2,374.11 in three runs of the same programs on the public weights, on other machines (row RES-46); on which NUMA node that run's GPU sat is not known. In the run on synthetic weights, also on other programs, on CPUs of the GPU's node, it took 2,248.7 seconds (a like run on other programs: 2,261.5) (row RES-9).
- On other programs no run reached 100 tokens per second with the start-up counted, at the default configuration (on the public weights: as fitted to them) (row RES-33), two commitments took far longer than their like (row RES-9), and one proof was longer than any of the timed runs', with no cause shown (row RES-6).
- In the timed runs, and in one set of three runs on other programs, the audit with the provider's secrets took longer in each run than in any of another set of three on other programs, on other machines: a small shift upward is neither shown nor ruled out, and no cause is named (the times in the timed runs, and the range of the others: row RES-15).
- On other programs: some tests notice little of a fault in `HasherT`, the class that feeds and finishes BLAKE2b (its source: `docs/sources-notes.tsv`; what that costs the self-checks: row E2.4). Faults were planted in a like class. Of three, 12, 10 and 12 of 50 CPU rows noticed each in turn. Of three more, planted for the GPU's code alone, the comparison of host and GPU noticed all three, three tests noticed one, and two tests noticed none.
- On other programs, in other runs of public tests or of tests not in this repository: a GPU test that is not in this repository failed on a wrong count in the test itself, with all its cases agreeing, and a runner marked two such tests as failed, against wrong counts in its own table (row H1.4); one CPU row, `integrations.unit`, failed for a cause in the test machine, and passed with that cause removed.

- Among the public tests: `driver.measure_rate` has failed on a busy machine, and `weight_sums` and `tools.confine` passed in part (above).

What this paper tells of other programs. *Other programs* are programs that are not the released ones: built from sources that differ from this repository's in more than "Which programs ran" tells, and are not part of it, or from these sources with other options (test programs). What a run found on them does not carry over to the released programs; what a reading found carries over only as far as the differences were read (row HW-679). A fault found on them may be in the released programs too.

The kinds of result below were got on other programs. Most bear the mark "not on the released programs" (†) or, in Table 6, that of test programs (‡); a run told with no mark, and named neither in the list, nor in "The public tests, in full", nor among the calls of the Lean verifier, nor among the timed runs, is to be taken as one of them, but for the test of row RES-39, the runs of the middleware's check and the four runs told under "Which programs ran".

A leak of weights found there is to be taken as holding of the released programs unless a page says otherwise. The calls of the Lean verifier, and any run of `tools/accepts/`, that this subsection tells outside its list are told apart: over which Lean files each ran stands at its sentence or at the head of the Lean verifier's bullet. The judgement on weight hiding is given of the released programs: two of its five tests have run on them, the fifth at the reduced configuration alone (Section 7.5). It takes results on other programs as evidence, not as shown for the released programs; other code, changed or built otherwise, it does not reach.

Many of the tests and readings that this paper reports were not made on the released programs, but on programs whose sources differ from the released ones; some, the tests with malicious provers among them, on programs that also held test-only code; and some on programs that differ from these in more code. The compiled programs differ too; the machine code that differs is not shown equivalent and has not been read.

- Runs with malicious or faulty programs, and searches for attacks. No malicious prover was built or run against the released programs. The 57 proof files from malicious provers of the list are files that test programs made beforehand; here `cp_verify` alone ran on them. This repository cannot build such a prover: the code is not in it. It does hold the functions that such a prover would use, which, by a reading, do nothing here (that their calls leave no instruction in the host code is not shown), and test code that can inject a fault in every build but the production one (row E29.1).

  No public test builds or runs a prover under a switch that makes it malicious; some run a prover that argues for something false, with code of the test's own. `verifier.forest_twin_test` has no malicious prover. Eleven of the twelve stored false files of `examples.protocol_toy_false_files` were made by builds that this repository cannot make.
- Tests of weight hiding, but for the list's: the second and the third of the five, on test programs built from other sources, which hold the code that injects a fault; the fourth, on programs of other sources that hold test-only code; the fifth's two audits together at the default configuration's size; and the audits by value that stand beside the five. What each found, and what went unnoticed: row LIM-63.2 and `docs/limits.md`.
- Other replays of altered, crossed or spliced honest files (what they found: row SO-15.7).
- Nine planning calls at reduced configurations, eight on parameter files made up for the call and one on the unchanged files of a small example of this repository: four were admitted, three refused, and two aborted at an internal check; none read a key, a copy's file or a proof (row SO-6.3.2). The released programs refuse a file with `answers` other than 1, with `NO: parameter-range (policy)` (read in the code; run at 2).
- Every figure marked †. A † time, rate or memory peak is an estimate at most.
- Every run of the Lean verifier that this paper tells and this subsection does not, and the runs of `cp_verify` side by side with a stand-alone program built from the text of the second verifier (Limitation G.23, Second verifier text) (row L14), which this repository does not build as one.
- The vLLM integration's runs, but for the check told under the timed runs (Section 7.4; `integrations/LIMITS.md`).
- Most readings of the code: a reading is of other sources, unless its line says otherwise.

Which refusals no test in this repository shows: `SUITE.md`. Which faults of hiding no test in this repository shows: `docs/limits.md`.

What a reader can check. `cp_verify` prints a build word (`build=`), the SHA-256 of a list of checksums of at least every file of this repository that it includes. A reader can compute this repository's word from the sources, and build a `cp_verify` that prints it; the word stands in `examples/README.md`. It covers sources alone, tells neither the compiler nor the flags, and does not cover the rest of the provers' sources, and a printed word shows only what a program says of itself.

This repository's sources and the programs of the timed runs. The sources under the provers' digest were compared, token by token: but for the comments, the names, the seven quoted texts and the wording of eight messages nothing differs. The built programs were compared in two steps, by way of a middle build, production and test, of sources that differ from this repository's in the wording of those messages and in one line of comment alone. What either step leaves out, the tie leaves out; what is told below as not compared, not held or not covered holds of both. First step: the middle build was compared, by tools made for it, over objdump, readelf and cuobjdump, with the programs of the timed runs and with one test build of the sources of those runs, without being run.

The programs that the test runs built for themselves were not in the comparison. `cp_verify`, `weight_sums` and `make_model` came out the same, byte for byte. The GPU code of `cp_commit` and `cp_prove` is the same, byte for byte: every function and its constant banks; every line of PTX too, but for the names that differ. In their host code every instruction of every function is the same, at the same place and of the same length, and each address in it points at the same thing. About half of the functions differ in raw bytes (by a count that is no part of the comparison; its program was read a second time and run anew, and it noticed a planted byte), in address fields alone, since names of other lengths move the data behind them.

The data that the programs read (constants, tables, texts) is the same, but for the names that differ, where a program holds them (a kernel's or a type's name), and for texts that hold one: some printed texts begin with a routine's name, and follow it, and one names a source file. The files of the two programs have other checksums. Not compared: the note that holds the build's identifier, `.gnu.hash`, `.eh_frame_hdr`, and the relocation tables beyond their targets. Not held: up to 31 zeros of padding behind a string or table, the program and section headers, and the entry point.

Of the files whose comments differ, four are compiled into nothing: two build scripts, a settings file and a header that nothing includes; of the seven quoted texts three stand in no program: two of `prover/common/hash_tree_ref.h`, and one that only the compiler prints. Code behind a switch that is off in both builds is not covered. Bytes were planted in the production `cp_commit` and `cp_prove`, and the comparison noticed each; their test programs were compared by the same tools with no plant. A second reader derived every figure of the comparison's result anew, with a program of its own. Second step: a production and a test build of this repository's sources were compared in the same way with the middle build, without being run.

`cp_verify`, `weight_sums` and `make_model` came out the same, byte for byte. The GPU code of `cp_commit` and `cp_prove` is the same, and the container of the GPU code is the same, byte for byte. In their host code every instruction of every function is the same, and each address in it points at the same thing. About three functions in ten differ in raw bytes, in address fields alone, since texts of other lengths move the data behind them. Of the data that the programs read, the quoted texts of those messages alone differ: eight texts, of the seven messages that the programs print, one of which is two texts. The eighth message stands in a build script, in no program.

As far as a reading of these sources goes, each is only printed, and none is a key, a label that goes into a hash or a part of anything hashed. One of them is the name under which a refusal is printed, and the test that looks for that name differs in the same way. Two test files of this repository differ too at this step: `prover/hash_commitment/tests/hf_params_test.cpp`, in texts that follow the messages, and `prover/tests/forest_lookup_test.cu`, in code; they lie outside the provers' digest and are in no compared program. At this step, by a look beside the comparison, the entry point and the program headers are the same in all four pairs of programs (`cp_commit` and `cp_prove`, production and test), but for two headers of each `cp_prove`, which differ by 32 bytes, the difference in length of its read-only data.

Not held at this step: where inside the container of the GPU code a pointer of the host code points. Faults were planted at this step too, 69 in all, and each was noticed: 60 on the host and GPU sides of the production `cp_commit` and `cp_prove`, 8 at the check of the texts that differ (2 for each of the four programs; none of these turns a byte of code), and one at the count of raw bytes. At 2 controls with nothing planted the tools stayed silent. The host and GPU code of the test programs was compared by the same tools with no plant.

The result and the tools were each read a second time, by readers looking for a fault, with nothing run: they found no way for the comparison to pass wrongly, beyond what is told here as not compared or not held. Neither comparison ran anything. No reader of either comparison is an outside party. Apart from the comparison, under one test seed the test programs of both sides of the first step gave the same files, byte for byte, at the reduced and at the default configuration, on synthetic weights, and so did both production builds of `cp_commit`, called with `-public-weights`, at the reduced configuration.

At the default configuration there was no control and no second run of one side; at the reduced configuration each of six controls came out different, as a control must. Proofs of the production builds are drawn, so nothing is claimed of them by bytes. Equal bytes under one seed show that the two sides computed the same on those inputs; they are no test of weight hiding. The like run on both sides of the second step is the comparison with the middle build that "Which programs ran" tells: it was at the reduced configuration alone; two runs of one side gave equal bytes, and each of seven controls came out different.

Programs outside the comparison. Both steps cover `cp_commit` and `cp_prove`, production and test programs,

`cp_verify`, `weight_sums` and `make_model`, and no other program: not the two helper programs that the audit tool (`tools/audit/`) builds for its own use, the programs of `blocks/`, the tests' own programs or the parser of `lean-verify`. The audits with the provider's secrets are runs of the audit tool. Its 14 scripts and the 2 sources of its helpers lie under neither digest.

These 16 files, and the 118 sources that the helpers compile beside them (a list that a program drew from the `#include` lines, not a compiler's), have the same checksums in this repository as in the tree that the rows ran in; from which tree the audits' own copy of the tool came is not shown. One file that the tool imports from outside its folder, `docs/format/format_rules.py`, differs from that tree's: by a comparison of tokens, in docstrings, a comment and two string literals, both headings that it prints into a file.

What a reader cannot check. The files of the programs that ran and the runs' records, the provers' build records among them, are not in this repository, so a reader cannot check these results against them.

Two builds of the sources of the runs. Two builds of those sources were compared file by file, without running them: each, by the account of those who built them, from a fresh copy with nothing kept from another build, on one machine, at one path, with the same compilers. `cp_verify`, `weight_sums` and `make_model` came out the same, byte for byte. `cp_commit` and `cp_prove`, production and test builds alike, differ only in the name of a temporary file of the compiler, which, by a reading, holds a number that differs from build to build and stays in the program's list of symbols: a few bytes (6 and 11 in this pair; 5 and 12 in the test programs), at the same sizes. All their GPU code is the same, byte for byte.

So the files of a second build of these two have another checksum. `cp_verify`, whose sources do not differ from this repository's, was also built on a second machine (Ubuntu 24.04, g++ 13.3.0), with the repository's own script: the same file, byte for byte. A third build of those sources, in the production build, in a fresh container on a GPU node, at the same path, with nvcc 13.2.86 installed at another place, was compared with the programs of the timed runs: `cp_verify` came out the same, byte for byte; in `cp_commit` and `cp_prove` no function differs, on the host or on the GPU, by a comparison function by function, but other parts of the files do.

Each of the two differs from its counterpart in the same eleven sections of the host file, among them the container of the GPU code and the section that says which libraries to load and where to look for them, and in one note inside that container; none of these differences has been traced to its cause. That pair does not show what another build path does. Whether it was made on another machine than the programs of the timed runs, and which compilers made those, is not in the records that were compared. No test build was in that comparison. That comparison ran none of the three programs. It noticed a byte planted in a host function and bytes planted in the GPU images; the unpacking itself is not tested, and nothing was planted in the PTX.

How other sources differ. How far the sources of other programs differ from these is told with each kind (row RES-39; the head of `docs/limits.md`).

Test code in the production build. That a production build holds no test code is checked by a search for strings, which does not read code (row E17.2); at least 7 of the 93 lines of `prover/tests/serving_build_words.txt`, which it reads, stand nowhere in the sources of `prover/`, `blocks/` and `verifier/` outside their tests, so their absence shows nothing; that the search ran is shown by a build's records, not by a log. Some code that no run can reach is in these sources (rows E29 and E29.1). The readings behind that were not independent (`docs/limits.md`).

Libraries and counts. Each run used the CUDA libraries, the driver and the C and C++ libraries of its machine, and says nothing of others. Counts of lines, files, kernels and functions in this paper may be off: no such count has been made on these sources, but for those that a row tells of the released programs (rows RES-39, RES-54, H39.1 to H39.4.1, HW-213 and HW-218).

Weight hiding and soundness. What is proved, of what, and what is only tested is told on `guide/what-is-proved.md` and `guide/weight-hiding.md`. Nothing in this subsection is a proof about the compiled programs.

H.2 Cost (Table 3)

Each row gathers three runs, each job's three on three machines. Each machine that served on the public weights at the default configuration as fitted to them made its own commitment first, nine in all; three of the nine were named beforehand as the runs behind the figures. Worse figures of these quantities, not on the released programs, stand in the release's table of limits (rows RES-13, RES-19, RES-24, RES-32 to RES-34 and RES-46).

- The first row's six executions of the prover, in the timed runs, gave between 110.07 and 112.04 tokens/second; two on one machine and commitment, one directly after the other, differed by at most 0.26 % (111.64 and 111.35 tokens/second). Three runs of the same kind on other programs, on other machines on another day, gave 105.3 ± 3.9 tokens/second (100.85 to 108.06); among their six executions two such differed by 6.3 %† of the larger (111.55 and 104.56 tokens/second). No cause is shown for either; the released programs did not run on those machines, nor the other programs on the timed runs'. A gap between rows that is no larger is uninformative. Apart from the clipped-gap row and its control, which ran in pairs on one machine and commitment, the rows

ran at other times or on other machines and are not comparable. These pairs show no cost of checking whole outputs; a cost smaller than the pairs' own spread would not show.

- Of the failed call and the failed service on 64 requests in the timed runs: the cause is not shown; the largest memory sample of each was 81,053 MiB of 81,559 MiB, which is a reading of the samples, not a line of the program's.
- Not on the released programs† six commitments to the same files, each under its own sampled seed, took 2,307.44 to 2,409.48 seconds. They are none of the nine commitments of the timed runs.
- Not on the released programs†, in one run on the public weights, the largest single proof had 18,811,964 bytes, more than any proof of the timed runs, whose largest has 18,804,532 (18.81 MB, rounded up); no cause is shown (row RES-6).
- Which proof the timed verifying calls took is not recorded.
- For one commitment the provider keeps 2.15 GB (in the timed runs). While serving at 24 requests a batch, the prover's own process held at most 257.1 GB of host memory on the public weight files and 265.5 GB on synthetic weights, and its container at most 269.7 GB on the public weight files and 278.1 to 278.3 GB on synthetic weights, file cache not counted; at 64 a batch on the public files the container reached 278.3 GB; not on the released programs†, on programs whose prover runs the verifier on every response before it hands it out (the released prover runs none), a session at 24 requests a batch on the public weights reached 290.7 GB in its container, file cache not counted, and 257.6 GB in the prover's own process (largest sample of 3 runs); on synthetic weights 299.1 GB and 266.1 GB; with the middleware's check running beside a serving prover the container reached 298.4 GB, against 278.2 GB without (one run on the released programs, none of the timed runs, on synthetic weights); in the three timed runs committing held at most 286.9 GB, and its container at most 299.6 GB; in one run of the example script, not on the released programs†, host memory in use reached 309.6 GB while the committer ran. The serving job held at most 4.69 GB of working disk (in the timed runs), the weight files and the commitment's directories apart. How host memory behaves over a longer service is not known.
- Two figures of Section 1.3, from the timed runs on the public weights at the default configuration as fitted to them ($n = 3$). Each commitment had one copy: its key file and its copy's file had 6.24 MB together (6,235,160 to 6,237,760 bytes); by Definition 4.7 a further or renewed copy would have a file of its own; no run here made one. The rate with start-up counted is the 122,880 prefill tokens of five batches of 24 over the time from the program's start to the last release. That time takes in a start-up of 74.8, 73.1 and 72.6 seconds, run by run, and a first batch longer than a steady gap between releases, so the rate depends on the number of batches: the longer the service, the nearer it comes to the steady rate; in the timed runs no service of more than five batches of 24, or six of any size, was run. In the timed runs the start-up at 64 requests a batch took 76.5 to 78.9 seconds.
- By a reading of other sources, not by a run, each call on a proof also makes the full check of the key file and of the copy's file, besides the proof's own check, so 21.14 ± 0.03 seconds called with its option for one thread, on the prover's machine (n = 3) takes both in, at every call.
- Whether the CPU cores were reserved to a run is not shown: by each run's own record the job ran on a set of 24 of its host's 96 CPUs, under a hard limit of 26 CPUs, and every program printed that it was pinned to 24 CPUs of its GPU's NUMA node; beyond that the host's other seven GPUs were in use when the machine was assigned (reported), the records do not tell what else ran on the host. What other jobs on a host did during a run is recorded only in part (`docs/limits-all-rows.tsv`).

"Masking on" is a run's mode: by the runs' own summaries the committer sampled the secret seed and no test option was set, so that, by readings of other sources (Section 1.3; Appendix H.1), values are masked. No run here or in Section 7.4 tested it (Section 7.5). The release's `docs/figures.tsv` gives the conditions and further figures.

H.3 The executable form on the prover's proofs (Table 4)

For the 109 calls told at this section's end and the second set of 41, by Theorem 5.5 each decision is that of V_Z on the proof as the parser of these calls cut it out of the files; for all other calls, that of the Lean verifier that the Lean files of that call define, on the proof as the parser of that call cut it out; in both cases under the assumption that Lean's evaluator, compiler and runtime, a helper library with its C compiler, and the file that drives the evaluation are correct: the kernel checks proofs, not evaluations. Some runs evaluated the executable form directly, in a separate build of the executable form, tied to V_Z by name and statement, not by bytes; the `lean-verify` command was not used for them. That build's Lean files are not shown to be the released ones; it alone made the decisions that Table 4 marks as by a separate build of the executable form, among them its side of the 43 comparisons and of the four matched alterations. No theorem covers any of these parsers.

- This evidence is thin in kind: at exact match, the only policy of Theorem 5.1, the 363 proofs made by one prover built from other sources† (as reported: those handed over, by a rule fixed before the runs, of the 723 it made) are 360 with one output token: 96 requests of one shape from one list, each proved under the same three commitments to, as reported, one set of synthetic weights, 24 of them twice under each, and three of the first stored instance; none of the 363 was blind, the C++ verifier having accepted each first in those runs, as reported (said of whole runs, no file checked by checksum); and that this prover made them is as reported: nothing in a file names the prover.
- The eighteen proofs at exact match by other provers (12 by one set of such programs, as reported; 6 by others) are of five instances: seven of one; nine of three others, each a prompt of 1,024 tokens with one output token, under three commitments to, as reported, the same weights; two beside the malicious proofs of the true claim below, one of them of one of those three instances.

  Under the clipped gap, where an acceptance states less, the four by other provers (1 by the set that made the 12, as reported), and one by the prover of the 363†, as reported, are of one further instance, that one at a new commitment and not blind; under that policy `lean-verify` has met honest proofs alone, and a separate build of the executable form two proofs of false claims, both rejected. Four proofs by other programs come from test programs, one of them a malicious program set, as reported, to alter no value.

  Not counted here are calls of `lean-verify` that we stopped on purpose before a decision (nearly all tests of interruption; all on the first stored example's proof, which was accepted whenever a call on the same files ran to its end) and one honest proof that, as reported, a separate build of the executable form alone accepted, at exact match.
- All 26 come, as reported, from one malicious test program, each set in advance to one false claim, none from a prover built to search for a false proof that passes. The 24 share one window and commitment, and every rejection occurs at the step that reads the proof file: no evaluation shows the policy's own test rejecting anything.
- Of 22 further proof files that a malicious test prover made for the true claim (the default configuration's size, synthetic weights, two requests under the commitment of the 24; neither the prover nor the files ship, so a reader cannot repeat this), neither verifier accepted any: the released C++ verifier rejected all 22; `lean-verify`, with sources and scripts as in Section 7.1, rejected 20, at the same step (within that step, by one evaluation of a test that does not ship and two readings of the order of checks, no theorem, at a later check than the 24's; the two sets differ in other ways too, so it is not known whether that comes from the kind of lie), and gave no verdict at 2, whose files its parser refused, which is no rejection.
- A second set of 41 honest proofs, from the timed runs' own files on synthetic weights, has been put to `lean-verify`, in calls counted apart from the 109: each of three runs' files holds proofs numbered 000 to 119 and one more; taken were that one and 000, 009 and so on, to 108 in two runs (13 each) and to 099 in the third (12): $3 + 13 + 13 + 12 = 41$. It accepted each, at exact match, one call each. These are other files than the 41 of the 363 at exact match that are among the 109 calls, and than the 41 that were put both to `lean-verify` and to a C++ verifier (told below this list); of neither is one a proof of the timed runs.
- That evidence is thin too: synthetic weights; not blind; one machine; one size of model, the default configuration's; two shapes (38 with a prompt of 1,024 tokens and one output token, 3 with a prompt of 24 tokens and 1,001 output tokens); the count and that the files are those runs' are as reported; and, like the 109 calls, the calls were made over the released Lean files. The command had accepted each of them before, over Lean files that differ from the released ones. No file is among the 47 honest proofs of the 109 calls; 7 of the 14 requests are.

From the release a reader can repeat two acceptances and two rejections, though untried with Lean and its packages fetched as the release instructs (`tools/lean-verify/README.md`); of the other evaluations, those through `lean-verify` were driven by the form of `lean-verify` that each run had, with scripts as in Section 7.1, on inputs that are not in the release; the rest were driven by files that are not in the release and have not been reviewed (`lean/THEOREM.md`).

Of the files put to `lean-verify` and to a C++ verifier, two groups. (1) The form of the command marked † in Section 7.1 judged 41 files over Lean sources that differ only in comments, in 25 files, from those of the three calls that were timed (Section 7.3); it judged 9 of the 41 again over the sources of these three calls, with the same decisions, and the nine proofs of three instances over these sources only. (2) The command of other calls, over Lean files that differ in code from the released ones, compiled beforehand (that the compiled files were built from those sources is not shown), judged those 41 and the file that its parser refuses, the 22 malicious proofs of the true claim with their two honest proofs and two controls (also judged in runs marked †), and the 364 with three controls. The command `lean-verify` was called 109 times, at the default configuration's size and on synthetic weights, on files that other runs had all judged: it accepted all 47 honest proofs put to it (45 at exact match, 2 under a clipped-gap policy), 42 of them of these 364, made by that prover†, and the others the two stored proofs and proofs by test programs, as reported; of the 57 proof files from malicious provers of Section 7.6 it accepted none (54 rejected; at 3 its parser refused the file, which is no verdict); and it rejected 5 controls (`lean/THEOREM.md`, "What has been run"). These calls were made over

the released Lean files (Remark B.17: one file was added after these calls). The command and the parser were held by checksums before and after the calls; every file of the command and of the parser outside `lean/` that the traced calls beside them opened has the released file's checksum (none of the run's own calls was traced: `lean/GAPS.md`, 14, "Which programs (k)"). At those of the 109 calls that reached Lean the command loaded this library's compiled Lean files. These were made beforehand; that they are made from the released Lean files is Lake's finding at the build in several calls that Remark B.17 tells, not the run's. Their checksums, taken before and after the 109 calls, are the same. Those of the packages were not held at the calls. The command evaluates the Lean verifier natively compiled in part (Appendix H.1); that part was not made by that build. The command compiled it itself, in a call of its own before the 109 (but for three objects of a package, which lay in the tree before that call). The program `cp_verify` was not called in the reference run (the 109 calls of this section). Apart from the 109 and the second set of 41, no call of this section was made over the Lean files of the 109.

H.4 Runs by policy (Table 5)

Under neither policy has the prover run as a service beside the engine: it was started after the outputs were served. Serving a trained model with an ordinary engine and proving its outputs in one service has never run end to end; ordinary serving and proving have run end to end only in scripted demonstrations at a reduced configuration on a synthetic model. The release's `integrations/LIMITS.md` and `docs/results.md` report further runs.

H.5 Tests of weight hiding (Table 6)

Definition H.1 (Weight hiding, judgement, honest run). *Weight hiding* is the goal that clients and the public learn nothing about the weights beyond a declared list. The *judgement* of Section 7.5 is a grade that we give to a goal of the programs, from the design, what is proved in Lean for the protocol, tests and code review, on a scale of four: proved; likely holds; cannot say; does not hold. *Likely holds* means that in honest runs, with the provider's operating conditions and the judgement's own conditions kept, no way to a leak beyond the list is known, while nothing is proved for the programs. A run is *honest* if nobody has changed the provider's programs and no injected, compiler or hardware fault has occurred; a bug in the published sources is not excluded.

Outside the judgement of Section 7.5 are a real model's weights, and what a fit to them writes (a parameter file, a file of scales): only one of that judgement's five tests has used any, once on the released programs and three times on other programs, and it does not ask what they reveal.

On the released programs two of the five tests have run (the first at the reduced and at the default configuration, there once on synthetic weights and once, as fitted to them, on the public weights; the fifth once, at the reduced one), and both audits once more under a clipped-gap policy (below): every published out-of-domain answer that the first recomputed was the row's sum plus its own mask, and the shipped audits accepted the honest files. A reading of the committer's and the prover's sources, once and fast, with nothing built or run, found no leak beyond the declared list (Remark G.10, What a client of the programs learns); how many of the files so read differ in the released sources is told below. Recovering weights from published files has barely been tried, and not on the released programs. Every "stopped" was observed in a test program or, in the fourth test, in a program that held test-only code: no self-check on randomness (a program's recomputation of masked values) has been observed to report a failure in the released programs, and it is assumed that the test programs and the released programs apply the masks in the same way.

- The first and the fifth ran on the released programs, the first also at the default configuration. The second, the third and the fourth ran on test programs or on other programs alone (Section 7.5), the third and the fourth on more than one set of such programs, with the same outcomes, the bad ones too. For the released programs that judgement has less evidence than there is for the programs on which most of its tests ran (Appendix H.1). A provider's own build from the released sources comes under it only as a correct build: every build tested here is our own. The second, the third and the fourth each have one fixed set of losses; a repeat of a test is no new witness. The test programs of the second and the third do not ship.

  What was read is the sources of programs other than the released ones: those of the committer and the prover were read whole, once and fast, in a text of 384 files, without the C++ verifier's own files, which both programs of that text include. The released sources differ in code in 68 of the 384, the file with the routine that feeds BLAKE2b its blocks and finishes the hash among them, and in 136 more in comments, names and the wording of messages alone; two more of the 384 are not in the released sources; in the 68 the differences in code were read for hiding, by difference alone, apart from renamed names and reworded messages, and the files were not read whole again; that routine's released code has been read for hiding twice, by readers working apart. For many single faults the one net that the whole reading found is a verifier's run on a response before anyone else holds it; the prover makes no such run, and that reading did not take in the serving scripts (Limitation G.16).

- At the default configuration's size two of the five tests have run, the first and the fifth; of the fifth, both audits there on other programs alone, and the audit of a commitment also in the timed runs. One test of hiding has run on the public weights: the first, once on the released programs and three times on other programs[†], at the default configuration as fitted to them. In the run on the released programs, on a commitment that the committer made under a sampled secret seed, all 4,096 OOD answers of its first table and its one copy (Definition 4.6) were as prescribed, no pad occurred twice or was 0, and the separate program found all 8 faults that it planted for itself as controls. The 8 faults show that the program's count on the pads can ring, not that each of its three counts (equal pads; equal single field values; pads that are 0) could have: no fault tells them apart, and that the program counts all three is read, not tested. This does not show that the masks hide, or what becomes of a response that does not verify (the run was honest); the test reads under 2 % of the bytes of the key file and of the copy's file, and of the five tests this one alone has run on such weights.

  At that size the shipped audit accepted, in the timed runs, four honest commitments of the released programs' to synthetic weights (with the provider's secrets the audit tools accept commitments to synthetic weights alone), one of them under a clipped-gap policy, which is no negative test. On other programs both audits ran at that size too, on synthetic weights: in one run they accepted the honest files and, in trial calls, rejected four of four false secrets. On the released programs the fifth has run once under exact match, at the reduced configuration, and the first three times at reduced configurations, on synthetic weights (1,536 answers of two commitments; 512 of one; 512 of one to a small synthetic model in files of the public weights' form, at a configuration of its own, whose parameter file that judgement does not reach), and twice at the default configuration: on a commitment that their committer made to the public weights, at that configuration as fitted to them, and on one to synthetic weights. On the released programs, of the audits with the provider's secrets, the audit of a commitment alone ran at the default configuration's size, in the timed runs (Appendix H.1). On the released programs, beside the first test, in each of its runs at the default configuration (on the public weights: as fitted to them), the audit tool's program, run without the provider's secrets over copies of what is published alone (the key file, the copy's file and four proofs), reported nothing on them and ended with an error at each of three copies with one bit changed; a changed bit is not a lost mask, and this is not a test of hiding. At reduced configurations it read four honest sets of such files and reported nothing; no altered copy was put to it there. Here and in Table 6 "as prescribed" means by the rules that the test carries; these were typed mainly from the release's pages, in a text that is not shown to be the released one.
- None of the five tests shows what an OOD answer reveals or an unnoticed loss exposes; far more samples of the kind of the nine are covered by no self-check; the audits cover no served proof and do not remove the risk of a lost mask.
- That judgement reaches a clipped-gap policy (in a parameter file that differs from the default configuration's, or from that of a reduced configuration at which it is given, in that policy's three numbers alone, and at the reduced configuration at which both audits ran under such a policy: In sum, below), on less evidence: audits apart, no test of hiding ran under such a policy. Both audits ran under one: once on the released programs, at the reduced configuration, on synthetic weights, on two proofs that each cover 32 output tokens. Every random value that they compare by value (in the batch's audit just under a fifth of the bytes) was what the secret seed gives, they rejected three false secrets, and the C++ verifier accepted both proofs as within the policy's limits. By the audits' own table such a proof has no kind of record that a proof at exact match lacks, so this is the exact-match check passed on such proofs, not a check of anything that only such a policy has. It shows that the masks are there as derived, and it is not a search for a leak (`docs/limits-all-rows.tsv`). The audit of a commitment ran once more on the released programs, at the same reduced configuration and under such a policy, in the making of a small example, and accepted the honest commitment. The reach rests on the design (the released programs run one scored output head under every policy: Limitation G.1), on one reading aimed at that policy's path, made in the code of other programs and not repeated in the released code, and on these audits. Lean's theorem on zero knowledge is for exact match: under a clipped-gap policy no Lean theorem covers what a whole proof reveals (what the answers themselves tell: Limitation G.21). Also untested: batches that overlap in time, the prover's default; several clients' proofs together; out-of-range prompts.
- One finding, in which no leak was shown, as far as its account goes: the number of events in a reply streamed by vLLM (`integrations/LIMITS.md`).

In sum: that judgement is given at the default configuration and at the reduced configurations of its tests, but for that of the small synthetic model in files of the public weights' form (above). It reaches, on less evidence, a configuration that differs from one of these in a clipped-gap policy's three numbers alone, and the reduced configuration at which both audits ran under such a policy, which also has a scale file of its own. It covers no other configuration, so none fitted to a real model's weights, no weight file with a tensor of 8 or 16 bits (Limitation G.17, Tensors of eight or sixteen bits), and not the mode for a public seed, which hides nothing (Limitation G.22, Unmarked public-seed mode). A provider that serves for long enough, or on a second GPU, needs a further commitment to the same weights

(Limitation G.12, Further commitments). The judgement of Section 7.5 takes that in, on thin ground, whether the commitments follow one another or serve at once on several GPUs, on five conditions.

1. Each commitment comes from its own call of the release's committing program, on a machine that was not cloned or restored, and samples its own secret seed.
2. Each commitment takes nothing from another.
3. The configuration files stay the same.
4. Each commitment stays within its limit (Limitation 8.4, Opening budget).
5. Nothing hidden may choose which commitment serves a batch. One exception: the commitment may be picked by load or queue length. That can depend on hidden values, and the pick gives away something of time and load, which the release declares as visible.

Each commitment also meets that judgement's other conditions. So two commitments to the same weights under different configuration files, for instance under two acceptance policies, may each be inside alone, but not together.

By design, one commitment already has several tables of the same weights (Definition 4.6), each with its own randomness and OOD answers. Brief readings of parts of the host code, all of them of the code of other programs, found no way in which two commitments tell more than one; none of them covered the GPU kernels, except the two kernels that add the pads to a table and to its columns. Not on the released programs, a count over two small commitments found no derived random value twice, a check for gross faults only.

No test has searched two commitments together for leaks, and Lean has no statement on hiding for more than one commitment. So the thin ground is the design, these readings and that count.

A further commitment resets only the count of opened columns: each proved request's share of any bound on hiding, and what the served-or-failed answers give, run on across commitments. That judgement on weight hiding does not cover a commitment that is in doubt: one of which a count, a key or a directory may not be what the provider's record says (the list: `docs/operators-duties.md`). The one that replaces it may be inside alone, on conditions of its own, but of these weights as a whole that judgement says nothing. It does not cover what the one before may already have shown.

The release's `docs/what-is-not-hidden.md` gives the declared list (Definition H.1), and `docs/limits.md` the provider's operating conditions (Limitation G.16, Operating conditions), that judgement's own conditions and its grounds; `docs/how-hiding-is-checked.md` and `docs/limits.md` give further tests and bad results.

H.6 Negative tests (Table 7)

No malicious program has been built from the released sources or run against the released prover and committer. The malicious provers and committers of Table 7 are test programs, not the released ones, some of them edited for a test; they do not ship, so the rows that use them cannot be repeated from the release. No reported test uses a searching prover, or any malicious prover on real weights. Most tests are weak, at reduced sizes and on synthetic weights: evidence, not a guarantee, resting on no Lean theorem about the programs.

H.7 The integer model

On 544 of 544 windows the range checks of the integer-model package (not on the released programs) found none failing at the public weights. Besides the runs of Section 7.2 and Section 7.4, this count outside Lean is the only evidence reported here that instances of a trained model can be served (Remark D.5, No real weights shown).