

Magnetar: High-speed post-quantum RV64 zkVM from standard hashes

Srinath Setty

Microsoft Research

October 5, 2026

Abstract.

We describe the design, implementation, and experimental evaluation of Magnetar, a transparent proof system for RV64I program executions (i.e., a zkVM). Magnetar is the first zkVM to simultaneously satisfy the following three properties:

1. **General-purpose proving.** Magnetar proves programs compiled to the general-purpose RV64I ISA. An implementation of the Ethereum Virtual Machine (EVM) written in Rust or Go can therefore run inside Magnetar, yielding a zkEVM for proving executions of existing EVM smart contracts.
2. **Hash-based, post-quantum security.** Magnetar’s cryptographic security is rooted in standard hash functions (e.g., SHA-256 or SHA-3).
3. **Efficient recursion.** Magnetar’s verification work is dominated by evaluating standard hash functions. Magnetar’s proof engine, BinarySpartan, is well-suited for such bit-oriented computations, so verifying a proof is itself an efficient proving workload.

Unlike prior general-purpose zkVMs, Magnetar does not rely on circuit-friendly algebraic hash functions such as Poseidon or Poseidon2, elliptic-curve or lattice assumptions, or a conjectured Reed–Solomon proximity gap. Even prior zkVMs commonly described as hash-based rely on elliptic-curve or lattice assumptions in their core proving machinery, provide only about 100 bits of security, or rest on conjectured soundness; some do more than one of these.

Finally, we implement Magnetar in Rust and experimentally evaluate it. We find that, at 120 bits of provable security and with 12 performance cores on an M4 Max MacBook Pro, Magnetar proves 2^{22} -cycle executions at 7.2–8.7 million cycles per second, with proofs of about 600 KiB that verify in 11–13 ms. Across 15 application workloads, Magnetar’s proving throughput is $24\text{--}66\times$ that of SP1 Hypercube, a hash-and-more-based zkVM; $12\text{--}37\times$ that of Jolt-Dory, a curve-based zkVM; and $6\text{--}20\times$ that of Jolt-Akita, a lattice-based zkVM, with geometric-mean speedups of about $39\times$, $18\times$, and $9.6\times$, respectively. An optimized Magnetar proves 2^{22} -cycle executions at 9.5–12.8 million cycles per second, which increases each geometric-mean speedup by about $1.3\times$. Magnetar’s proofs are about $4.6\times$ smaller than SP1 Hypercube’s; Jolt’s proofs are under 100 KiB, but they rely on elliptic-curve or lattice assumptions.

Table of Contents

1	Introduction	3
1.1	The zkVM design space	3
1.2	Our work: Magnetar	5
1.3	Closely related work	6
2	A technical overview of Magnetar	8
2.1	End-to-end protocol	8
2.2	Key ideas in Magnetar	10
3	Preliminaries	13
3.1	Multilinear extensions	13
3.2	The sum-check protocol	13
3.3	Preprocessing SNARKs	13
3.4	Reductions of knowledge	14
3.5	R1CS	15
3.6	Polynomial commitments	16
4	Magnetar’s proof-system building blocks	16
4.1	Spartan	17
4.2	Spartan for SIMD R1CS	19
4.3	BinarySpartan	20
4.4	BinaryTwist and BinaryShout	21
5	The Magnetar construction	25
5.1	Arithmetizing CPU semantics with two SIMD R1CS relations	26
5.2	MosaicPCS	26
5.3	Virtual Spartan	28
5.4	BinaryRouter	30
5.5	The complete protocol	32
6	Implementation and evaluation	34
6.1	Experimental setup	35
6.2	Core zkVM speed on sample applications	36
6.3	Comparison with ZisK and Lineth	38
6.4	Scaling and memory	39
6.5	Scaling to larger traces	39
6.6	Optimized Magnetar	40

1 Introduction

A zero-knowledge virtual machine (zkVM) is a general-purpose proof system for program executions. Specifically, given a program and its inputs, the prover runs the program on a specified machine, such as RISC-V, and produces a succinct proof that the execution resulted in the claimed output. The program may also take private, non-deterministic inputs from the prover. The verifier checks the proof without repeating the execution.¹

The main advantage of a zkVM is general-purpose proving. A proof system ultimately requires an arithmetic description of the computation proven. One can design and optimize a circuit for a fixed application, but adding functionality or changing existing functionality requires a new circuit. A zkVM instead arithmetizes the processor once and supplies each application as machine code. Developers can write programs in Rust, Go, or C and compile them with existing toolchains; the same processor arithmetization proves all such programs. A zkVM targeting a general-purpose ISA such as RV64 provides a direct path to a zkEVM: an implementation of the Ethereum Virtual Machine (EVM) written in Rust or Go can be compiled to RV64 and run inside the zkVM, yielding proofs of existing EVM smart contracts without rewriting those contracts for a specialized ISA. This realizes, for Ethereum, the approach of replacing replicated execution with replicated verification of proofs [34].

The benefit of general-purpose proving depends on the machine that is arithmetized. A standard ISA such as RISC-V gives developers access to existing compilers, languages, and libraries. A restricted ISA (e.g., Cairo [16], leanVM’s leanISA [32]) can make it easier to build an end-to-end prover for a specific application such as post-quantum signature aggregation, but applications must target that ISA and its specialized toolchain, and work within the restrictions the machine imposes (e.g., write-once memory). General-purpose execution, however, is not the only goal. The underlying proof system also determines whether the zkVM is post-quantum, which cryptographic assumptions it makes, whether it requires a trusted setup, and whether it supports recursive proof composition [61,6,31]. Recursive proof composition allows a prover to handle executions that are too large to fit in a single machine’s memory, and it also lets a continuously evolving computation maintain a succinct proof of its execution so far, making a proof available at any point in time. Succinct blockchains [9] are a canonical example of an application that requires the zkVM to provide recursive proof composition.

This paper asks whether all of these goals can be achieved while being faster than existing zkVMs.

1.1 The zkVM design space

We use three properties to compare zkVMs.

¹ Most zkVMs provide only a succinct argument of virtual-machine execution; zero knowledge is an additional property.

1. **General-purpose proving.** Applications are written in high-level languages and compiled to a standard ISA (e.g., RV64). The programmer does not construct a circuit or target a restricted machine.
2. **Hash-based, post-quantum security.** Computational security is based on standard hashes such as SHA-256 or SHA-3, rather than elliptic curves, lattices, or algebraic hashes such as Poseidon [19] or Poseidon2 [20]. This requirement covers the entire proving machinery, not only the commitment scheme: a zkVM that commits with Merkle trees but uses an elliptic-curve or lattice-based hash elsewhere in its proving machinery does not qualify. Additionally, the system does not require a trusted setup and does not rely on a conjectured proximity gap for the underlying error-correcting code.
3. **Efficient recursion.** Verifying a proof is itself an efficient workload for the zkVM, allowing proofs of long executions to be recursively composed without constructing one monolithic proof.

Existing zkVMs occupy four broad cryptographic families.

Elliptic-curve-based zkVMs. Proof systems built from elliptic-curve commitments can produce short proofs and support efficient recursive proof composition. Folding schemes such as Nova [31] and follow-up works [27,28,41,29,10,14,30] demonstrate how recursive proof composition can scale proving to long computations. General-purpose systems can also use curve-based commitments, as Jolt [1], an RV64IMAC zkVM, does when instantiated with Dory [33]. Jolt-Dory relies on elliptic-curve assumptions that are not post-quantum and has not demonstrated end-to-end recursive composition of complete zkVM proofs.

Lattice-based zkVMs. Lattice commitments provide a post-quantum replacement for elliptic-curve commitments, with similar homomorphic properties suitable for folding schemes. Lattice-based folding schemes such as LatticeFold [8], Neo [42], and SuperNeo [43] show that lattices can support efficient recursive proof composition. Jolt [1] instantiated with Akita [11] is a lattice-based zkVM that substantially improves over Jolt-Dory. This construction achieves general-purpose and post-quantum proving, but its computational security rests on lattice assumptions. Additionally, Jolt-Akita has not demonstrated end-to-end recursive composition of complete zkVM proofs. Support for recursive proof composition in Jolt-Akita is work in progress at the time of writing [24], and the performance of that recursive composition is not yet publicly known.²

Hash-and-more-based zkVMs. These systems are commonly called hash-based because they commit with Merkle trees, but, as we explain below, their security rests on hash functions and more: algebraic hashes, and in some cases elliptic curves or lattices. They generally arithmetize execution over small prime fields, commonly using primes of roughly 32 bits. They encode the witness vectors with an error-correcting code (e.g., a Reed–Solomon code) and commit to the

² Jolt can prove executions of up to 2^{28} cycles on an M4 Max MacBook Pro within a memory limit of 100 GiB (Section 6.5), so it does not need recursion for such trace lengths.

resulting codewords with a Merkle tree. Examples include SP1 Hypercube [56], RISC Zero [47], and ZisK [62], which prove programs compiled to RISC-V with transparent core proof systems. In practice, however, none of them rests on standard hash functions alone, and their security levels are not uniform. All three use Poseidon or Poseidon2,³ hash functions chosen for circuit efficiency rather than for decades of scrutiny, in their proof or recursion machinery, and the cryptanalysis of such hashes is still evolving, with recent work developing algebraic attacks on Poseidon and Poseidon2 [2,21,38]. SP1 Hypercube’s memory consistency across shards relies on an elliptic-curve multiset hash [49,57], whose collision resistance rests on the discrete logarithm problem in a curve, so its core proof is not post-quantum, and it targets 100 bits of security [59,58]. ZisK uses LtHash, an SIS-based lattice multiset hash, for distributed Fiat-Shamir challenge derivation [37]. The label “hash-based” thus describes these systems’ commitment schemes rather than the assumptions their security rests on.

Hash-based zkVMs. We reserve the term hash-based for zkVMs whose security rests on standard hash functions alone. Among existing systems, only leanVM [32] qualifies. It uses a Spartan-based prover over binary fields and BLAKE2s as its hash function, and it does not assume a proximity-gap conjecture. Its current leanISA, however, is a small specialized machine rather than a standard general-purpose ISA, so it does not support arbitrary programs compiled to RV64.

Summary. None of the existing zkVMs simultaneously provides all three properties: general-purpose RV64 execution; hash-based, post-quantum security from standard hashes without a trusted setup or a conjectured proximity gap for the underlying error-correcting code; and efficient recursion, both for scaling to long executions and for applications that inherently require it.

1.2 Our work: Magnetar

Magnetar occupies this missing point in the design space. It proves programs compiled to RV64I. Its core proving system is BinarySpartan [51], which instantiates Spartan over binary fields. In the same spirit, we instantiate Twist and Shout [55] over binary fields for memory checking, with a refinement to their one-hotness check; we call the resulting protocols BinaryTwist and BinaryShout. We also devise BinaryRouter, a family of sum-check arguments for computations expressible as bit routings, and instantiate it for shifts and sub-word stores. Finally, we devise MosaicPCS, which packs everything the prover commits into a single polynomial over a hash-based polynomial commitment scheme, so that each proof carries one commitment and one evaluation argument.

³ Concurrently with this work, ZisK switched its default hash from Poseidon to BLAKE3 [45]; its September 2026 release describes the BLAKE3 configuration as available for testing ahead of a planned official release [46]. The commit we benchmark in Section 6 predates the switch and defaults to Poseidon. Even after the switch, ZisK’s distributed challenge derivation retains its lattice-based multiset hash, so its security does not rest on standard hash functions alone.

The entire proof stack operates over binary fields. Its computational security rests on standard hashes such as SHA-256 or SHA-3, the same foundation as NIST’s stateless hash-based signature standard [40]. Magnetar does not require a trusted setup, so it is a transparent proof system. It also does not rely on elliptic curves, pairings, lattice assumptions, circuit-friendly hash functions (e.g., Poseidon), or a conjectured Reed–Solomon proximity gap.

Magnetar’s verifier performs Merkle path checks, transcript hashing, and binary-field arithmetic. Its work is therefore dominated by bit-oriented hashing and binary-extension-field arithmetic, the two workloads for which BinarySpartan is designed. Recursive proof verification is thus an efficient workload for Magnetar’s prover. Hence, Magnetar does not require a circuit-friendly hash function.

We implement Magnetar in Rust. At 120 bits of provable security and with 12 performance cores on an M4 Max MacBook Pro, the prover sustains 7.2–8.7 million cycles/s on 2^{22} -cycle executions, with proofs of about 600 KiB that verify in 11–13 ms.

We also compare Magnetar to existing zkVMs on 15 workloads. We report completed application-level operations rather than VM-specific cycles to make an apples-to-apples comparison. Across these workloads, Magnetar’s proving throughput is $24\text{--}66\times$ that of SP1 Hypercube, a hash-and-more-based zkVM; $12\text{--}37\times$ that of Jolt-Dory, a curve-based zkVM; and $6\text{--}20\times$ that of Jolt-Akita, a lattice-based zkVM, with geometric-mean speedups of about $39\times$, $18\times$, and $9.6\times$, respectively. Magnetar’s proofs are about $4.6\times$ smaller than SP1 Hypercube’s; Jolt’s proofs are under 100 KiB, but they rely on elliptic-curve or lattice assumptions.

Optimized Magnetar. Magnetar (opt) includes optimizations on top of Magnetar. On the same workloads, with the same cycle counts and at the same 120 bits of provable security, Magnetar (opt) proves 2^{22} -cycle executions at 9.5–12.8 million cycles/s, $1.32\times$ the rate of Magnetar in geometric mean. Its geometric-mean speedups over SP1 Hypercube, Jolt-Dory, and Jolt-Akita are thus about $52\times$, $24\times$, and $12.7\times$, respectively. Its prover uses 36–40% less memory than that of Magnetar (Section 6.6).

1.3 Closely related work

Hash-and-more-based zkVMs. Many zkVMs, including RISC Zero [47], SP1 [56], and ZisK [62], organize execution into multiple tables. A main CPU table records the machine state and connects each instruction to an operation-specific execution table, such as a table for bitwise AND or XOR. Lookup or permutation arguments connect the CPU rows to these tables, while an AIR enforces each table’s local constraints. In the traditional realization, the prover commits to the resulting trace columns and proves the AIR constraints with a FRI-based STARK.

More recent iterations of these systems use multilinear sum-check-based proofs instead of STARKs. Although this changes an important part of the proving

backend, it retains the CPU and operation tables, their individual relations, and the cross-table lookup or permutation arguments. Their memory-consistency checks likewise rely on offline memory checking [7,52], realized with cross-table permutation arguments, rather than state-of-the-art memory arguments [55]. Permutation and grand-sum arguments also carry a soundness loss linear in the table size, about $\ell/|\mathbb{F}|$ for tables of ℓ rows (Section 2), so over the roughly 124-bit extension fields that SP1 Hypercube and RISC Zero use, tables of 2^{20} rows or more forfeit 20 bits or more, consistent with their security targets of 96–100 bits [48,58]. For efficient recursion, these systems use circuit-friendly hashes such as Poseidon2, and, as Section 1.1 notes, SP1 Hypercube’s memory consistency relies on an elliptic-curve multiset hash and ZisK’s distributed challenge derivation on a lattice-based one, so none of them rests on standard hash functions alone.

Magnetar differs from these systems in execution architecture, proving machinery, and assumptions. Where these systems encode, commit to, and prove separate CPU and operation-table witnesses, Magnetar proves a single shared witness, which reduces prover work. Magnetar combines SIMD R1CS, Virtual Spartan, BinaryRouter, and MosaicPCS to reduce arithmetization and commitment costs and avoid cross-table permutation arguments. It also uses state-of-the-art memory-checking arguments.

Jolt. Jolt [1] uses a “Just One Lookup Table” architecture that reduces instruction semantics to lookups into one large, structured table. For operations on two 64-bit inputs, this conceptual table can have at least 2^{128} entries. Neither the prover nor the verifier materializes it. Instead, the verifier evaluates its multilinear extension at a random point, and a lookup argument proves that the execution accessed the prescribed entries. Jolt combines Spartan [50], for the R1CS that ties the lookups together, with Lasso [54] as that lookup argument; the most recent Jolt replaces Lasso with Twist and Shout [55,25]. Additionally, Jolt supports the elliptic-curve Dory backend [33] and the lattice-based Akita backend [11], both over prime fields. Jolt-Akita is post-quantum: it sizes its Module-SIS instances for 128 bits of security, with a modest loss from its sum-check arguments over a 128-bit field (Section 6).

Like Jolt [1], Magnetar builds on Spartan [50] and on Twist and Shout [55], which it instantiates over binary fields as BinarySpartan [51], BinaryTwist, and BinaryShout. As a result, the two systems share some of their core principles: (1) a sum-check-based proof system and memory-checking arguments, (2) committing to minimal data, and (3) exploiting the repeated structure of CPU executions. Magnetar departs fundamentally from Jolt, however, in how it arithmetizes the processor: it proves instruction semantics with a combination of circuits and BinaryRouter rather than with lookups into a large structured table. Two SIMD R1CS relations form the core of its CPU arithmetization, BinaryRouter handles shifts and sub-word stores, the two operations that neither R1CS nor lookups prove cheaply, and BinaryShout serves only the bytecode. Over binary fields, this approach to arithmetization is faster than proving the same instruction semantics as lookups into a structured table of size 2^{128} or more. Magnetar’s implementation

instantiates the polynomial commitment scheme underlying MosaicPCS with Ligerito [44], a hash-based commitment over a binary extension field, rather than with a curve- or lattice-based commitment.

Magnetar’s approach to scaling the prover to executions that exceed one machine’s memory is different from Jolt’s proposed approach. Nair, Thaler, and Zhu [39] describe a small-space Jolt prover that streams trace contributions across sum-check rounds. It reduces prover space without recursive proof composition, which applications such as succinct blockchains nonetheless require [9]. Their analysis estimates a runtime overhead below a factor of two; we are not aware of an experimental evaluation of a complete streaming zkVM prover. Magnetar instead proposes to prove successive execution chunks and recursively compose their proofs by proving an implementation of its own verifier with the same BinarySpartan-based protocol. Neither proposed scaling path has been implemented. Since Magnetar’s verifier consists primarily of standard hashing and binary-field operations, the recursive verifier workload is well suited to the underlying proof system.

2 A technical overview of Magnetar

We first give an end-to-end overview of the Magnetar protocol and then explain the key ideas behind its design.

2.1 End-to-end protocol

Magnetar proves the fetch-decode-execute loop of an RV64I machine under a Harvard architecture: instruction memory is public and read-only, and data memory is separate and mutable. In a nutshell, we express each cycle’s transition as R1CS over $\mathbb{F}_2$ and over $\mathbb{F}_{2^{128}}$, prove all cycles at once as two SIMD R1CS relations [60] with BinarySpartan [51] (Spartan [50] instantiated over binary fields), and delegate everything that is not a local transition (register and RAM consistency, instruction fetch, shifts, and sub-word stores) to sum-check-based arguments that share one witness with the two relations. For memory checking we use BinaryTwist and BinaryShout (i.e., Twist and Shout [55] instantiated over binary fields). We also devise BinaryRouter, a family of sum-check arguments for computations expressible as bit routings, and instantiate it for shifts and sub-word stores (a store of a byte, halfword, or word into selected lanes of a 64-bit RAM word). Finally, MosaicPCS packs everything the prover commits into one multilinear polynomial over a hash-based polynomial commitment scheme, so the protocol has a single commitment and a single evaluation argument.

A straightforward way to combine these protocols is to commit to the complete witness they share and then run them one after another, using the commitment to connect their claims. We instead sequence them so that the R1CS can read witness segments that other arguments prove correct (e.g., the decoded instruction fields or the shift outputs) without the prover committing to them. We call such segments *virtual*, and the protocols are connected through evaluation claims on them rather than through a commitment. This is the same mechanism by which Spartan itself avoids committing to $A\mathbf{z}$, $B\mathbf{z}$, and $C\mathbf{z}$ (where A , B , and C are

the R1CS constraint matrices and $\mathbf{z}$ is the full assignment): the claims about them that the prover makes at the end of the outer sum-check argument are checked by the inner sum-check argument rather than against a commitment, much as GKR [17] reduces a claim about one layer of a circuit to a claim about the next. Jolt [1,25] uses a similar approach to connect Spartan with Twist and Shout. For ease of reference, we call our variant of BinarySpartan that outputs one evaluation claim per witness segment, committed or virtual, *Virtual Spartan*.

During preprocessing, the verifier deterministically expands each instruction of the public RV64I program into one or more *virtual instructions*, which together form a public bytecode table that depends only on the program and the machine configuration. Each row of this table is identified by a *micro-PC*, the program counter of the expanded program. During proving, the CPU circuit enforces each cycle’s transition using a decoded instruction fetched from this table and delegates the remaining consistency checks: BinaryTwist for registers and BinaryTwist for RAM prove access consistency, the two BinaryRouter instances prove shifts and sub-word stores, and BinaryShout for bytecode authenticates the fetched row and the metadata linked to it. In more detail, the online phase proceeds as follows.

1. *Execute*. The prover runs the expanded program and records the resulting virtual cycles as one trace. This trace is the CPU circuit’s logical witness, shared by its two SIMD R1CS relations. We divide it into directly committed segments and virtual segments (for bytecode, registers, RAM, and shifts), and the prover also builds the auxiliary vectors that BinaryTwist, BinaryShout, and BinaryRouter need in order to check the virtual segments (e.g., one-hot address bits and RAM increments).
2. *Commit*. The prover packs the directly committed CPU segments and the auxiliary vectors into one MosaicPCS polynomial and sends the protocol’s only commitment, before any verifier challenge.
3. *Reduce the CPU claims with Virtual Spartan*. Virtual Spartan reduces the two SIMD R1CS relations, jointly and against their shared witness, to one evaluation claim per witness segment. Claims on direct segments are held for MosaicPCS batching, and claims on virtual segments are passed to the specialized arguments in the next step.
4. *Reduce the virtual claims*. BinaryTwist for registers and BinaryTwist for RAM reduce the two access claims, the BinaryRouter instances reduce the shift-output claim and a RAM-increment claim from BinaryTwist for RAM, and BinaryShout for bytecode then authenticates every claim that is linked to the bytecode. Together, these arguments turn the claims on virtual segments into evaluation claims on vectors committed in the second step.
5. *Prove the committed evaluations*. At this point every remaining claim refers to a component of the MosaicPCS polynomial. MosaicPCS lifts and batches these claims, ring switching reduces them to one claim on the packed polynomial, and the prover proves that claim with a single PCS evaluation argument.

To verify a proof, the verifier evaluates a standard hash function (e.g., SHA-256 or SHA-3) and performs arithmetic over a binary extension field. Magnetar’s underlying proof system, BinarySpartan, proves both efficiently, so recursion does not need a circuit-friendly hash function.

2.2 Key ideas in Magnetar

We now discuss the key ideas in Magnetar, to provide a self-contained treatment of the core ideas before the formal description in Sections 4 and 5.

(1) Dual SIMD R1CS relations over one shared witness. Since BinarySpartan proves R1CS over binary fields, a straightforward way to arithmetize the CPU is as one SIMD R1CS [60] over $\mathbb{F}_2$, where every wire is a bit and every constraint is a product of linear combinations of bits. Bit-level checks (e.g., the XOR and majority logic of the 64-bit adder) are as cheap as they can be in this representation, but word-level operations are not. Selecting one of two 64-bit words under a condition bit costs 64 product constraints, one per bit, and the CPU performs such a selection several times per cycle. The other extreme is to arithmetize everything over $\mathbb{F}_{2^{128}}$. Now a word-wide selection is a single constraint (pack each word into one field element and multiply by the condition bit), but every bit-level product becomes a full 128-bit field multiplication.

We take both. Magnetar arithmetizes the CPU as two SIMD R1CS relations that read the *same* bit-valued witness W : one over $\mathbb{F}_2$ for the bit constraints and one over $\mathbb{F}_{2^{128}}$ for the packed products and packed linear checks (Section 5.1). Each SIMD instance covers several consecutive cycles, and cross-step constraints tie adjacent instances together, so the two relations also enforce the continuity of CPU state (e.g., the micro-PC) from one cycle to the next. Of course, a prover handed two relations could try to satisfy them with two different witnesses. What rules this out is that the evaluation claims produced by both relations are checked against the same commitment (for direct segments) or by the same auxiliary argument (for virtual segments), so the two relations are forced to agree on every witness value. Virtual Spartan (see idea (2) below) additionally reduces both relations with a single inner sum-check argument, which ensures that the prover only needs to prove the evaluation of the witness polynomial at a single point.

(2) Virtual Spartan lets the R1CS read uncommitted segments. Recall that in Spartan the prover commits to the entire R1CS witness before the first challenge, and two sum-check arguments reduce satisfiability to one evaluation claim about that committed polynomial. Naively applying this recipe to our CPU circuit would have the prover commit to all of W , including values whose correctness BinaryTwist, BinaryShout, and the two instances of BinaryRouter described in idea (3) below establish anyway, such as values read from registers and RAM, decoded instruction fields, and shift outputs. Committing to such a segment is unnecessary work unless another argument needs it committed. We instead let the R1CS *read* such segments without committing to them. We call these segments *virtual* and the rest *direct*.

Unfortunately, this does not immediately work with Spartan: its inner sum-check argument ends with a single evaluation claim about the entire witness polynomial, and a PCS can check such a claim only if the whole polynomial is committed. Virtual Spartan (Section 5.3) replaces that claim with one claim per segment. To make this cheap, we lay out W so that each segment occupies an aligned subcube of the Boolean hypercube (i.e., each segment has power-of-two size and starts at a multiple of its size). The claim on W is then a fixed linear combination of one claim per segment, so the prover sends the per-segment values and the verifier checks the combination with a handful of field operations, with no additional sum-check argument. The direct claims are held for the PCS. Each virtual claim is forwarded to the argument that owns it: register accesses to BinaryTwist for the register file, RAM accesses to BinaryTwist for RAM, shift outputs to BinaryRouter for shifts, and the micro-PC and decoded fields to BinaryShout for the bytecode. Each of these arguments discharges its claim by reducing it, through further sum-check arguments, to evaluations of polynomials that *are* committed (e.g., one-hot address bits and RAM increments). Crucially, every claim descends from a single sum-check argument over a single witness, so there is no second trace to reconcile with the first; as a result, Magnetar does not rely on permutation or grand-product arguments. Besides their cost, such arguments carry a soundness loss linear in the instance size, as noted in Section 1.3: a grand-product or grand-sum argument over ℓ elements has soundness error about $\ell/|\mathbb{F}|$, so over a 128-bit field even an instance with $\ell = 2^{20}$ loses 20 bits of security. Magnetar’s sum-check arguments lose only a number of bits logarithmic in the instance size: a sum-check argument over $\ell = 2^{20}$ elements has 20 rounds, and with degree- d rounds its soundness error is about $20d/|\mathbb{F}|$, a loss of only about 6 bits for $d = 3$.

(3) BinaryRouter for bit routings. Two RV64 operations fit neither R1CS relation in Magnetar’s dual SIMD R1CS well. A shift moves every bit of a word to a position determined by the shift amount, and a sub-word store (a byte, halfword, or word) replaces some byte lanes of a 64-bit RAM word while preserving the others. There are two natural options. The first is to encode these operations directly in R1CS. The second, following Jolt [1], is to treat each operation as a lookup into a structured table indexed by its operands. Both options work, but both are more expensive than instantiating BinaryRouter for these operations.

Our observation is that both operations are *bit routings*: every output bit is either zero or a copy of one input bit, and which input bit is fixed by the instruction (and, for a store, by the three low address bits), not by the values being moved. An exception is a shift by a register-specified amount, which we expand into a fixed sequence of virtual instructions, each a static routing (Section 5.4). A routing is therefore the sum $Y(o) = \sum_{u,h} X(u) \cdot Q(h) \cdot R(o, u, h)$, where X holds the candidate source bits, Q is a one-hot selector of the routing h in use, and R is a public tensor recording which source feeds which output under each routing.

Virtual Spartan hands us an evaluation claim about the output Y , and we invoke the sum-check protocol on this identity to reduce it to an evaluation of X and an

evaluation of Q . Since R is public and sparse (under any one routing, each output bit is fed by at most two source bits), the verifier can evaluate its multilinear extension on its own, and the prover’s work in the sum-check argument is linear in the trace length. X consists of direct segments of W , so its evaluation goes to the PCS. The evaluation of Q is a field of the bytecode row, so we have BinaryShout authenticate it with one additional query to the bytecode table.

For stores, we apply the same identity with the old RAM word and the store source as the sources and BinaryTwist’s committed write increment (the XOR of the old and new words) as the output (Section 5.4). The new word is a routing of the two sources, so each increment bit is either zero or the XOR of an old bit and a store-source bit; R expresses the XOR by feeding that output bit from both source bits. Both sources are virtual segments of W , so W also has a direct segment that holds a copy of the two words. The R1CS binds this copy to the originals, and BinaryRouter reads the copy. A store’s routing also depends on the three low address bits, so R takes them as extra arguments, and the reduction also outputs evaluation claims on these bits, which lie in a direct segment of W .

(4) MosaicPCS, one commitment for everything. By the end of the reductions above, the verifier holds evaluation claims about dozens of distinct bit-valued polynomials of wildly different sizes: the direct witness segments, the one-hot address components of two BinaryTwist instances and one BinaryShout instance, and the register and RAM increments. Naively, each needs its own commitment and its own evaluation argument, and with a hash-based PCS every evaluation argument carries its own Merkle authentication paths.

We avoid this cost with MosaicPCS (Section 5.2), a packing layer on top of the PCS. Every component in Magnetar has power-of-two size, so we place all of them on disjoint, aligned subcubes of one larger Boolean hypercube, define a single polynomial G that agrees with each component p_i on its own subcube and is zero outside the union of the subcubes, and commit to G once. The point of insisting on aligned subcubes is that a claim $\tilde{p}_i(\mathbf{r}) = v$ lifts *exactly* to $\tilde{G}(\mathbf{r} \parallel s_i) = v$, where s_i is the fixed Boolean suffix selecting the subcube. The verifier appends s_i to the point, and nothing else happens. The lifted claims are batched, and ring switching [13] reduces the result to one evaluation of the packed representation of G , which a single PCS argument proves against the single commitment. Proving throughput is unaffected, since the prover commits to essentially the same bits it would have committed to anyway.

Remark 1. MosaicPCS is related to prior techniques for proving evaluations of sparse multilinear polynomials. Spark [50] supports general sparse multilinear polynomials. JaggedPCS [23] specializes sparse polynomial commitments to stacked polynomial families with arbitrary heights. With JaggedPCS, a claim on a sparse component must first be reduced, through an additional adapter sum-check argument, to a claim on the committed dense polynomial. MosaicPCS needs no such adapter. Because each segment occupies an aligned subcube, MosaicPCS lifts a local virtual evaluation claim to a claim on the committed polynomial by

simply appending fixed Boolean selector coordinates. This lift does not require a sum-check reduction.

3 Preliminaries

We work over the binary field $\mathbb{B} = \mathbb{F}_2$ and its degree-128 extension $\mathbb{K} = \mathbb{F}_{2^{128}}$. The field $\mathbb{K}$ has 2^{128} elements, and we identify $\mathbb{B}$ with its canonical image in $\mathbb{K}$. Bit-level witnesses and constraint coefficients may lie in $\mathbb{B}$, but all verifier challenges and multilinear evaluation claims lie in $\mathbb{K}$. For a positive integer ℓ , we identify an index in $[2^\ell]$ with its little-endian binary representation in $\{0, 1\}^\ell$.

3.1 Multilinear extensions

Index $\mathbf{v} \in \mathbb{K}^{2^\ell}$ by $\mathbf{b} \in \{0, 1\}^\ell$. Its multilinear extension is the unique multilinear polynomial that agrees with $\mathbf{v}$ on the Boolean hypercube:

$$\tilde{\mathbf{v}}(\mathbf{x}) = \sum_{\mathbf{b} \in \{0, 1\}^\ell} \mathbf{v}[\mathbf{b}] \cdot \text{eq}(\mathbf{x}, \mathbf{b}), \quad \text{eq}(\mathbf{x}, \mathbf{b}) = \prod_{i=1}^{\ell} (x_i \cdot b_i + (1 - x_i) \cdot (1 - b_i)).$$

The same convention applies to matrices and tensors after fixing an explicit order for their Boolean index coordinates.

3.2 The sum-check protocol

Let g be an m -variate polynomial over $\mathbb{K}$ with individual degree at most d . We use the sum-check protocol [36] as a public-coin reduction of knowledge (Definition 2) from the relation

$$S = \sum_{\mathbf{b} \in \{0, 1\}^m} g(\mathbf{b})$$

to the polynomial-evaluation relation

$$y = g(\mathbf{r}), \quad \mathbf{r} \in \mathbb{K}^m.$$

Here g is bound before the first sum-check challenge; in every use below, it is determined by public data, prior transcript messages, and witness polynomials that are committed or defined from committed ones. The protocol runs for m public-coin rounds, is perfectly complete, communicates $O(md)$ field elements, and has soundness error at most $md/|\mathbb{K}|$.

The definitions below are adapted from prior work [26, 29, 30].

3.3 Preprocessing SNARKs

Definition 1 (Preprocessing SNARK). Consider a relation $\mathcal{R}$ over public-parameter, structure, instance, and witness tuples $(\text{pp}, \mathbf{s}, x, w)$. A preprocessing SNARK for $\mathcal{R}$ consists of the following polynomial-time algorithms:

- $\text{Setup}(1^\lambda) \rightarrow \text{pp}$ generates public parameters;
- $\text{KeyGen}(\text{pp}, \mathbf{s}) \rightarrow (\text{pk}, \text{vk})$ preprocesses the structure $\mathbf{s}$;

- $\text{Prove}(\text{pk}, x, w) \rightarrow \pi$ proves that $(\text{pp}, s, x, w) \in \mathcal{R}$; and
- $\text{Verify}(\text{vk}, x, \pi) \rightarrow \{0, 1\}$ verifies the proof.

The algorithms satisfy the following requirements.

(i) **Perfect completeness.** For any polynomial-time adversary $\mathcal{A}$,

$$\Pr \left[\text{Verify}(\text{vk}, x, \pi) = 1 \mid \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda), \\ (s, (x, w)) \leftarrow \mathcal{A}(\text{pp}), \\ (\text{pp}, s, x, w) \in \mathcal{R}, \\ (\text{pk}, \text{vk}) \leftarrow \text{KeyGen}(\text{pp}, s), \\ \pi \leftarrow \text{Prove}(\text{pk}, x, w) \end{array} \right] = 1.$$

(ii) **Knowledge soundness.** For every expected polynomial-time adversary $\mathcal{A}$ and prover $\mathcal{P}^*$, there exists an expected polynomial-time extractor $\mathcal{E}$ such that

$$\begin{aligned} & \Pr \left[(\text{pp}, s, x, w) \in \mathcal{R} \mid \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda), \\ (s, x, \text{st}) \leftarrow \mathcal{A}(\text{pp}), \\ w \leftarrow \mathcal{E}(\text{pp}, s, x, \text{st}) \end{array} \right] \\ & \geq \Pr \left[\text{Verify}(\text{vk}, x, \pi) = 1 \mid \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda), \\ (s, x, \text{st}) \leftarrow \mathcal{A}(\text{pp}), \\ (\text{pk}, \text{vk}) \leftarrow \text{KeyGen}(\text{pp}, s), \\ \pi \leftarrow \mathcal{P}^*(\text{pk}, x, \text{st}) \end{array} \right] - \text{negl}(\lambda). \end{aligned}$$

A non-interactive argument of knowledge is succinct if its proof length and verifier time are polylogarithmic in the size of the structure and witness. It is transparent if setup and key generation require no secret trapdoor.

3.4 Reductions of knowledge

We recall the reductions-of-knowledge framework of Kothapalli and Parno [26]. A reduction of knowledge generalizes an argument of knowledge: the verifier interactively reduces checking knowledge of a witness in a relation $\mathcal{R}_1$ to checking knowledge of a witness in another relation $\mathcal{R}_2$. The prover and the verifier take as input an instance u_1 , and the prover additionally takes a corresponding witness w_1 . After the interaction, the prover and the verifier output a new instance u_2 , and the prover additionally outputs a corresponding witness w_2 .

Definition 2 (Preprocessed reduction of knowledge). Consider relations $\mathcal{R}_1$ and $\mathcal{R}_2$ over public-parameter, structure, instance, and witness tuples. A preprocessed reduction of knowledge from $\mathcal{R}_1$ to $\mathcal{R}_2$ consists of a setup algorithm, a deterministic key-generation algorithm, and an interactive prover and verifier with the following interface:

- $\text{Setup}(1^\lambda) \rightarrow \text{pp}$ generates public parameters;

- $\text{KeyGen}(\text{pp}, \text{s}_1) \rightarrow (\text{pk}, \text{vk}, \text{s}_2)$ preprocesses the input structure and outputs the structure of the target relation;
- $\mathcal{P}(\text{pk}, u_1, w_1) \rightarrow (u_2, w_2)$ interactively reduces the input instance-witness pair to an output instance-witness pair; and
- $\mathcal{V}(\text{vk}, u_1) \rightarrow u_2$ interactively reduces the input instance to the output instance.

The prover and verifier output the same instance u_2 , while only the prover outputs w_2 . The algorithms satisfy the following requirements.

- (i) **Completeness.** For any $(\text{pp}, \text{s}_1, u_1, w_1)$ in $\mathcal{R}_1$, if $\text{KeyGen}(\text{pp}, \text{s}_1)$ outputs $(\text{pk}, \text{vk}, \text{s}_2)$ and the honest interaction using these keys and (u_1, w_1) outputs (u_2, w_2) , then $(\text{pp}, \text{s}_2, u_2, w_2) \in \mathcal{R}_2$.
- (ii) **Knowledge soundness.** For every expected polynomial-time adversary $\mathcal{A}$ and prover $\mathcal{P}^*$, there exists an expected polynomial-time extractor $\mathcal{E}$. Define

$$\epsilon_{\text{ext}} = \Pr \left[(\text{pp}, \text{s}_1, u_1, w_1) \in \mathcal{R}_1 \mid \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda), \\ (\text{s}_1, u_1, \text{st}) \leftarrow \mathcal{A}(\text{pp}), \\ w_1 \leftarrow \mathcal{E}(\text{pp}, \text{s}_1, u_1, \text{st}) \end{array} \right]$$

and

$$\epsilon_{\text{out}} = \Pr \left[(\text{pp}, \text{s}_2, u_2, w_2) \in \mathcal{R}_2 \mid \begin{array}{l} \text{pp} \leftarrow \text{Setup}(1^\lambda), \\ (\text{s}_1, u_1, \text{st}) \leftarrow \mathcal{A}(\text{pp}), \\ (\text{pk}, \text{vk}, \text{s}_2) \leftarrow \text{KeyGen}(\text{pp}, \text{s}_1), \\ (u_2, w_2) \leftarrow \langle \mathcal{P}^*, \mathcal{V} \rangle((\text{pk}, \text{vk}), u_1, \text{st}) \end{array} \right].$$

Knowledge soundness requires $\epsilon_{\text{ext}} \geq \epsilon_{\text{out}} - \text{negl}(\lambda)$.

- (iii) **Public reducibility.** There exists a deterministic polynomial-time function φ such that, for every interaction transcript tr , $\varphi(\text{pp}, \text{s}_1, u_1, \text{tr}) = u_2$.

Reductions of knowledge compose sequentially and in parallel. If $\Pi_1 : \mathcal{R}_1 \rightarrow \mathcal{R}_2$ and $\Pi_2 : \mathcal{R}_2 \rightarrow \mathcal{R}_3$, then $\Pi_2 \circ \Pi_1$ is a reduction from $\mathcal{R}_1$ to $\mathcal{R}_3$. Reductions on independent relations compose in parallel, denoted by $\times$.

All reductions used below are public-coin and publicly reducible. Applying Fiat-Shamir [15] makes them non-interactive in the random oracle model.

3.5 R1CS

Definition 3 (R1CS). Fix a field $\mathbb{F}$ and size bounds $m, n, \ell \in \mathbb{N}$ with $m > \ell$. An R1CS structure is $\mathbf{s} = (A, B, C)$, where $A, B, C \in \mathbb{F}^{m \times m}$ have at most n non-zero entries each. An instance $\mathbf{x} \in \mathbb{F}^\ell$ contains the public inputs and outputs. It is satisfied by a witness $\mathbf{w} \in \mathbb{F}^{m-\ell-1}$ if, for $\mathbf{z} = (\mathbf{w}, 1, \mathbf{x})$,

$$(A\mathbf{z}) \circ (B\mathbf{z}) = C\mathbf{z},$$

where $\circ$ denotes the Hadamard product. We write $(\mathbf{s}, \mathbf{x}, \mathbf{w}) \in \mathcal{R}_{\text{R1CS}}$ for a satisfying structure-instance-witness tuple.

Definition 4 (SIMD R1CS). Let q be a power of two. A SIMD R1CS applies one structure $\mathbf{s} = (A, B, C)$ to q instance-witness pairs $(\mathbf{x}_i, \mathbf{w}_i)_{i \in [q]}$. It is satisfied if, for $\mathbf{z}_i = (\mathbf{w}_i, 1, \mathbf{x}_i)$,

$$(A\mathbf{z}_i) \circ (B\mathbf{z}_i) = C\mathbf{z}_i \quad \text{for every } i \in [q].$$

Equivalently, it is the block-diagonal constraint system with matrices $I_q \otimes A$, $I_q \otimes B$, and $I_q \otimes C$ acting on the concatenation of the vectors $\mathbf{z}_i$. We write

$$(\mathbf{s}, (\mathbf{x}_i)_{i \in [q]}, (\mathbf{w}_i)_{i \in [q]}) \in \mathcal{R}_{\text{SIMD-R1CS}}$$

for a satisfying tuple. We write $\mathcal{R}_{\text{SIMD-R1CS}}^{\mathbb{B}}$ for the restriction in which every witness entry is in $\mathbb{B}$; the constraints may be evaluated over $\mathbb{K}$ after the canonical embedding $\mathbb{B} \hookrightarrow \mathbb{K}$.

3.6 Polynomial commitments

Definition 5 (Virtual evaluation relation). A virtual evaluation claim states an evaluation of a multilinear polynomial without yet tying that polynomial to a commitment. For a field $\mathbb{F}$, its relation is

$$\mathcal{R}_{\text{veval}}[\mathbb{F}] = \left\{ ((\mathbf{r}, y), \mathbf{v}) \mid \mathbf{v} \in \mathbb{F}^{2^\ell}, \mathbf{r} \in \mathbb{F}^\ell, y \in \mathbb{F}, \tilde{\mathbf{v}}(\mathbf{r}) = y \right\}.$$

We abbreviate $\mathcal{R}_{\text{veval}}[\mathbb{K}]$ as $\mathcal{R}_{\text{veval}}$. We write $\mathcal{R}_{\text{veval}}^*$ for a finite product of these relations and $\mathcal{R}_{\text{veval}}^{\mathbb{B}}$ for the restriction in which $\mathbf{v}$ is bit-valued and its multilinear extension is evaluated over $\mathbb{K}$.

Definition 6 (Multilinear polynomial commitment scheme). A multilinear polynomial commitment scheme is a commitment scheme for vectors $\mathbf{v} \in \mathbb{K}^{2^\ell}$, viewed through their multilinear extensions, equipped with an argument of knowledge for the relation

$$\mathcal{R}_{\text{ceval}} = \left\{ (\text{pp}, (\text{cm}, \mathbf{r}, y), (\mathbf{v}, \rho)) \mid \begin{array}{l} \mathbf{v} \in \mathbb{K}^{2^\ell}, \\ \mathbf{r} \in \mathbb{K}^\ell, \quad y \in \mathbb{K}, \\ \tilde{\mathbf{v}}(\mathbf{r}) = y, \\ \text{cm} = \text{Commit}(\text{pp}, \mathbf{v}; \rho) \end{array} \right\}.$$

We call an instance of $\mathcal{R}_{\text{ceval}}$ a committed evaluation claim.

We use this PCS interface as a black box. Since our witnesses are bit-valued but the PCS commits to vectors over $\mathbb{K}$, we rely on ring switching [13], which reduces a virtual evaluation claim on a binary vector to one on the corresponding packed polynomial over $\mathbb{K}$ (i.e., 128 bits per field element).

4 Magnetar's proof-system building blocks

This section describes the three building blocks that Magnetar composes: BinarySpartan [51], its core proof engine, and BinaryTwist and BinaryShout, which

handle memory checking. In a nutshell, we present each of them as a reduction of knowledge (RoK) [26] from its native relation (R1CS, or a memory-checking relation) to virtual evaluation claims about bit-valued polynomials, since that is the form in which Section 5 composes them. This section borrows from prior work [50,51,55]. The RoK view of Spartan differs from its usual presentation as an argument of knowledge or as a polynomial IOP [18].

4.1 Spartan

Assume without loss of generality that m is a power of two and that

$$|\mathbf{w}| = 1 + |\mathbf{x}| = m/2,$$

where m is the number of constraints in Definition 3; both conditions can be met with padding. Let $\mathbf{z} = (\mathbf{w}, 1, \mathbf{x}) \in \mathbb{F}^m$. For $\mathbf{y} \in \mathbb{F}^{\log m}$, write $\mathbf{y}[0]$ for its first coordinate and $\mathbf{y}[1..]$ for the remaining coordinates. Then

$$\tilde{\mathbf{z}}(\mathbf{y}) = (1 - \mathbf{y}[0]) \cdot \tilde{\mathbf{w}}(\mathbf{y}[1..]) + \mathbf{y}[0] \cdot \widetilde{(1, \mathbf{x})}(\mathbf{y}[1..]). \quad (1)$$

Indeed, the RHS is a multilinear polynomial that agrees with $\mathbf{z}$ on the Boolean hypercube and is therefore the unique multilinear extension of $\mathbf{z}$, so the verifier can reconstruct an evaluation of $\tilde{\mathbf{z}}$ from one evaluation of the witness and the public input.

Spartan [50] reduces R1CS satisfiability to one virtual evaluation claim on the witness polynomial using two sum-check arguments; we restate it here as a reduction of knowledge. The prover commits to $\mathbf{w}$ before the reduction begins (i.e., before any verifier challenge), and a PCS evaluation argument proves the final claim against that commitment.

The outer sum-check argument reduces the task of proving $(A\mathbf{z}) \circ (B\mathbf{z}) = C\mathbf{z}$ to evaluating the multilinear extensions of $A\mathbf{z}$, $B\mathbf{z}$, and $C\mathbf{z}$ at one random point by proving

$$0 = \sum_{x \in \{0,1\}^{\log m}} \text{eq}(\tau, x) \cdot (\tilde{A\mathbf{z}}(x) \cdot \tilde{B\mathbf{z}}(x) - \tilde{C\mathbf{z}}(x)), \quad (2)$$

where $\tau \in \mathbb{F}^{\log m}$ is chosen at random by the verifier after the prover has committed to $\mathbf{w}$. If $\mathbf{z}$ satisfies the R1CS, this equality holds for every τ . Otherwise, the probability that it holds for a random τ is at most $\log m/|\mathbb{F}|$ by the Schwartz–Zippel lemma. The outer sum-check argument binds x to a random $r_x \in \mathbb{F}^{\log m}$. At the end of the sum-check argument, the prover sends three values v_A , v_B , and v_C that are claimed to equal $\tilde{A\mathbf{z}}(r_x)$, $\tilde{B\mathbf{z}}(r_x)$, and $\tilde{C\mathbf{z}}(r_x)$, and the verifier checks that $e = \text{eq}(\tau, r_x) \cdot (v_A \cdot v_B - v_C)$, where e is the claim left by the last round of the sum-check protocol. The verifier is now left with checking the correctness of the three claimed values, which is the job of the inner sum-check argument.

To check all three claims with one sum-check argument, the verifier samples $r \in \mathbb{F}$, sets $c = v_A + r \cdot v_B + r^2 \cdot v_C$, and checks

$$c = \sum_{y \in \{0,1\}^{\log m}} \left(\tilde{A}(r_x, y) + r \cdot \tilde{B}(r_x, y) + r^2 \cdot \tilde{C}(r_x, y) \right) \cdot \tilde{\mathbf{z}}(y). \quad (3)$$

Because v_A, v_B, v_C are fixed before r is sampled, an incorrect triple passes this batching step with probability at most $2/|\mathbb{F}|$. The inner sum-check argument binds y to a random $r_y \in \mathbb{F}^{\log m}$. The prover supplies $v_w = \tilde{\mathbf{w}}(r_y[1..])$. The verifier evaluates the public matrix polynomials and reconstructs $\tilde{\mathbf{z}}(r_y)$ from v_w and $(1, \mathbf{x})(r_y[1..])$ using Equation (1). The remaining claim is therefore a single evaluation claim on the witness polynomial, $\tilde{\mathbf{w}}(r_y[1..]) = v_w$. The reduction outputs it as a virtual evaluation claim (Definition 5), i.e., without tying it to the commitment to $\mathbf{w}$; the PCS evaluation argument does that afterwards.

Remark 2. Since the commitment to $\mathbf{w}$ is sent before the reduction begins, the Spartan reduction could equally output a committed evaluation claim (Definition 6). We keep the output commitment-free because Magnetar will later split this claim by segment and tie only some of the pieces to a commitment (Section 5.3).

Construction 1 (Spartan reduction). On input $\mathbf{s} = (A, B, C)$, **KeyGen** stores the matrices, dimensions, and padding convention in the prover and verifier keys and outputs the target evaluation structure for vectors of length $m/2$. The reduction itself sends no commitments; we assume that the surrounding protocol has already sent a commitment to $\mathbf{w}$ before the first step. On input $(\mathbf{x}, \mathbf{w})$, with $\mathbf{z} = (\mathbf{w}, 1, \mathbf{x})$, the online reduction is:

1. $\mathcal{V} \rightarrow \mathcal{P}$: $\tau \leftarrow \mathbb{F}^{\log m}$.
2. $\mathcal{P}$ and $\mathcal{V}$ run the sum-check protocol for the claim $\sum_{x \in \{0,1\}^{\log m}} g(x) = 0$, where

$$g(x) = \text{eq}(\tau, x) \cdot (\tilde{A}\mathbf{z}(x) \cdot \tilde{B}\mathbf{z}(x) - \tilde{C}\mathbf{z}(x))$$
 has degree three in each variable. Over its $\log m$ rounds, $\mathcal{V}$ samples $r_x \in \mathbb{F}^{\log m}$ one coordinate per round, and the protocol ends with a claim $e = g(r_x)$.
3. $\mathcal{P} \rightarrow \mathcal{V}$: $v_A, v_B, v_C \in \mathbb{F}$, claimed to equal $\tilde{A}\mathbf{z}(r_x)$, $\tilde{B}\mathbf{z}(r_x)$, and $\tilde{C}\mathbf{z}(r_x)$. $\mathcal{V}$ checks that $e = \text{eq}(\tau, r_x) \cdot (v_A \cdot v_B - v_C)$ and aborts otherwise.
4. $\mathcal{V} \rightarrow \mathcal{P}$: $r \leftarrow \mathbb{F}$. Both parties set $c = v_A + r \cdot v_B + r^2 \cdot v_C$.
5. $\mathcal{P}$ and $\mathcal{V}$ run the sum-check protocol for the claim $\sum_{y \in \{0,1\}^{\log m}} h(y) = c$, where

$$h(y) = (\tilde{A}(r_x, y) + r \cdot \tilde{B}(r_x, y) + r^2 \cdot \tilde{C}(r_x, y)) \cdot \tilde{\mathbf{z}}(y)$$
 has degree two in each variable. $\mathcal{V}$ samples $r_y \in \mathbb{F}^{\log m}$ one coordinate per round, and the protocol ends with a claim $e' = h(r_y)$.
6. $\mathcal{P} \rightarrow \mathcal{V}$: $v_w \in \mathbb{F}$, claimed to equal $\tilde{\mathbf{w}}(r_y[1..])$. $\mathcal{V}$ computes $v_z = (1 - r_y[0]) \cdot v_w + r_y[0] \cdot (1, \mathbf{x})(r_y[1..])$ by Equation (1), evaluates $\tilde{A}(r_x, r_y)$, $\tilde{B}(r_x, r_y)$, and $\tilde{C}(r_x, r_y)$ directly, and checks that $e' = (\tilde{A}(r_x, r_y) + r \cdot \tilde{B}(r_x, r_y) + r^2 \cdot \tilde{C}(r_x, r_y)) \cdot v_z$, aborting otherwise.
7. $\mathcal{P}$ and $\mathcal{V}$ output the $\mathcal{R}_{\text{eval}}[\mathbb{F}]$ instance $(r_y[1..], v_w)$; $\mathcal{P}$ additionally outputs the witness $\mathbf{w}$.

Remark 3. In Construction 1, the verifier evaluates the multilinear extensions of A , B , and C on its own, which takes time linear in their number of non-zero entries. Spartan [50] also provides a preprocessing variant in which **KeyGen** commits to the three matrices with a sparse polynomial commitment scheme, the verifier retains only those succinct commitments, and the prover later proves the required evaluations of the committed matrices, so that the verifier’s online work is succinct even for unstructured R1CS. We do not consider this general variant: Magnetar works with structured R1CS (SIMD R1CS with a small, fixed base structure), for which direct evaluation by the verifier is already cheap.

4.2 Spartan for SIMD R1CS

Phalanx [60] extends the Spartan reduction to SIMD R1CS (Definition 4). Consider $q = 2^s$ instance-witness pairs for the same $m \times m$ structure $\mathbf{s} = (A, B, C)$. The SIMD relation can be viewed either as q copies of the base R1CS, one for each pair, or as one R1CS with block-diagonal matrices $I_q \otimes A$, $I_q \otimes B$, and $I_q \otimes C$. The prover arranges the q witnesses as the rows of a $q \times m/2$ matrix W , so that one multilinear polynomial $\widetilde{W}(u, y)$ represents all of them, with $u \in \{0, 1\}^{\log q}$ indexing the instance and y the column.

The outer sum-check argument applies to the block-diagonal R1CS without change. A row index for the block-diagonal R1CS consists of s instance coordinates and $\log m$ coordinates for a row of the base R1CS. The verifier chooses $r_x = (r_{\text{inst}}, r_{\text{row}})$ one coordinate at a time over the course of the sum-check argument.

The shared structure matters in the inner sum-check argument. For $M \in \{A, B, C\}$, block diagonality gives

$$\widetilde{I_q \otimes M}((r_{\text{inst}}, r_{\text{row}}), (u, y)) = \text{eq}(r_{\text{inst}}, u) \cdot \widetilde{M}(r_{\text{row}}, y).$$

The sum over u therefore folds the q vectors $\mathbf{z}_i$ at r_{inst} , and the prover and the verifier run the inner sum-check argument once, over the columns y of the base R1CS. Note that the inner sum-check argument does not check any one SIMD instance in isolation but rather a random fold (i.e., a random linear combination) of all q of them, which is all that soundness requires. The reduction ends with one virtual evaluation of the jointly defined witness polynomial.

Only the outer sum-check argument gains $\log q$ rounds, while the inner sum-check argument and the three matrix evaluations keep the dimensions of the base R1CS. In our context the base R1CS is fixed and small (a fixed number of consecutive CPU transitions), so the verifier evaluates its matrices directly rather than invoking Spartan’s sparse-matrix evaluation argument (Remark 3).

Lemma 1 (Spartan reductions for R1CS and SIMD R1CS [50,60]). *For an $m \times m$ R1CS structure $\mathbf{s} = (A, B, C)$, Construction 1 and its SIMD extension have the following guarantees.*

1. *The base reduction is a public-coin RoK*

$$\mathcal{R}_{\text{R1CS}} \longrightarrow \mathcal{R}_{\text{veval}}[\mathbb{F}].$$

It has $O(\log m)$ public-coin rounds, communicates $O(\log m)$ field elements, and has knowledge error $O(\log m)/|\mathbb{F}|$. The prover performs $O(\text{nnz}(A, B, C) + m)$ field operations. With direct evaluation of the three public matrix polynomials, the verifier performs $O(\text{nnz}(A, B, C) + |\mathbf{x}| + \log m)$ field operations.

2. For q instance-witness pairs for the same structure, the SIMD reduction is a public-coin RoK

$$\mathcal{R}_{\text{SIMD-RICS}} \longrightarrow \mathcal{R}_{\text{veval}}[\mathbb{F}].$$

It has $O(\log q + \log m)$ public-coin rounds, communicates $O(\log q + \log m)$ field elements, and has knowledge error $O(\log(qm))/|\mathbb{F}|$. The prover performs

$$O(q \cdot \text{nnz}(A, B, C) + qm)$$

field operations. With direct evaluation of the three base matrix polynomials, the verifier performs $O(\text{nnz}(A, B, C) + \sum_{i \in [q]} |\mathbf{x}_i| + \log q + \log m)$ field operations.

Note that the verifier's cost in the SIMD reduction is essentially that of a single instance: it runs only $\log q$ more sum-check rounds (plus reading the public inputs, which need not grow with q). The prover's work is linear in q , the same as proving the instances one by one.

4.3 BinarySpartan

BinarySpartan [51] restricts the witnesses in Phalanx's SIMD construction to $\mathbb{B}$. BinarySpartan incorporates standard Spartan optimizations, including refinements to the sum-check arguments [60,53,22,12,3]. For ease of reference, we call its RICS-to-evaluation stage the *BinarySpartan reduction*. The standalone BinarySpartan SNARK wraps this reduction with one commitment, ring switching, and one PCS evaluation argument.

BinarySpartan proceeds in three steps, treating the multilinear PCS from Definition 6 as a black box.

1. **Commit to the binary witness.** Representing the witness as a vector over $\mathbb{K}$ would use one field element per bit. Instead, for a witness of length 2^ℓ , view its entries as a table

$$w : \{0, 1\}^7 \times \{0, 1\}^{\ell-7} \rightarrow \mathbb{B}.$$

Let $u_0, \dots, u_{127}$ enumerate $\{0, 1\}^7$, and let $\beta_0, \dots, \beta_{127}$ be a $\mathbb{B}$ -basis of $\mathbb{K}$. Define the packed table

$$P(v) = \sum_{j=0}^{127} \beta_j w(u_j, v).$$

Each entry of P contains 128 witness bits. The prover commits to P with the PCS.

2. **Reduce SIMD R1CS over binary witnesses.** After sending the commitment, the prover and the verifier run the SIMD Spartan reduction. The binary restriction applies to the witness, not to the protocol’s arithmetic: both sum-check arguments run over $\mathbb{K}$. The SIMD Spartan reduction outputs the claim $\tilde{w}(r) = v_w$, so the target relation is $\mathcal{R}_{\text{veval}}^{\mathbb{B}}$ rather than $\mathcal{R}_{\text{veval}}$.
3. **Prove the binary-witness evaluation.** Ring switching [13] reduces the binary-witness virtual evaluation claim to a virtual evaluation claim for $\tilde{P}$. The PCS evaluation argument proves the resulting committed evaluation claim against the commitment sent in the first step.

Corollary 1 (BinarySpartan [51]). *The SIMD reduction in Lemma 1, restricted to bit-valued witnesses and run over $\mathbb{K}$, is a public-coin RoK*

$$\Pi_{\text{BSp}} : \mathcal{R}_{\text{SIMD-R1CS}}^{\mathbb{B}} \longrightarrow \mathcal{R}_{\text{veval}}^{\mathbb{B}}.$$

Let Π_{RS} denote the ring-switching reduction and Π_{PCS} a transparent, succinct PCS evaluation argument with perfect completeness and knowledge soundness. If the public matrix polynomials are verifier-efficiently evaluable, then the protocol that commits once to P before Π_{BSp} and then runs Π_{RS} and Π_{PCS} is a transparent preprocessing SNARK for SIMD R1CS over binary witnesses.

4.4 BinaryTwist and BinaryShout

BinarySpartan proves that each cycle’s local constraints hold, but nothing so far ties the register values, RAM words, and decoded instructions that the CPU circuit reads to a consistent register file, memory, and bytecode. We establish this consistency with BinaryShout and BinaryTwist (i.e., Shout and Twist [55] instantiated over binary fields). Recall that Shout proves reads from a public read-only table, and Twist proves p reads followed by one write per cycle against a mutable memory; we use one instance of BinaryShout for the bytecode and two instances of BinaryTwist, one for the register file ($p = 2$) and one for RAM ($p = 1$). As with Spartan, the reductions below send no commitments: the prover commits to every component polynomial they use up front (in Magnetar, as part of the single MosaicPCS commitment cm_G), each reduction outputs virtual evaluation claims on those components, and the PCS evaluation argument at the end proves the corresponding committed claims.

Although Twist and Shout were introduced for prime fields and homomorphic commitments, they suit binary fields and hash-based commitments well: the data they commit to, one-hot addresses and increments, consists of bit vectors, and the addresses are moreover sparse, so both are cheap to commit once packed into binary-field elements.

The two protocols share their address encoding and their one-hot check, which we describe first. We then summarize each protocol and state its guarantees. Each reduction below works with a few polynomials that the prover commits to and several *virtual* ones that are never committed; a virtual polynomial enters only

through a sum-check argument that reduces a claim about it to claims about committed ones.

Shared address representation. Let $K = 2^a$ be the number of memory cells and $T = 2^n$ the number of cycles. Choose $d \mid a$. For each address sequence, with one address per cycle, the prover commits to d bit-valued component polynomials

$$\mathbf{ra}_i : \{0, 1\}^{a/d} \times \{0, 1\}^n \rightarrow \mathbb{B}, \quad i \in [d].$$

If $k = (k^{(1)}, \dots, k^{(d)})$ is the corresponding partition of a memory address, the virtual full-address polynomial satisfies

$$\mathbf{ra}(k, j) = \prod_{i=1}^d \mathbf{ra}_i(k^{(i)}, j) \quad \text{on Boolean cycles } j.$$

The prover commits only to the d component polynomials $\mathbf{ra}_1, \dots, \mathbf{ra}_d$, which are much shorter than $\mathbf{ra}$: the full address has $K \cdot T$ entries, whereas each component has $K^{1/d} \cdot T$, so the prover commits to $d \cdot K^{1/d}$ bits per cycle rather than K . The full address $\mathbf{ra}$, the ordinary address bits, and the values read or written are virtual: the prover never commits to them. Whenever the verifier needs an evaluation of one of these virtual polynomials (e.g., $\tilde{\mathbf{ra}}(r_{\text{address}}, r_{\text{cycle}})$), a sum-check argument over the identity that defines it reduces that evaluation to evaluations of the committed component polynomials $\tilde{\mathbf{ra}}_i$. We refer to this step as *address virtualization*.

One-hotness over the binary field. Over $\mathbb{F}_2$, checking that the entries of a bit vector sum to one establishes only that its Hamming weight is odd, so BinaryTwist and BinaryShout must separately prove that every row $\mathbf{ra}_i(\cdot, j)$ of each address component has Hamming weight at most one. Fix one component and write $A : \{0, 1\}^{a/d} \times \{0, 1\}^n \rightarrow \mathbb{B}$ for its Boolean table and $\tilde{A}$ for its multilinear extension. Twist and Shout [55] check this with the *collision polynomial*

$$\Delta(j; r, s) = \tilde{A}(r, j) \cdot \tilde{A}(s, j) - \sum_{u \in \{0, 1\}^{a/d}} A(u, j) \cdot \mathbf{eq}(r, u) \cdot \mathbf{eq}(s, u),$$

which, for a bit-valued row $A(\cdot, j)$, is the zero polynomial in (r, s) exactly when the row has Hamming weight at most one. (Expanding the product shows that Δ is the sum of $\mathbf{eq}(r, k) \cdot \mathbf{eq}(s, k')$ over all pairs $k \neq k'$ of positions that are both set.) A sum-check argument over the cycle variable at random r , s , and ρ then checks that $\sum_j \mathbf{eq}(\rho, j) \cdot \Delta(j; r, s) = 0$, and it leaves three evaluations of $\tilde{A}$: at r , at s , and at a third point that arises from the diagonal term.

We refine this check as follows. Instead of sampling s independently, the verifier sets $s_h = r_h^2$ for every coordinate h . Since the entries of A are bits, the Frobenius map gives $\tilde{A}(s, j) = \tilde{A}(r, j)^2$, so the first term of Δ does not need a separate evaluation. For the diagonal term, the verifier computes

$$c_h = 1 + r_h + r_h^2, \quad r'_h = \frac{r_h^3}{c_h}, \quad C(r) = \prod_h c_h,$$

resampling r if some $c_h = 0$. Then $\text{eq}(r, u)^3 = C(r) \cdot \text{eq}(r', u)$ for every Boolean u , and the collision polynomial collapses to

$$\Delta^*(j; r) = \tilde{A}(r, j)^3 + C(r) \cdot \tilde{A}(r', j).$$

The verifier samples $\rho \in \mathbb{K}^n$, and a sum-check argument over the cycle variable proves

$$0 = \sum_{j \in \{0,1\}^n} \text{eq}(\rho, j) \cdot \Delta^*(j; r),$$

leaving only two evaluations of $\tilde{A}$ rather than three. If any Boolean cycle contains a collision, $\Delta^*(\cdot; r)$ is a non-zero polynomial, so the random points r and ρ detect it except with Schwartz–Zippel error.

BinaryShout. BinaryShout proves reads from a public read-only table $\text{Val} : \{0,1\}^a \rightarrow \mathbb{K}^c$ with c columns. On Boolean cycles j , the value returned by a read and the bits of the address read are the virtual polynomials $\tilde{\text{rv}}_h(j) = \sum_k \text{ra}(k, j) \cdot \text{Val}_h(k)$ and $\tilde{\text{raf}}_t(j) = \sum_k k_t \cdot \text{ra}(k, j)$. Given claimed evaluations of these polynomials at a cycle point r_{cycle} , we check all of them with one sum-check argument. The verifier samples a non-zero α_0 , address coefficients $\alpha_1, \dots, \alpha_a$, and value coefficients $\gamma_1, \dots, \gamma_c$. Both parties then define the polynomial

$$\Phi(k) = \alpha_0 + \sum_{t=1}^a \alpha_t \cdot k_t + \sum_{h=1}^c \gamma_h \cdot \tilde{\text{Val}}_h(k).$$

The prover proves that

$$\alpha_0 + \sum_{t=1}^a \alpha_t \cdot \tilde{\text{raf}}_t(r_{\text{cycle}}) + \sum_{h=1}^c \gamma_h \cdot \tilde{\text{rv}}_h(r_{\text{cycle}}) = \sum_{k \in \{0,1\}^a} \tilde{\text{ra}}(k, r_{\text{cycle}}) \cdot \Phi(k).$$

Since Φ is a random linear combination of the address bits and the table's columns, this one identity checks every address-bit and value claim at once. The α_0 term does one more job: it contributes α_0 on the left and $\alpha_0 \cdot \sum_k \text{ra}(k, r_{\text{cycle}})$ on the right, so the identity also forces $\sum_k \text{ra}(k, \cdot) = 1$, which together with the collision check means that the address has Hamming weight exactly one. BinaryShout therefore needs no separate non-emptiness check. Finally, we reduce the remaining evaluation of the full address $\tilde{\text{ra}}$ to d evaluations of its components with the address-virtualization sum-check argument.

BinaryTwist. BinaryTwist proves that a mutable memory with K cells of $L = 2^\ell$ bits each, initialized to a public state M_0 , is accessed consistently when every cycle performs p reads followed by one write. Besides the address components of the p reads and of the write, the prover commits to one more polynomial, the *write increments* $\text{lnc} : \{0,1\}^\ell \times \{0,1\}^n \rightarrow \mathbb{B}$, where $\text{lnc}(\cdot, j)$ is the XOR of the old and new contents of the cell written in cycle j . Everything else is virtual. In particular, the complete memory history $\text{Val}(k, o, j)$, bit o of cell k at the start of cycle j , has $K \cdot L \cdot T$ entries and is never materialized, let alone committed.

On Boolean points, the increments determine it, and it in turn determines the values read and written:

$$\begin{aligned}\widetilde{\text{Val}}(k, o, j) &= \widetilde{\text{M}}_0(k, o) + \sum_{j' \in \{0,1\}^n} \widetilde{\text{LT}}(j', j) \cdot \widetilde{\text{wa}}(k, j') \cdot \widetilde{\text{Inc}}(o, j'), \\ \widetilde{\text{rv}}_s(o, j) &= \sum_{k \in \{0,1\}^a} \widetilde{\text{ra}}_s(k, j) \cdot \widetilde{\text{Val}}(k, o, j), \quad s \in [p], \\ \widetilde{\text{wv}}(o, j) &= \sum_{k \in \{0,1\}^a} \widetilde{\text{wa}}(k, j) \cdot (\widetilde{\text{Val}}(k, o, j) + \widetilde{\text{Inc}}(o, j)),\end{aligned}$$

where $\text{LT}(j', j) = 1$ exactly when $j' < j$. The first identity reconstructs a cell from its initial value and the increments written to it in earlier cycles; the second selects the contents of the cells read, before the current write; the third gives the contents of the written cell after the write. Given claimed evaluations of $\widetilde{\text{rv}}_s$ and $\widetilde{\text{wv}}$ at a cycle point and a value point, we run one sum-check argument per identity. The second and third reduce the read and write values to evaluations of the address polynomials, of $\widetilde{\text{Val}}$, and of $\widetilde{\text{Inc}}$; the first reduces the evaluation of $\widetilde{\text{Val}}$ to further evaluations of $\widetilde{\text{wa}}$ and $\widetilde{\text{Inc}}$, with the verifier evaluating $\widetilde{\text{M}}_0$ and $\widetilde{\text{LT}}$ on its own; the ordinary address bits are handled as in BinaryShout; and address virtualization reduces every full-address evaluation to evaluations of the components. At this point, every remaining claim is an evaluation of a committed polynomial. Note that, unlike in BinaryShout, the three identities above are homogeneous in the address polynomials, so an all-zero address would satisfy them trivially. BinaryTwist therefore proves, with a separate sum-check argument, that every row of every address component sums to one; together with the collision check, this gives Hamming weight exactly one.

Definition 7 (Memory-checking relations). *An instance of $\mathcal{R}_{\text{BinaryShout}}$ consists of the public table Val , claimed evaluations of $\widetilde{\text{raf}}_t$ ($t \in [a]$) and $\widetilde{\text{rv}}_h$ ($h \in [c]$) at a constant number of cycle points, and, at each such point r_{cycle} , the claim $\sum_k \widetilde{\text{ra}}(k, r_{\text{cycle}}) = 1$. Its witness consists of the components $\text{ra}_1, \dots, \text{ra}_d$. It holds if every row $\text{ra}_i(\cdot, j)$ has Hamming weight at most one and every claim is correct.*

An instance of $\mathcal{R}_{\text{BinaryTwist}}$ consists of K, L, T, p, d , the initial memory M_0 , and claimed evaluations of the address bits of every read and of the write (defined as for BinaryShout), of $\widetilde{\text{rv}}_s$ ($s \in [p]$), and of $\widetilde{\text{wv}}$ at a common cycle point and value point. It may also request one evaluation of $\widetilde{\text{Inc}}$ at a separate point. Its witness consists of the address components of the p reads and of the write, and Inc . It holds if every row of every address component has Hamming weight exactly one and every claim is correct, where $\widetilde{\text{Val}}$, $\widetilde{\text{rv}}_s$, and $\widetilde{\text{wv}}$ are given by the identities above.

Lemma 2 (BinaryShout and BinaryTwist reductions [55]). *Assume that the address components all have the same width (i.e., $d \mid a$ as above). Then:*

1. *BinaryShout is a public-coin RoK*

$$\mathcal{R}_{\text{BinaryShout}} \longrightarrow (\mathcal{R}_{\text{veval}}^{\mathbb{B}})^{3 \cdot d}.$$

It uses $O(\log T + \log K)$ rounds, communicates $O(d \cdot \log T + \log K)$ field elements, and has knowledge error $O(d \cdot (\log T + \log K)) / |\mathbb{K}|$.

2. *BinaryTwist is a public-coin RoK*

$$\mathcal{R}_{\text{BinaryTwist}} \longrightarrow (\mathcal{R}_{\text{veval}}^{\mathbb{B}})^{4 \cdot (p+1) \cdot d + \mathbf{1}_{d=1} + 2 + \mathbf{1}_{\text{inc}}},$$

where $\mathbf{1}_{\text{inc}} \in \{0, 1\}$ indicates whether the instance requests a separate increment evaluation, and $\mathbf{1}_{d=1}$ accounts for a single address component: in that case the $p + 2$ full-address claims (one per read and two for the write address) are already component evaluations and address virtualization is unnecessary, whereas for $d > 1$ the two write-address claims share a product term and virtualization yields $(p + 1) \cdot d$ component evaluations. It uses $O(\log T + \log K)$ rounds, communicates $O(d \cdot \log T + \log K + (p + 1) \cdot d)$ field elements, and has knowledge error $O(d \cdot \log T + (p + 1) \cdot \log K + \log L) / |\mathbb{K}|$.

Note that, for a constant number of reads per cycle, each reduction runs a logarithmic number of rounds, communicates $O(d \cdot \log T + \log K)$ field elements, and leaves a constant number of evaluation claims per address component. The prover's cost is linear in T .

5 The Magnetar construction

In this section, we describe Magnetar, our SNARK for RV64I executions, which we build from the reductions of Section 4. In a nutshell: (1) we arithmetize each cycle's transition as two SIMD R1CS relations over one bit-valued witness; (2) we use Virtual Spartan to reduce them to one evaluation claim per witness segment; (3) we discharge the claims on virtual segments, and check sub-word stores, with BinaryTwist, BinaryShout, and BinaryRouter; and (4) we batch everything that remains into a single committed evaluation with MosaicPCS. In Section 5.5, we put these pieces together into the complete protocol and prove its security.

Definition 8 (Magnetar execution). *Let $\mathcal{P}$ be an RV64I program. For public input inp , private auxiliary input w , and public output out , the relation $\mathcal{R}_{\mathcal{P}}$ contains $((\text{inp}, \text{out}), w)$ exactly when $\mathcal{P}(\text{inp}, w) = \text{out}$.*

Recall from Section 2 that Magnetar expands each instruction of $\mathcal{P}$ into one or more virtual instructions during public preprocessing, and that a micro-PC identifies each of them. We write ROM for the resulting public bytecode table, indexed by micro-PC. The execution that the prover proves is a sequence of virtual cycles, each of which executes one virtual instruction.

5.1 Arithmetizing CPU semantics with two SIMD R1CS relations

We could enforce every CPU check with one SIMD R1CS relation over $\mathbb{B}$, but some word-wide operations are far cheaper when packed over $\mathbb{K}$. Consider a word-wide selection, i.e., $y_i = cx_i$ for bits $c, x_i, y_i \in \mathbb{B}$ and $0 \leq i < b \leq 128$. Over $\mathbb{B}$ this costs one product constraint per bit. Over $\mathbb{K}$, with a $\mathbb{B}$ -basis $\beta_0, \dots, \beta_{127}$, the same b constraints are equivalent to the single field constraint

$$c \cdot \left(\sum_{i=0}^{b-1} \beta_i \cdot x_i \right) = \sum_{i=0}^{b-1} \beta_i \cdot y_i.$$

We therefore use a SIMD R1CS relation over $\mathbb{K}$ for such packed product families and for packed linear checks, and a SIMD R1CS relation over $\mathbb{B}$ for the remaining bit constraints (e.g., the bitwise logic of the datapath). Together, they enforce fetch-decode-execute semantics. Each SIMD instance covers a fixed number of consecutive cycles, so state carried from one cycle to the next, such as the micro-PC, is chained by ordinary constraints within an instance. At instance boundaries, the binary relation additionally contains cross-step constraints relating the state of instance j to that of instance $j-1$; Spartan supports these through the standard `next` multilinear extension [53], and the public initial and final states are bound at the first and last instances. The two relations defer the memory-checking claims and the routing identities for shifts and sub-word stores to later reductions.

Formally, the two relations share one witness. For SIMD instance j , let $\mathbf{w}_j$ be its witness and $\mathbf{x}_j$ its public inputs and outputs, and let W be the bit-valued matrix whose row j is $\mathbf{w}_j$. With $\mathbf{z}_j = (\mathbf{w}_j, 1, \mathbf{x}_j)$, we require, for every j ,

$$\begin{aligned} (A_1 \mathbf{z}_j) \circ (B_1 \mathbf{z}_j) &= C_1 \mathbf{z}_j && \text{over } \mathbb{B}, \\ (A_2 \mathbf{z}_j) \circ (B_2 \mathbf{z}_j) &= C_2 \mathbf{z}_j && \text{over } \mathbb{K}. \end{aligned}$$

Here A_1, B_1 , and C_1 have entries in $\mathbb{B}$, whereas A_2, B_2 , and C_2 have entries in $\mathbb{K}$, which is what lets one row of the second relation pack a word-wide check as in the example above. Because both relations constrain the same W , a prover cannot satisfy them with two different witnesses. Section 5.3 shows how Virtual Spartan reduces both relations at the cost of one additional outer sum-check argument, with a single inner sum-check argument over W .

Remark 4. For simplicity, we omit the cross-step constraints and the binding of the public initial and final states from our descriptions. Virtual Spartan enforces them with the `next` multilinear extension and one additional sum-check argument over the instance coordinates, and Lemma 3 still holds.

5.2 MosaicPCS

Every reduction in this section ends with evaluation claims about bit-valued polynomials: the direct segments of the R1CS witness, and the polynomials that the auxiliary reductions commit to, such as the one-hot address components and the RAM increments. There are many of these polynomials, and they have

different sizes. If we committed to each of them separately, each would also need its own PCS evaluation argument, and with a hash-based PCS those evaluation arguments would dominate the proof length. We avoid this with MosaicPCS: we place all of the polynomials on disjoint subcubes of one larger polynomial, commit to that polynomial once, and prove every evaluation claim with a single PCS evaluation argument.

Concretely, let $p_1, \dots, p_N$ be the polynomials listed above, the direct segments and the auxiliary polynomials, with $p_i : \{0, 1\}^{\ell_i} \rightarrow \mathbb{B}$. We choose $\ell \geq \max_i \ell_i$ and Boolean suffixes $s_i \in \{0, 1\}^{\ell - \ell_i}$ such that the subcubes $\{x \parallel s_i : x \in \{0, 1\}^{\ell_i}\}$ are pairwise disjoint, and we define $G : \{0, 1\}^\ell \rightarrow \mathbb{B}$ to agree with p_i on the i th subcube, i.e., $G(x \parallel s_i) = p_i(x)$ for $x \in \{0, 1\}^{\ell_i}$, and to be zero outside all subcubes. The prover commits once, to the packed representation P_G of G , and sends $\text{cm}_G = \text{Commit}(P_G)$.

The point of this layout is that a claim about any p_i lifts to a claim about G for free. Fixing the suffix variables of $\tilde{G}$ to the Boolean point s_i restricts G to the i th subcube, where it equals p_i , so for every $\mathbf{r} \in \mathbb{K}^{\ell_i}$,

$$\tilde{G}(\mathbf{r} \parallel s_i) = \tilde{p}_i(\mathbf{r}). \quad (4)$$

In other words, to turn a claim $\tilde{p}_i(\mathbf{r}_i) = v_i$ into the claim $\tilde{G}(\mathbf{r}_i \parallel s_i) = v_i$, the verifier simply appends the public suffix s_i to the point. Once every claim is a claim on $\tilde{G}$, two standard steps remain.

- *Batching.* We now have several claims $\tilde{G}(\mathbf{r}_i \parallel s_i) = v_i$ about the same polynomial at different points, and we reduce them to one claim $\tilde{G}(\mathbf{r}_G) = v_G$ at a single point $\mathbf{r}_G \in \mathbb{K}^\ell$. The verifier samples $\alpha_i \leftarrow \mathbb{K}$ for every claim, and the two parties run the sum-check protocol for

$$\sum_i \alpha_i \cdot v_i = \sum_{x \in \{0, 1\}^\ell} \left(\sum_i \alpha_i \cdot \text{eq}(\mathbf{r}_i \parallel s_i, x) \right) \cdot G(x),$$

whose summand has degree two in each variable; $\mathbf{r}_G$ is the point chosen over its rounds, and v_G is the prover's claimed value of $\tilde{G}(\mathbf{r}_G)$. This step is a public-coin RoK with knowledge error $O(\ell)/|\mathbb{K}|$, and it sends no commitments.⁴

- *Ring switching.* $\tilde{G}$ is the multilinear extension of a bit-valued vector, whereas the PCS commits to P_G , the packing of G into elements of $\mathbb{K}$. Ring switching [13] is a public-coin RoK that reduces the claim $\tilde{G}(\mathbf{r}_G) = v_G$ to a claim $\tilde{P}_G(\mathbf{r}^*) = v^*$ on P_G , again without sending commitments.

⁴ This sum-check argument does not reintroduce the adapter that JaggedPCS [23] needs and MosaicPCS avoids. The adapter is a per-component sum-check argument that turns a claim on p_i into a claim on the committed polynomial, and Equation (4) does this for free. Batching several claims on one committed polynomial into a single evaluation is a separate, standard step that every scheme with more than one claim performs, and JaggedPCS performs it as well.

We write Π_{MosRS} for the composition of the lift and these two steps. It is a RoK from $(\mathcal{R}_{\text{veval}}^{\mathbb{B}})^*$ to $\mathcal{R}_{\text{veval}}$, and its knowledge error is the sum of the two errors above. The verifier then pairs the output $(\mathbf{r}^*, v^*)$ with cm_G to form the committed evaluation claim $(\text{cm}_G, \mathbf{r}^*, v^*)$, and the prover proves it with one PCS evaluation argument.

5.3 Virtual Spartan

Recall that the two SIMD R1CS relations of Section 5.1 share the witness W , and that we want to commit to only part of it. If we ran the SIMD Spartan reduction on each relation separately, we would end up with two evaluation claims about all of W , and neither could be discharged. A PCS cannot check a claim about W because not all of W is committed, and the auxiliary arguments cannot either, because each of them accepts claims about a single segment of W , not about W as a whole.

Virtual Spartan addresses this. Fix a public layout that divides W into segments $W_1, \dots, W_N$. We include some of the segments in the MosaicPCS polynomial G and call them *direct*. The remaining segments are *virtual*: the R1CS relations read them, but no commitment contains them. Virtual Spartan first reduces both SIMD R1CS relations jointly to a single evaluation claim $\widetilde{W}(\mathbf{r}) = v_W$ on the shared witness. Then the prover sends one evaluation claim $\widetilde{W}_i(\mathbf{r}_i) = v_i$ per segment, and the verifier checks that these claims are consistent with the claim on W . Because every segment occupies an aligned subcube of W , this consistency check is a fixed linear identity in the v_i . The per-segment claims are then discharged in two ways. The claims on direct segments go to MosaicPCS and are proven by the final PCS evaluation argument, and the claim on each virtual segment goes to the auxiliary reduction that owns that segment.

Construction 2 (Virtual Spartan). Suppose the two SIMD R1CS relations have the same padded SIMD dimension q and size bound m , index the entries of $\mathbf{z}$ identically, and share the bit-valued witness W . Fix its public segment layout. The reduction proceeds as follows.

1. Run the BinarySpartan outer sum-check argument for the binary relation and the SIMD Spartan outer sum-check argument for the $\mathbb{K}$ -coefficient relation. Run the two sum-check arguments in lockstep. In the rounds that bind the SIMD instance variables, the verifier sends one challenge that both sum-check arguments use, so that both end at the same instance point $\mathbf{r}_{\text{inst}}$; in the rounds that bind the constraint row variables, each sum-check argument receives its own challenge, ending at $\mathbf{r}_{\text{row},t}$ for relation $t \in \{\text{bin}, \text{fld}\}$.⁵
2. Each outer argument ends with three terminal values. Denote the six values by $v_{t,M}$ for $t \in \{\text{bin}, \text{fld}\}$ and $M \in \{A, B, C\}$.

⁵ Sharing $\mathbf{r}_{\text{inst}}$ is what allows a single inner sum-check argument to serve both relations in Step 4. The row challenges are not shared because the two outer sum-check arguments use different univariate-skip parameters [22].

3. After the six values are fixed, the verifier samples independent coefficients $\alpha_{t,M} \leftarrow \mathbb{K}$ and sets

$$v^* = \sum_{t,M} \alpha_{t,M} v_{t,M}.$$

4. Run one inner sum-check argument for

$$v^* = \sum_y \left(\sum_{t,M} \alpha_{t,M} \cdot \widetilde{M}_t(\mathbf{r}_{\text{row},t}, y) \right) \cdot \widetilde{\mathbf{z}}(\mathbf{r}_{\text{inst}}, y),$$

where M_t is the corresponding public base matrix and $\widetilde{\mathbf{z}}(\mathbf{r}_{\text{inst}}, \cdot)$ is the shared vector $\mathbf{z}$ folded at the common SIMD instance point. The verifier chooses $\mathbf{r}_y$ one coordinate at a time over the course of this sum-check argument. Set $\mathbf{r}_{\text{col}} = \mathbf{r}_y[1..]$. The prover supplies $v_W = \widetilde{W}(\mathbf{r}_{\text{inst}}, \mathbf{r}_{\text{col}})$, and the verifier uses Equation (1) to check the resulting identity.

5. The layout places segment W_i on an aligned subcube of the column space: its columns are those whose binary index ends in a fixed Boolean suffix $\mathbf{b}_i$, and the remaining index bits are the segment's local coordinates. Split the column point accordingly as $\mathbf{r}_{\text{col}} = (\mathbf{r}_i, \mathbf{q}_i)$, where $\mathbf{r}_i$ holds the coordinates local to W_i and $\mathbf{q}_i$ those of its suffix. Since W agrees with W_i on that subcube and the subcubes partition the columns,

$$\widetilde{W}(\mathbf{r}_{\text{inst}}, \mathbf{r}_{\text{col}}) = \sum_i \text{eq}(\mathbf{q}_i, \mathbf{b}_i) \cdot \widetilde{W}_i(\mathbf{r}_{\text{inst}}, \mathbf{r}_i).$$

6. $\mathcal{P} \rightarrow \mathcal{V}$: v_i for every segment, claimed to equal $\widetilde{W}_i(\mathbf{r}_{\text{inst}}, \mathbf{r}_i)$; padding segments are identically zero, so $v_i = 0$ for them.
7. $\mathcal{V}$ checks that $v_W = \sum_i \text{eq}(\mathbf{q}_i, \mathbf{b}_i) \cdot v_i$ and aborts otherwise.
8. $\mathcal{P}$ and $\mathcal{V}$ output the $\mathcal{R}_{\text{veval}}^{\mathbb{B}}$ instance $(\mathbf{r}_{\text{inst}} \parallel \mathbf{r}_i, v_i)$ for every non-padding segment W_i ; $\mathcal{P}$ additionally outputs W_i .

Lemma 3 (Virtual Spartan reduction). *Let $\mathcal{R}_{\text{dual}}$ be the relation whose instances are the public inputs $(\mathbf{x}_j)_{j \in [q]}$ and whose witnesses are bit-valued matrices W such that both SIMD R1CS relations of Section 5.1 hold with the rows of W as their witnesses. Construction 2 is a public-coin RoK*

$$\mathcal{R}_{\text{dual}} \longrightarrow \prod_i \mathcal{R}_{\text{veval}}^{\mathbb{B}},$$

where the product ranges over the non-padding segments and the i th factor has instance $(\mathbf{r}_{\text{inst}} \parallel \mathbf{r}_i, v_i)$ and witness W_i . Its knowledge error is $O(\log q + \log m)/|\mathbb{K}|$.

Proof (sketch). Completeness follows from the two outer identities, the batched inner identity, and the subcube identity of Step 5. For knowledge soundness,

consider the steps in reverse. Step 5 is an exact identity, so segment polynomials W_i satisfying the output claims concatenate, along the public layout, to a W with $\widetilde{W}(\mathbf{r}_{\text{inst}}, \mathbf{r}_{\text{col}}) = v_W$. The inner sum-check argument binds this evaluation to the batched terminal claim, and since the batching coefficients $\alpha_{t,M}$ are sampled after the six terminal values are fixed, a false set of terminal values survives the batch with probability at most $1/|\mathbb{K}|$. What remains is, for each of the two relations, the SIMD Spartan reduction of Lemma 1, except that the two inner sum-check arguments are batched into one. Its extractor, applied to this W , therefore yields a witness for both relations, i.e., for $\mathcal{R}_{\text{dual}}$. The knowledge error is the sum of the errors of the two Spartan reductions and of the batching step. $\square$

Each output claim of Virtual Spartan is then discharged by the argument that knows how to check it. The direct claims concern committed polynomials, so they wait for MosaicPCS. Each virtual claim concerns a polynomial that one of the auxiliary arguments defines, and that argument replaces the claim with claims about the polynomials it commits to; Section 5.5 spells out the order of the arguments and the claims linked between them.

5.4 BinaryRouter

Two RV64 operations have no compact R1CS encoding. A shift moves every bit of a source word to a new position, and a sub-word store replaces some byte lanes of a 64-bit RAM word while preserving the rest. We observe that both operations are *bit routings*: every output bit is either zero or a copy of one input bit, and which input bit is fixed by the instruction (and, for a store, by the low address bits) rather than by the values being moved. We devise a family of sum-check arguments for exactly this structure and call it BinaryRouter.

A routing is an identity of the form

$$Y(o, j) = \sum_{u, h} X(u, j) \cdot Q(h, j) \cdot R(o, u, h),$$

where Y is the output table, X collects the candidate source bits, $Q(\cdot, j)$ is a one-hot selector (or zero) naming the routing in use in cycle j , and R is a public, sparse routing structure fixed during preprocessing, like an R1CS matrix.

BinaryRouter reduces an evaluation claim on the output Y to evaluation claims on the inputs X and Q (and, for stores, on the low address bits) with a single sum-check argument over the cycle, source, and routing indices; the verifier evaluates R itself, which is cheap because R is public and sparse. The reduction does not commit to any additional polynomials.

Lemma 4 (BinaryRouter reduction). *Fix a public routing structure R . BinaryRouter is a public-coin RoK that reduces an evaluation claim on Y , with witnesses X and Q satisfying the routing identity, to a constant number of $\mathcal{R}_{\text{veval}}^{\mathbb{B}}$ instances on X and Q . If R also takes the per-cycle byte offset $\mathbf{a}(j)$, as for stores (Section 5.4), the witness also includes the offset bits, and the output also includes*

one such instance per offset bit. For fixed word, routing, and address widths, it has $O(\log T)$ rounds and knowledge error $O(\log T)/|\mathbb{K}|$.

Proof (sketch). Substituting the routing identity into $\tilde{Y}(\mathbf{r})$ turns the claim into a sum over Boolean (u, h, j) of a product of X , Q , $\tilde{R}$, and $\tilde{\mathbf{e}}\mathbf{q}$. The sum-check protocol reduces this sum to evaluations of X and Q at a random point, and the verifier evaluates $\tilde{R}$ and $\tilde{\mathbf{e}}\mathbf{q}$ itself. If R takes $\mathbf{a}(j)$, the factor $\tilde{R}$ in the summand takes the multilinear extensions of the offset bits as its last three arguments. The prover then sends their values at the random cycle point, the verifier evaluates $\tilde{R}$ at these values, and the values become the claims on the offset bits. The summand has constant degree in every variable, so the knowledge error is $O(\log T)/|\mathbb{K}|$. $\square$

We now describe the two instances of BinaryRouter in Magnetar: (1) shifts and (2) sub-word stores.

(1) BinaryRouter for shifts.

A shift moves the bits of a source word to new positions and fills the remaining positions with zero or, for an arithmetic right shift, with the sign bit. Which bit goes where is fixed by the bytecode row (the shift kind and, for immediate shifts, the amount). In the notation of the routing identity, $X(u, j)$ collects the candidate source bits in cycle j , including one source fixed to zero; $Y(o, j)$ is bit o of the shift output; h ranges over the possible routings; $Q_{\text{sh}}(h, j)$ is one for the routing used in cycle j and zero otherwise, and is identically zero on a cycle without a shift; and $R_{\text{sh}}(o, u, h)$ is one exactly when routing h maps source u to output o . The shift witness must satisfy

$$Y(o, j) = \sum_{u, h} X(u, j) \cdot Q_{\text{sh}}(h, j) \cdot R_{\text{sh}}(o, u, h). \quad (5)$$

When the selector is one-hot, the right-hand side returns the source prescribed for output o ; when it is zero, the identity forces $Y(o, j) = 0$. There are a few hundred routings, so rather than storing Q_{sh} as one wide one-hot field, the bytecode row stores the routing index in two shorter one-hot fields, and $Q_{\text{sh}}(h, j) = Q_{\text{lo}}(h_{\text{lo}}, j) \cdot Q_{\text{hi}}(h_{\text{hi}}, j)$ for $h = (h_{\text{lo}}, h_{\text{hi}})$. This raises the degree of the sum-check argument in each cycle variable by one; other aspects remain the same.

The shift output Y is a virtual segment of the dual R1CS witness W , so its evaluation claim comes from Virtual Spartan, and the reduction discharges that claim to claims on the inputs. The candidate source bits lie in direct segments of W , so MosaicPCS proves their evaluations together with everything else. The two factors of Q_{sh} are fields of the bytecode row fetched in each cycle, so BinaryShout for bytecode reduces the evaluation claims on them to a claim on the committed fetch addresses, using the public table ROM (Section 5.5). Shifts whose amount comes from a register do not have a static routing, so we expand them into a short, fixed sequence of virtual instructions, each of which is a static routing.

(2) BinaryRouter for sub-word stores.

A byte, halfword, or word store replaces part of a 64-bit RAM word and preserves the rest, and a doubleword store replaces all of it. Recall that BinaryTwist for RAM commits to Inc , the bitwise XOR of the old and new RAM words, and proves that RAM evolves by these increments. It does not prove that each increment performs the store that the bytecode row asked for. We prove this with BinaryRouter.

The new word of a store is a routing of the old RAM word and the store-source word, and we apply the routing identity with Inc as its output. Outside the bytes being written, the increment is zero. Inside them, the new bit is the store-source bit, so the increment bit is the XOR of the old RAM bit and the store-source bit, and the routing structure feeds that output from both source bits, whose sum over $\mathbb{B}$ is their XOR. Which positions are inside depends only on the store width, which the bytecode row fixes, and on the byte offset, i.e., the three low address bits, which vary per cycle. We therefore take the old RAM word and the store-source word as the sources X , a one-hot selector Q_{st} over the four widths from the bytecode row, and a routing structure $R_{\text{st}}(o, u, h, \mathbf{a})$ that takes the byte offset $\mathbf{a} \in \mathbb{B}^3$ as three additional Boolean arguments, so that it stays public although the offset varies. With $\mathbf{a}(j)$ the offset in cycle j , the store witness must satisfy

$$\text{Inc}(o, j) = \sum_{u, h} X(u, j) \cdot Q_{\text{st}}(h, j) \cdot R_{\text{st}}(o, u, h, \mathbf{a}(j)). \quad (6)$$

Both words are virtual segments of W (BinaryTwist for RAM owns the old word, and BinaryTwist for registers owns the store-source word), so we add to W a direct segment that holds a copy of the two words, and the R1CS binds the copy to the originals in every cycle. X is this copy, so, as for shifts, MosaicPCS proves its evaluations.

Because R_{st} depends on the cycle through $\mathbf{a}(j)$, the summand of the sum-check argument contains the factor $\tilde{R}_{\text{st}}(\mathbf{r}_o, u, h, \tilde{\mathbf{a}}(j))$, where $\mathbf{r}_o$ is the output point of the claim and $\tilde{\mathbf{a}}$ collects the multilinear extensions of the three offset bits. At the end of the sum-check argument, the prover sends the values of $\tilde{\mathbf{a}}$ at the cycle point, which the verifier chooses one coordinate at a time over the course of the argument, and the verifier evaluates $\tilde{R}_{\text{st}}$ at these values itself. The offset bits are the low bits of the store address, which the R1CS computes in a direct segment of W , so the three values become evaluation claims that MosaicPCS proves.

Finally, unlike for shifts, the output claim does not come from Virtual Spartan. Inc is committed, because BinaryTwist for RAM proves the RAM history from it, so the claim that starts this instance is an evaluation of Inc at a random point that the verifier chooses, and the PCS checks this claim against the commitment. The remaining aspects are similar to BinaryRouter for shifts.

5.5 The complete protocol

We now assemble the components into the complete protocol.

Construction 3 (Magnetar). For a trace length T , KeyGen derives from the program $\mathcal{P}$ the bytecode table ROM, the two SIMD R1CS structures $\mathbf{s}_1$ over $\mathbb{B}$ and $\mathbf{s}_2$ over $\mathbb{K}$, the witness layout $\mathcal{L}$, the routing structures R_{sh} and R_{st} , and the MosaicPCS layout. The online phase proceeds as follows.

1. The prover forms the shared witness W and uses MosaicPCS to commit to its direct segments together with the auxiliary polynomials of the memory-checking arguments, and sends the resulting commitment cm_G to the verifier. This is the protocol's only commitment.
2. The prover and verifier run Virtual Spartan (Construction 2), which reduces the satisfaction of $\mathbf{s}_1$ and $\mathbf{s}_2$ on W to one evaluation claim per witness segment.
3. The prover and verifier run BinaryTwist for registers on the virtual register claims from Step 2. It reduces them to evaluation claims on the committed one-hot register addresses and increments, and to claims on the register indices, which Step 6 checks against the bytecode.
4. The prover and verifier run BinaryTwist for RAM on the virtual RAM claims from Step 2. It reduces them to evaluation claims on the committed one-hot RAM addresses and increments. In addition, the verifier chooses a random point and the prover sends the evaluation of Inc there. This claim is both an output of this step, so that the PCS checks it against the commitment, and the input to BinaryRouter for sub-word stores in Step 5.
5. The prover and verifier run BinaryRouter for shifts on Virtual Spartan's claim on the shift output and BinaryRouter for sub-word stores on the increment claim of Step 4. Each reduces its claim to evaluation claims on the routing's sources, which are direct segments of W , and on the bytecode fields that define its selector, which Step 6 checks. The store instance also outputs evaluation claims on the three low address bits, which lie in a direct segment of W . The verifier evaluates the public routing structures R_{sh} and R_{st} itself.
6. The prover and verifier run BinaryShout for bytecode with $\text{Val} = \text{ROM}$ on the virtual bytecode claims from Step 2, the register-index claims from Step 3, and the selector claims from Step 5. All of these are read values of the bytecode lookup, and the reduction turns them into evaluation claims on the committed one-hot fetch addresses.
7. Every claim now outstanding, namely the direct claims from Step 2 and the claims output by Steps 3–6 that Step 6 has not consumed, is an evaluation claim on a component of G . The prover and verifier lift each to a claim on $\tilde{G}$, batch them, and apply ring switching as in Section 5.2, obtaining a single evaluation claim on P_G . The prover proves this claim against cm_G with one PCS evaluation argument, and the verifier accepts if and only if that argument does.

Lemma 5 (Execution arithmetization). *Let $\mathcal{P}$ be a program, T a trace length, and $\text{ROM}, s_1, s_2, R_{\text{sh}},$ and R_{st} the corresponding outputs of KeyGen . For a public input inp and output out , there exist a witness W and auxiliary polynomials that satisfy the two SIMD R1CS relations and the BinaryTwist, BinaryRouter, and BinaryShout relations of Steps 2–6 if and only if there is an auxiliary input w such that $\mathcal{P}(\text{inp}, w) = \text{out}$ within T cycles.*

Proof. If $\mathcal{P}(\text{inp}, w) = \text{out}$ within T cycles, the execution trace, padded to T cycles, and the memory-checking bookkeeping define all witnesses. Conversely, suppose that all relations hold. BinaryShout for bytecode ensures that each cycle fetches the row of ROM at its micro-PC. The two SIMD R1CS relations enforce, in every cycle, the state transition prescribed by that row, apart from the checks that they defer. BinaryTwist for registers and for RAM ensure that every read returns the most recently written value, and the two BinaryRouter instances enforce the shift and store semantics. Induction over cycles shows that the trace is the execution of $\mathcal{P}$ on inp and the private input w recorded in W , and the R1CS relations bind inp and out to the initial and final state. $\square$

Theorem 1. *Suppose that the PCS is transparent and succinct and that its evaluation argument is perfectly complete and public-coin with knowledge error ε_{PCS} . Let $\varepsilon_{\text{stat}}$ be the sum of the knowledge errors of the reductions in Steps 2–7, each of which is $O(\log T)/|\mathbb{K}|$ for fixed machine widths. Then Construction 3 is a transparent preprocessing SNARK for $\mathcal{R}_{\mathcal{P}}$, restricted to executions of at most T cycles, with knowledge error at most $\varepsilon_{\text{stat}} + \varepsilon_{\text{PCS}}$.*

Proof. Perfect completeness follows from the perfect completeness of the underlying RoKs and of the PCS evaluation argument. For knowledge soundness, observe that once cm_G is sent, Steps 2–7 are a sequential composition of public-coin reductions of knowledge: Virtual Spartan (Lemma 3); the BinaryTwist, BinaryRouter, and BinaryShout reductions (Lemmas 2 and 4); and the batching and ring-switching reduction of Section 5.2, followed by the PCS evaluation argument as the terminal argument of knowledge. Passing a direct claim through unchanged and lifting a claim on a component of G to a claim on $\tilde{G}$ are exact, so they add no error. By the composition theorem for reductions of knowledge [26], the protocol is an argument of knowledge for the relations of Lemma 5 with knowledge error at most $\varepsilon_{\text{stat}} + \varepsilon_{\text{PCS}}$, and Lemma 5 turns an extracted witness into an auxiliary input w with $\mathcal{P}(\text{inp}, w) = \text{out}$. For fixed machine widths, the proof consists of a constant number of succinct components, so it is succinct. No setup step uses a secret trapdoor, so the construction is transparent. Finally, every step is public-coin, so the Fiat–Shamir transformation yields a succinct non-interactive argument of knowledge in the random oracle model. $\square$

6 Implementation and evaluation

We implement Magnetar in Rust on top of BinarySpartan [51]. We instantiate the PCS with Ligerito [44].

#	Short name	Description
1	<code>fibonacci</code>	Fibonacci loop with wrapping 64-bit addition.
2	<code>hot-cell-increment</code>	Increments the same 64-bit counter in a loop.
3	<code>stream-copy</code>	Copies a 16-word buffer.
4	<code>pointer-chase</code>	Follows a 64-node pointer chain.
5	<code>binary-search</code>	Searches a sorted 64-entry table.
6	<code>radix-sort-u64</code>	Radix-sorts 16 unsigned 64-bit keys.
7	<code>lz4-decode</code>	Decodes a 17-byte LZ4 block to 32 bytes.
8	<code>varint-decode</code>	Decodes ten LEB128 values, one to ten bytes long.
9	<code>sha256</code>	Runs SHA-256 compression in software in a loop.
10	<code>keccak-f1600</code>	Runs Keccak-f1600 in software.
11	<code>sha256-merkle-path</code>	Hashes an eight-level SHA-256 Merkle path in software.
12	<code>erc20-transfer</code>	Moves a 256-bit amount between two balances.
13	<code>erc20-transfer-from</code>	Transfers with allowance checks.
14	<code>erc20-mixed-block</code>	Transfers among accounts; every 16th transfer fails.
15	<code>evm-erc20-transfer</code>	Runs a balance transfer in a small EVM interpreter.

Table 1. Application workloads used in our experimental evaluation.

6.1 Experimental setup

We run all experiments on the 12 performance cores of an M4 Max MacBook Pro with 128 GiB of memory, one prover process at a time. For every system, we report the median of three runs after one warm-up run. We report two metrics for Magnetar. *Proving time* covers proof generation. *End-to-end time* also includes guest execution and trace construction. We configure Magnetar for 120 bits of provable security: every round of its protocol has soundness error at most 2^{-120} , and this bound uses only proven proximity gaps for Reed–Solomon codes within the unique-decoding radius [5]. Section 6.6 evaluates Magnetar (opt), which includes optimizations on top of Magnetar; all other results are for Magnetar.

Workloads. We evaluate 15 application workloads spanning arithmetic, memory access, data processing, cryptographic kernels, and smart-contract execution. Table 1 provides details. The ERC-20 workloads exercise only the token logic. They do not verify signatures or check Keccak Merkle paths.

Baselines. We run Jolt-Dory, Jolt-Akita [11], SP1 Hypercube, ZisK, and Lineth on the same machine. We benchmark Jolt [25] at commit `7a052796e373`, SP1 Hypercube from the published v6.8.0 crates [59] (source revision `58c4aeadb50`), ZisK [45] at commit `b08d856c0f72`, and Lineth [35] at commit `b75b5ca955d8`.

The baselines target different security levels, and we compare the implementations as released. SP1 Hypercube targets 100 bits [59,58]. Jolt-Dory relies on the BN254 pairing group, which offers about 100 bits of security [4]. Jolt-Akita’s Module-SIS commitment targets 128 bits of security [11], but its sum-check arguments over a 128-bit field [25] leave the overall level at about 122 bits. ZisK states a 128-bit target [62]. Lineth’s prover commits with Vortex, whose hashing rests on the

#	Workload	Unit	Magnetar		Jolt-Dory	Jolt-Akita	SP1 Hypercube
			Proving	End-to-end	Proving	Proving	Proving
1	fibonacci	steps/s	1,454,419	1,326,305	97,189	149,936	33,701
2	hot-cell-increment	updates/s	1,386,435	1,242,999	119,034	221,880	38,056
3	stream-copy	words/s	1,212,866	1,109,764	66,836	111,308	28,404
4	pointer-chase	chases/s	2,183,642	1,951,046	94,095	153,723	33,023
5	binary-search	searches/s	102,997	93,019	5,274	11,562	2,203
6	radix-sort-u64	keys/s	13,172	11,968	997	2,282	468
7	lz4-decode	bytes/s	550,207	499,288	14,883	28,040	11,934
8	varint-decode	values/s	96,491	88,168	5,981	11,838	4,041
9	sha256	hashes/s	1,356	1,249	51	91	32
10	keccak-f1600	permutations/s	888	807	53	82	27
11	sha256-merkle-path	nodes/s	530	486	24	44	16
12	erc20-transfer	tx/s	111,236	101,016	8,262	16,834	3,193
13	erc20-transfer-from	tx/s	79,357	72,144	4,741	9,707	1,855
14	erc20-mixed-block	tx/s	94,416	86,009	7,240	15,027	2,707
15	evm-erc20-transfer	tx/s	21,665	19,418	966	2,057	446
Proof size (KiB)			599	599	89	91	2,766
Verifier time (ms)			13	13	61	8	63
Peak RSS (GiB)			6	6	2	2	26

Table 2. Proving throughput in application-level operations per second. The proof-size and verifier-time rows give the largest per-workload value, and the peak-RSS row gives the largest value over all runs, where peak RSS is the resident memory of the prover process. All systems prove the hash workloads (rows 9–11) in software, without precompiles.

Ring-SIS lattice assumption; its code declares a 128-bit target, but we did not verify the resulting level [35]. Magnetar’s 120 bits are thus above the targets of SP1 Hypercube and Jolt-Dory, comparable to Jolt-Akita’s overall level, and below ZisK’s stated target.

Each system counts cycles differently (Magnetar executes RV64I, Jolt executes RV64IMAC, and SP1 Hypercube counts instructions differently again), so cycle counts are not comparable across systems. We therefore fix, for every system and workload, the largest whole number of application-level operations that fits in a fixed budget of the system’s own cycles, and compare completed operations per second. Every calibrated baseline run fills at least 98.8% of its budget. For calibration, the counted Fibonacci step takes 6 Magnetar virtual cycles, 5 Jolt cycles, and about 6 SP1 instructions.

Jolt’s prover times do not include tracing. SP1 Hypercube’s prover times cover its core prove phase and do not include setup or recursive compression. Both are therefore comparable to Magnetar’s proving time.

6.2 Core zkVM speed on sample applications

Table 2 reports proving throughput for Magnetar, Jolt-Dory, Jolt-Akita, and SP1 Hypercube on every workload at 2^{22} cycles. Section 6.3 covers ZisK and Lineth.

Prover speed. At 2^{22} cycles, Magnetar proves 7.2–8.7 million cycles per second across the 15 workloads (Table 3). Its per-workload proving speedup is $12\text{--}37\times$

#	Workload	Trace length			
		2^{20}	2^{22}	2^{23}	2^{24}
1	<code>fibonacci</code>	7.54	8.73	8.90	8.83
2	<code>hot-cell-increment</code>	7.18	8.32	8.56	8.37
3	<code>stream-copy</code>	6.53	7.51	7.60	7.41
4	<code>pointer-chase</code>	7.43	8.74	9.00	8.78
5	<code>binary-search</code>	7.00	8.06	8.34	7.97
6	<code>radix-sort-u64</code>	6.43	7.19	7.45	7.13
7	<code>lz4-decode</code>	6.62	7.62	7.37	7.26
8	<code>varint-decode</code>	6.64	7.68	7.73	7.38
9	<code>sha256</code>	6.53	7.55	7.68	7.31
10	<code>keccak-f1600</code>	6.36	7.33	7.34	6.92
11	<code>sha256-merkle-path</code>	6.45	7.31	7.40	6.94
12	<code>erc20-transfer</code>	6.86	8.01	8.06	7.64
13	<code>erc20-transfer-from</code>	6.62	7.70	7.79	7.29
14	<code>erc20-mixed-block</code>	6.70	7.66	7.66	7.29
15	<code>evm-erc20-transfer</code>	6.66	7.80	7.84	7.46

Table 3. Magnetar’s proving rate on each workload, in million cycles per second. The rate is the trace length divided by the proving time.

over Jolt-Dory, 6–20 $\times$ over Jolt-Akita, and 24–66 $\times$ over SP1 Hypercube, with geometric means of about 18 $\times$, 9.6 $\times$, and 39 $\times$. Even Magnetar’s end-to-end throughput, which includes guest execution, is about 8.7 $\times$ Jolt-Akita’s proving throughput in geometric mean over the 15 workloads. On `erc20-mixed-block`, Magnetar’s proving rate rises with trace length, from 1.9 million cycles per second at 2^{16} to 7.7 million at 2^{22} (Section 6.4).

The speedup varies across workloads because it is the product of two ratios: Magnetar’s proving rate in cycles per second over the baseline’s, and the baseline’s cycles per operation over Magnetar’s. The first ratio varies modestly, since Magnetar’s rate stays within 7.2–8.7 million cycles per second. The second depends on how each system’s instruction set handles the workload. Jolt’s trace, for example, has 1.4 times as many cycles per SHA-256 hash as Magnetar’s but only 0.8 times as many per sorted key, so Magnetar’s lead over Jolt-Akita is about 15 $\times$ on `sha256` but about 6 $\times$ on `radix-sort-u64`.

Memory. Further space optimizations of Magnetar’s prover remain to be done. Even so, it uses less memory than SP1 Hypercube, ZisK, and Lineth: about a quarter of SP1 Hypercube’s peak RSS at 2^{22} cycles, and significantly less than ZisK’s and Lineth’s (Section 6.3). The prover of Magnetar (opt) uses 36–40% less memory than that of Magnetar at 2^{22} cycles (Section 6.6).

Proof size and verifier time. At 2^{22} cycles, Magnetar’s proofs are about 600 KiB and verify in 11–13 ms. For comparison, Jolt-Dory’s proofs are 89 KiB and verify in about 60 ms, Jolt-Akita’s are 91 KiB and verify in under 9 ms, and SP1 Hypercube’s core proofs, before recursive compression, are about 2.7 MiB

System	Transfers	Proving (s)	Rate (tx/s)	Verify (ms)	Proof size	Peak RSS
ZisK	71,078	212	336	30	405 KiB	35 GiB
Magnetar	58,240	0.52	111,236	11	599 KiB	6 GiB

Table 4. ZisK and Magnetar on `erc20-transfer` at 2^{22} cycles. Magnetar’s speedup is about $330\times$ in proving and about $300\times$ end to end, where its end-to-end rate is 101,016 tx/s.

and verify in about 63 ms. The proofs of Magnetar (opt) verify in 8–11 ms (Section 6.6).

Jolt’s proofs are smaller because curve- and lattice-based commitments prove evaluations with a logarithmic number of group or ring elements, whereas a hash-based commitment proves them with Merkle authentication paths for many queries. We expect the gap to narrow under recursive composition. The final proof in a recursive chain covers only the verifier of the preceding proof, a small computation, so we can produce it at a much lower code rate; a lower rate means fewer queries and hence a smaller proof, and the extra prover work is modest because the computation is small. Magnetar’s recursion would thus prove the application at a high rate for speed and the final proof at a low rate for size, without affecting end-to-end proving speed.

Over the 64-fold increase in trace length from 2^{16} to 2^{22} cycles, Magnetar’s proofs grow by 218 KiB and verifier time by about 3.7 ms (Section 6.4).

Trace length. We choose 2^{22} cycles because we expect Magnetar to prove larger executions by breaking them into chunks of 2^{22} to 2^{24} cycles and composing the chunk proofs recursively. Jolt instead proves an execution as one monolithic proof, so Section 6.5 also compares the two systems on traces longer than 2^{22} cycles. Magnetar remains faster even when Jolt runs on traces of 2^{28} cycles, the longest that fit in our machine’s memory.

6.3 Comparison with ZisK and Lineth

The ZisK and Lineth provers take minutes to produce a proof on our machine, and neither may be optimized for a laptop, so we run these two systems only on `erc20-transfer`.

ZisK. We calibrate ZisK like the other baselines: 71,078 transfers fill nearly 100% of its 2^{22} -step budget. At 2^{22} cycles, Magnetar’s prover is about $330\times$ faster than ZisK’s local CPU prover (Table 4), although ZisK may not be optimized for a MacBook. ZisK’s main-machine, ROM, memory, and binary-operation tables have a fixed size of 2^{22} rows, so its proving time does not fall at shorter traces: on 17,761 transfers, which fill nearly 100% of its 2^{20} -step budget, it takes 212 s, and Magnetar’s speedup at 2^{20} cycles grows to about $1,100\times$.

Lineth. On 15,360 transfers, Lineth’s prover already peaks at 94 GiB, whereas the other systems prove 3.8–4.6 times as many transfers at 2^{22} cycles. We therefore

System	Transfers	Proving (s)	Rate (tx/s)	Verify (ms)	Proof size	Peak RSS
Lineth	15,360	406	38	292	22.9 MiB	94 GiB
Magnetar	14,528	0.15	95,047	10	530 KiB	2 GiB

Table 5. Lineth and Magnetar on **erc20-transfer**, with Magnetar at 2^{20} cycles. Magnetar’s speedup is about $2,500\times$ in proving and about $2,310\times$ end to end, where its end-to-end rate is 87,457 tx/s.

Trace length	Proving (ms)	Cycles/s	Peak RSS	Proof size	Verify (ms)
2^{16}	35	1.9M	0.2 GiB	380 KiB	8
2^{18}	59	4.5M	0.4 GiB	462 KiB	9
2^{20}	157	6.7M	1.6 GiB	529 KiB	10
2^{22}	547	7.7M	6.4 GiB	598 KiB	12

Table 6. Magnetar’s proving time, throughput, peak RSS, proof size, and verifier time on the **erc20-mixed-block** workload as the trace length grows from 2^{16} to 2^{22} cycles.

run Lineth on 15,360 transfers and compare it with Magnetar at 2^{20} cycles, whose 14,528-transfer input is the closest (Table 5). Since Magnetar’s throughput grows with trace length, this comparison yields a smaller speedup than one at 2^{22} cycles would. Magnetar’s prover is about $2,500\times$ faster than Lineth’s, and Magnetar’s proofs are also smaller and verify faster.

6.4 Scaling and memory

Table 6 reports the **erc20-mixed-block** workload at four trace lengths, with peak resident set size measured over the whole process.

Throughput rises with trace length up to 2^{22} , where it is 14% above 2^{20} . Over the 64-fold increase in trace length, peak RSS grows from 0.2 GiB to 6.4 GiB. Proof size and verifier time grow far more slowly still: the proof grows by 218 KiB and verifier time by about 3.7 ms.

6.5 Scaling to larger traces

Jolt’s throughput grows with trace length: the cost of its PCS evaluation argument grows only with the square root of the trace length, so it becomes a smaller fraction of proving time on longer traces. We therefore also compare the systems on traces longer than 2^{22} cycles. The version of Jolt that we benchmark does not yet implement the streaming prover [39], which would add overhead to its throughput. We keep the setup above, start at 2^{23} cycles, and double the trace length as long as twice the system’s largest peak RSS at the current length stays within 96 GiB. Magnetar runs all 15 workloads. Jolt-Dory and Jolt-Akita run three: **sha256**, which the published Jolt-Akita results use [11]; **erc20-transfer**; and **radix-sort-u64**, where Magnetar’s lead over Jolt-Akita at 2^{22} cycles is smallest. Table 7 reports these three workloads.

	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}	Lead
sha256 (hashes/s)								
Magnetar	1,356	1,381	1,313	—	—	—	—	
Jolt-Dory	51	64	69	71	81	90	97	14.2×
Jolt-Akita	91	94	91	120	130	160	134	8.6×
erc20-transfer (tx/s)								
Magnetar	111,236	111,963	106,093	—	—	—	—	
Jolt-Dory	8,262	9,618	10,155	11,368	13,589	13,997	15,411	7.3×
Jolt-Akita	16,834	18,269	18,079	21,198	21,911	26,758	23,436	4.2×
radix-sort-u64 (keys/s)								
Magnetar	13,172	13,663	13,066	—	—	—	—	
Jolt-Dory	997	1,214	1,280	1,578	1,859	1,955	2,135	6.4×
Jolt-Akita	2,282	2,450	2,398	2,918	3,041	3,785	3,261	3.6×
Peak RSS, maximum over the three workloads (GiB)								
Magnetar	6	13	25	—	—	—	—	
Jolt-Dory	2	4	7	16	25	46	82	
Jolt-Akita	2	3	7	13	21	37	67	

Table 7. Application-level throughput and peak RSS on longer traces. Bold marks each system’s best trace length, and *Lead* is Magnetar’s best throughput divided by the baseline’s best. We run Magnetar only up to 2^{24} cycles, the largest chunk size we expect to use (Section 6.2).

We expect to run Magnetar only on traces of at most 2^{24} cycles and to prove longer executions in chunks (Section 6.2). At these lengths, Magnetar’s prover uses about 1.6 KiB of memory per cycle, and we have not yet pursued further space optimizations.

Jolt needs less memory per cycle, about 0.5 KiB at 2^{22} cycles and 0.3 KiB at 2^{28} . From 2^{22} cycles to its best length, Jolt-Dory’s throughput roughly doubles (best at 2^{28}) and Jolt-Akita’s grows by 1.6–1.8× (best at 2^{27}). Jolt-Akita’s throughput falls at 2^{28} on all three workloads. Jolt-Akita’s throughput on our machine is consistent with its authors’ report [11].

Comparing each system at its best length, Magnetar’s lead on these three workloads is 6.4–14.2× over Jolt-Dory and 3.6–8.6× over Jolt-Akita. As in Section 6.2, the spread reflects cycles per operation rather than proving rate: in cycles per second alone, Magnetar’s prover at its best length is 4.7–6.3× faster than Jolt-Akita’s at its best length on the three workloads.

6.6 Optimized Magnetar

Magnetar (opt) includes optimizations on top of Magnetar. We repeat every Magnetar experiment of this section with Magnetar (opt), on the same machine and with the same setup. Magnetar (opt) proves the same executions as Magnetar,

#	Workload	Rate (M cycles/s)		Gain ($\times$)
		Magnetar	Magnetar (opt)	
1	fibonacci	8.73	12.78	1.46
2	hot-cell-increment	8.32	10.08	1.21
3	stream-copy	7.51	10.15	1.35
4	pointer-chase	8.74	10.01	1.15
5	binary-search	8.06	10.40	1.29
6	radix-sort-u64	7.19	9.98	1.39
7	lz4-decode	7.62	9.89	1.30
8	varint-decode	7.68	10.06	1.31
9	sha256	7.55	11.08	1.47
10	keccak-f1600	7.33	10.94	1.49
11	sha256-merkle-path	7.31	11.00	1.50
12	erc20-transfer	8.01	9.63	1.20
13	erc20-transfer-from	7.70	9.76	1.27
14	erc20-mixed-block	7.66	9.53	1.24
15	evm-erc20-transfer	7.80	9.56	1.23
Geometric mean				1.32
Verifier time (ms)		12.7	10.7	
Peak RSS (GiB)		6.4	4.0	

Table 8. Proving rates of Magnetar and Magnetar (opt) at 2^{22} cycles. *Gain* is the rate of Magnetar (opt) divided by that of Magnetar. The bottom rows give the largest value over the 15 workloads.

with the same numbers of operations and cycles, at 120 bits of provable security, and every proof that we time verifies.

Table 8 compares the two systems at 2^{22} cycles. Magnetar (opt) proves 9.5–12.8 million cycles per second, 1.15–1.50 $\times$ the rate of Magnetar and 1.32 $\times$ in geometric mean. Its geometric-mean proving speedups over Jolt-Dory, Jolt-Akita, and SP1 Hypercube are thus about 24 $\times$, 12.7 $\times$, and 52 $\times$, against about 18 $\times$, 9.6 $\times$, and 39 $\times$ for Magnetar. From 2^{20} to 2^{24} cycles, the gain grows with trace length: in geometric mean over the 15 workloads, Magnetar (opt) proves 1.30 $\times$, 1.32 $\times$, 1.36 $\times$, and 1.44 $\times$ as fast as Magnetar at 2^{20} , 2^{22} , 2^{23} , and 2^{24} cycles, respectively. Comparing each system at its best length as in Section 6.5, the lead of Magnetar (opt) on the three workloads of Table 7 is 9.2–21.6 $\times$ over Jolt-Dory and 5.3–13.1 $\times$ over Jolt-Akita, against 6.4–14.2 $\times$ and 3.6–8.6 $\times$ for Magnetar.

At 2^{22} cycles, the prover of Magnetar (opt) uses 36–40% less memory than that of Magnetar, and its largest peak RSS is 4.0 GiB, less than a sixth of SP1 Hypercube’s. From 2^{22} to 2^{24} cycles, it uses about 1.0 KiB of memory per cycle, against 1.6 KiB for Magnetar.

References

1. Arun, A., Setty, S., Thaler, J.: Jolt: SNARKs for VMs via lookups. In: EUROCRYPT (2024)
2. Ashur, T., Buschman, T., Mahzoun, M.: Algebraic cryptanalysis of the HADES design strategy: Application to Poseidon and Poseidon2. Cryptology ePrint Archive, Paper 2023/537 (2023)
3. Bagad, S., Dao, Q., Domb, Y., Thaler, J.: Speeding up sum-check proving. Cryptology ePrint Archive, Paper 2025/1117 (2025)
4. Barbulescu, R., Duquesne, S.: Updating key size estimations for pairings. Journal of Cryptology **32**(4), 1298–1336 (2019)
5. Ben-Sasson, E., Carmon, D., Haböck, U., Kopparty, S., Saraf, S.: On proximity gaps for Reed–Solomon codes. Electronic Colloquium on Computational Complexity, Report TR25-169 (2025)
6. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for SNARKs and proof-carrying data. In: STOC (2013)
7. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: FOCS (1991)
8. Boneh, D., Chen, B.: LatticeFold: A lattice-based folding scheme and its applications to succinct proof systems. Cryptology ePrint Archive, Paper 2024/257 (2024)
9. Bonneau, J., Meckler, I., Rao, V., Shapiro, E.: Coda: Decentralized cryptocurrency at scale. Cryptology ePrint Archive, Paper 2020/352 (2020)
10. Büinz, B., Chen, B.: Protostar: Generic efficient accumulation/folding for special sound protocols. In: ASIACRYPT (2023)
11. Dao, Q., Bodaghi, O., Khajepour, A., Vitto, G., Badakhshan, M., Georgiades, M., Liu, F., Zhang, J., Thaler, J.: Akita: A high-performance lattice-based polynomial commitment scheme. <https://assets.layerzero.network/pdf/akita.pdf> (2026)
12. Dao, Q., Thaler, J.: More optimizations to sum-check proving. Cryptology ePrint Archive, Paper 2024/1210 (2024)
13. Diamond, B.E., Posen, J.: Polylogarithmic proofs for multilinear over binary towers. Cryptology ePrint Archive, Paper 2024/504 (2024)
14. Eagen, L., Gabizon, A.: Protogalaxy: Efficient ProtoStar-style folding of multiple instances. Cryptology ePrint Archive, Paper 2023/1106 (2023)
15. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: CRYPTO. pp. 186–194 (1986)
16. Goldberg, L., Papini, S., Riabzev, M.: Cairo – a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063 (2021), <https://eprint.iacr.org/2021/1063>
17. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: Interactive proofs for muggles. In: STOC (2008)
18. Golovnev, A., Lee, J., Setty, S., Thaler, J., Wahby, R.: Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In: CRYPTO (2023)
19. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schafneggger, M.: Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Paper 2019/458 (2019)
20. Grassi, L., Khovratovich, D., Schafneggger, M.: Poseidon2: A faster version of the Poseidon hash function. Cryptology ePrint Archive, Paper 2023/323 (2023)
21. Grassi, L., Koschatko, K., Rechberger, C.: Poseidon and Neptune: Gröbner basis cryptanalysis exploiting subspace trails. Cryptology ePrint Archive, Paper 2025/954 (2025)

22. Gruen, A.: Some improvements for the PIOP for ZeroCheck. Cryptology ePrint Archive, Paper 2024/108 (2024)
23. Hemo, T., Jue, K., Rabinovich, E., Roh, G., Rothblum, R.D.: Jagged polynomial commitments (or: How to stack multilinear). Cryptology ePrint Archive, Paper 2025/917 (2025)
24. Jolt Team: feat(recursion): Akita verifier guest under 2^{26} rows on main. Pull request 1911, <https://github.com/a16z/jolt/pull/1911> (2026), accessed October 2026
25. Jolt Team: Jolt codebase. <https://github.com/a16z/jolt> (2026)
26. Kothapalli, A., Parno, B.: Algebraic reductions of knowledge. In: CRYPTO (2023)
27. Kothapalli, A., Setty, S.: SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive (2022)
28. Kothapalli, A., Setty, S.: Cyclefold: Folding-scheme-based recursive arguments over a cycle of elliptic curves. Cryptology ePrint Archive, Paper 2023/1192 (2023), <https://eprint.iacr.org/2023/1192>, <https://eprint.iacr.org/2023/1192>
29. Kothapalli, A., Setty, S.: HyperNova: recursive arguments for customizable constraint systems. In: CRYPTO. pp. 345–379 (2024)
30. Kothapalli, A., Setty, S.: NeutronNova: Group-based folding done right. In: CRYPTO (2026)
31. Kothapalli, A., Setty, S., Tzialla, I.: Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In: CRYPTO (2022)
32. Lean Ethereum: leanVM. <https://github.com/leanEthereum/leanVM> (2026)
33. Lee, J.: Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. Cryptology ePrint Archive, Report 2020/1274 (2020)
34. Lee, J., Nikitin, K., Setty, S.: Replicated state machines without replicated execution. In: S&P (2020)
35. LF Decentralized Trust: Lineth codebase. <https://github.com/LFDT-Lineth/lineth-monorepo> (2026)
36. Lund, C., Fortnow, L., Karloff, H., Nisan, N.: Algebraic methods for interactive proof systems. In: FOCS (Oct 1990)
37. Massip, H.: Secure challenge derivation in ZisK. <https://zisk.technology/secure-challenge-derivation-in-zisk/> (2025)
38. Merz, S.P., García, À.R.: Skipping class: Algebraic attacks exploiting weak matrices and operation modes of Poseidon2. Cryptology ePrint Archive, Paper 2026/306 (2026)
39. Nair, V., Thaler, J., Zhu, M.: Proving CPU executions in small space. Cryptology ePrint Archive, Paper 2025/611 (2025)
40. National Institute of Standards and Technology: Stateless hash-based digital signature standard. FIPS 205 (2024), <https://doi.org/10.6028/NIST.FIPS.205>
41. Nguyen, W., Boneh, D., Setty, S.: Revisiting the Nova proof system on a cycle of curves. Cryptology ePrint Archive, Paper 2023/969 (2023)
42. Nguyen, W., Setty, S.: Neo: Lattice-based folding scheme for CCS over small fields and pay-per-bit commitments. Cryptology ePrint Archive, Paper 2025/294 (2025)
43. Nguyen, W., Setty, S.: Neo and SuperNeo: Post-quantum folding with pay-per-bit costs over small fields. Cryptology ePrint Archive, Paper 2026/242 (2026)
44. Novakovic, A., Angeris, G.: Ligerito: A small and concretely fast polynomial commitment scheme. Cryptology ePrint Archive, Paper 2025/1187 (2025)
45. Polygon Hermez: ZisK codebase. <https://github.com/0xPolygonHermez/zisk> (2026)
46. Polygon Hermez: ZisK v1.3.0-alpha release notes. <https://github.com/0xPolygonHermez/zisk/releases/tag/v1.3.0-alpha> (2026), september 21, 2026

47. RISC ZERO: <https://www.risczero.com/>
48. RISC Zero: Cryptographic security model. <https://dev.risczero.com/api/security-model> (2026), accessed October 2026
49. Roh, G., Rothblum, R.: SP1 V4 Turbo: Memory argument via elliptic curve based multiset hashing. Succinct, https://docs.succinct.xyz/assets/files/SP1_Turbo_Memory_Argument-b042ba18b58c4add20a8370f4802f077.pdf (2024)
50. Setty, S.: Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In: CRYPTO (2020)
51. Setty, S.: BinarySpartan: Spartan over binary fields. Cryptology ePrint Archive, Paper 2026/1656 (2026)
52. Setty, S., Angel, S., Gupta, T., Lee, J.: Proving the correct execution of concurrent services in zero-knowledge. In: OSDI (Oct 2018)
53. Setty, S., Thaler, J., Wahby, R.: Customizable constraint systems for succinct arguments. Cryptology ePrint Archive (2023)
54. Setty, S., Thaler, J., Wahby, R.: Unlocking the lookup singularity with Lasso. In: EUROCRYPT (2024)
55. Setty, S., Thaler, J., Zhu, M.: Twist and shout: Faster memory checking arguments via one-hot addressing and increments. In: CRYPTO (2026)
56. Succinct: SP1. <https://github.com/succinctlabs/sp1> (2026)
57. Succinct: SP1 Hypercube: Global interactions. <https://docs.succinct.xyz/docs/sp1/hypercube/global> (2026), accessed October 2026
58. Succinct: SP1 Hypercube security model. <https://docs.succinct.xyz/docs/sp1/security/security-model> (2026), accessed October 2026
59. Succinct: sp1-sdk crate, version 6.8.0. <https://crates.io/crates/sp1-sdk> (2026)
60. Tzialla, I., Kothapalli, A., Parno, B., Setty, S.: Transparency dictionaries with succinct proofs of correct operation. In: NDSS (2022)
61. Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: TCC. pp. 552–576 (2008)
62. ZisK: ZisK. <https://zisk.technology/> (2026)