Jolt: SNARKs for Virtual Machines via Lookups

Arasu Arun[1], Srinath Setty[2(✉)], and Justin Thaler[3,4]

[1] New York University, New York, USA
[2] Microsoft Research, New York, USA
srinath@microsoft.com
[3] a16z crypto research, Washington, USA
[4] Georgetown University, Washington, USA

Abstract. Succinct Non-interactive Arguments of Knowledge (SNARKs) allow an untrusted prover to establish that it correctly ran some "witness-checking procedure" on a witness. A zkVM (short for zero-knowledge virtual machine) is a SNARK that allows the witness-checking procedure to be specified as a computer program written in the assembly language of a specific instruction set architecture (ISA).

A *front-end* converts computer programs into a lower-level representation such as an arithmetic circuit or generalization thereof. A SNARK for circuit-satisfiability can then be applied to the resulting circuit.

We describe a new front-end technique called Jolt that applies to a variety of ISAs. Jolt arguably realizes a vision called the *lookup singularity*, which seeks to produce circuits that only perform lookups into pre-determined lookup tables. The circuits output by Jolt primarily perform lookups into a gigantic lookup table, of size more than 2^{128}, that depends only on the ISA. The validity of the lookups are proved via a new *lookup argument* described in a companion work called Lasso [STW23]. Although size-2^{128} tables are vastly too large to materialize in full, the tables arising in Jolt are structured, avoiding costs that grow linearly with the table size.

We describe performance and auditability benefits of Jolt compared to prior zkVMs, focusing on the popular RISC-V ISA as a concrete example. The dominant cost for the Jolt prover applied to this ISA (on 64-bit data types) is equivalent to cryptographically committing to under eleven 256-bit field elements per step of the RISC-V CPU. This compares favorably to prior zkVM provers, even those focused on far simpler VMs.

1 Introduction

A SNARK (succinct non-interactive argument of knowledge) is a cryptographic protocol that lets an untrusted prover $\mathcal{P}$ convince a verifier $\mathcal{V}$ that they know

The full version of this work is presented in [AST23], and is accompanied by its companion work, Lasso [STW23].

M. Joye and G. Leander (Eds.): EUROCRYPT 2024, LNCS 14656, pp. 3–33, 2024.
https://doi.org/10.1007/978-3-031-58751-1_1

a witness w satisfying some property. A trivial proof is for $\mathcal{P}$ to send w to $\mathcal{V}$, who can then directly check that w satisfies the claimed property. A SNARK achieves the same effect, but with better costs to the verifier. Specifically, the term *succinct* roughly means that the proof should be shorter than this trivial proof (i.e., the witness w itself), and verification should be much faster than direct witness checking.

As an example, the prover could be a cloud service provider running an expensive computation on behalf of its client (the verifier). A SNARK gives the client confidence that the prover ran the computation honestly. Alternatively, in a blockchain setting, the witness could be a list of valid digital signatures authorizing several transactions. A SNARK can be used to prove that one *knows* the (valid) signatures, so that the signatures themselves do not have to be stored and verified by all blockchain nodes. Instead, only the SNARK needs to be stored and verified on-chain.

1.1 SNARKs for Virtual Machine Abstractions

A popular approach to SNARK design today is to prove the correct execution of *computer programs*. This means that the prover proves that it correctly ran a specified computer program Ψ on a witness. In the example above, Ψ might take as input a list of blockchain transactions and associated digital signatures authorizing each of them, and verify that each of the signatures is valid.

Many projects today accomplish this via a CPU abstraction, also often called a *virtual machine* (VM). Here, a VM abstraction entails fixing a set of *primitive instructions*, known as an instruction set architecture (ISA), analogous to assembly instructions in processor design. A full specification of the VM also includes the number of registers and the type of memory that is supported. The program Ψ to be proved is written in this language.

Systems that generate proofs for these VM abstractions are commonly called "zkVMs". While this is a misnomer as they do not necessarily provide zero-knowledge, we stick with this terminology throughout this work due to its popularity. To list a few examples, several so-called "zkEVM" projects seek to achieve "byte-code level compatibility" with the Ethereum Virtual Machine (EVM). This means that the set of primitive instructions is the 141 opcodes available on the EVM and the types of memory supported are those required in the EVM (such as a stack containing 256-bit elements, a byte-addressable memory, and a key-value store with 256-bit keys and values).

Many other zkVM projects choose (or design) ISAs for their purported "SNARK-friendliness", or for surrounding infrastructure and tooling, or for a combination thereof. For example, Cairo-VM is a very simple virtual machine designed specifically for compatibility with SNARK proving [GPR21, AGL+22]. Another example is the RISC Zero project, which uses the RISC-V instruction set. RISC-V is popular in the computer architecture community, and comes with a rich

ecosystem of compiler tooling. Other zkVM projects include Polygon Miden,[1] Valida,[2] and many others.

Front-End, Back-End Paradigm. SNARKs are built using protocols that perform certain probabilistic checks, so to apply SNARKs to program executions, one must express the execution of a program in a specific form that is amenable to probabilistic checking (e.g., as arithmetic circuits or generalizations thereof). Accordingly, most SNARKs consist of a so-called *front-end* and *back-end*: the front-end transforms a witness-checking computer program Ψ into an equivalent circuit-satisfiability instance, and the back-end allows the prover to establish that it knows a satisfying assignment to the circuit.

Typically, the circuit will "execute" each step of the compute program one at a time (with the help of untrusted "advice inputs"). Executing a step of the CPU conceptually involves two tasks: (1) identify which primitive instruction should be executed at this step, and (2) execute the instruction and update the CPU state appropriately. Existing front-ends implement these tasks by carefully devising gates or so-called constraints that implement each instruction. This is time-intensive and potentially error-prone. As we show in this work, it also leads to circuits that are substantially larger than necessary.

Pros and Cons of the zkVM Paradigm. One major benefit of zkVMs that use pre-existing ISAs is that they can exploit extant compiler infrastructure and tooling. This applies, for example, to the RISC-V and EVM instruction set, and leads to a developer-friendly toolchain without building the infrastructure from scratch. One can directly invoke existing compilers that transform witness-checking programs written in high-level languages down to assembly code for the ISA, and also benefit from prior audits or other verification efforts of these compilers.

Another benefit of zkVMs is that a single circuit can suffice for running all programs up to a certain time bound, whereas alternative approaches may require re-running a front-end for every program. Finally, frontends for VM abstractions output circuits with repeated structure. For a given circuit size, backends targeting circuits with repeated structure [Set20, BSBHR19, WTS+18] can be much faster than backends that do not leverage repeated structure [CHM+20, GWC19, Gro16].

However, zkVMs also have downsides that render them less efficient for some applications. Circuits implementing a VM abstraction must pay for their generality – they must support all possible sequences of CPU instructions as opposed to being tailored for a specific program. This leads to an overhead in circuit size and ultimately, proving costs.

[1] https://polygon.technology/polygon-miden.
[2] https://github.com/valida-xyz/valida-compiler/issues/2.

Another issue is that implementing certain important operations in a zkVM (e.g., cryptographic operations such as Keccak hashing or ECDSA signature verification) is extremely expensive-e.g., ECDSA signature verification takes up to 100 microseconds to verify on real CPUs, which translates to millions of RISC-V instructions.[3] This is why zkVM projects contain so-called gadgets or built-ins, which are hand-optimized circuits and lookup tables computing specific functionalities.

The Conventional Wisdom on zkVMs. The prevailing viewpoint today is that simpler VMs can be turned into circuits with fewer gates per step of the VM. This is most apparent in the design of particularly simple and ostensibly SNARK-friendly VMs such as TinyRAM [BSCG+13a] and the Cairo-VM [4]. However, this comes at a cost, because primitive operations that are standard in real-world CPUs require many primitive instructions to implement on the simple VM. In part to minimize the overheads in implementing standard operations on such limited VMs, many projects have designed domain specific languages (DSLs) that are exposed to the programmer who writes the witness-checking program.

Moreover, existing zkVMs remain expensive for the prover, even for very simple ISAs. For example, the prover for Cairo-VM programs described in [GPR21, AGL+22] cryptographically commits to 51 field elements per step of the Cairo-VM. This means that a single primitive instruction for the Cairo-VM may cause the prover to execute millions of instructions on real CPUs. This severely limits the applicability of SNARKs for VM abstractions, to applications involving only very simple witness-checking procedures.

1.2 Jolt: a0- New Paradigm for zkVM Design

In this work, we introduce a new paradigm in zkVM design. The result is zkVMs with much faster provers, as well as substantially improved auditability and extensibility (i.e., a simple workflow for adding additional primitive instructions to the VM). Our techniques are general. As a concrete example, we instantiate them for the RISC-V instruction set (with multiplication extension [WA17]), a popular open-source ISA developed by the computer architecture community without SNARKs in mind.

Our results upend the conventional wisdom that simpler instruction sets necessarily lead to smaller circuits and associated faster provers. First, our prover is

[3] See https://github.com/risc0/risc0/tree/v0.16.0/examples/ecdsa.

[4] The Cairo-VM has 3 registers, memory that is read-only (each cell can only be written to once) and must be "continuous", and the primitive instructions are roughly addition and multiplication over a finite field, jumps, and function calls. Even the high-level language only exposes write-once (also known as immutable) memory to the programmer and does not offer signed integer data types. See https://www.cairo-lang.org/ for information on the high-level language and [GPR21, AGL+22] and https://github.com/lambdaclass/cairo-vm for information on the virtual machine.

faster per step of the VM than existing SNARK provers for much simpler VMs. Second, the complexity of our prover primarily depends on the size (i.e., number of bits) of the inputs to each instruction. This holds so long as all of the primitive instructions satisfy a natural notion of structure, called *decomposability*. Roughly speaking, decomposability means that one can evaluate the instruction on a given pair of inputs (x, y) by breaking x and y up into smaller chunks, evaluating a small number of functions of each chunk, and combining the results. A primary contribution of our work is to show that decomposability is satisfied by all instructions in the RISC-V instruction set.

Lookup Arguments and Lasso. In a lookup argument, there is a predetermined "table" T of size N, meaning that $T \in \mathbb{F}^N$. An (*unindexed*) lookup argument allows the prover to commit to any vector $a \in \mathbb{F}^m$ and prove that every entry of a resides somewhere in the table. That is, for every $i \in \{1, \ldots, m\}$, there exists some k such that $a_i = T[k]$. In an *indexed* lookup argument, the prover commits not only to $a \in \mathbb{F}^m$, but also a vector $b \in \mathbb{F}^m$, and the prover proves that for every i, $a_i = T[b_i]$. In this setting, we call a the vector of *lookups* and b the vector of associated *indices*.

In a companion paper [STW23], we describe a new lookup argument called Lasso (which applies to both indexed and unindexed lookups). One distinguishing feature of Lasso is that it applies even to tables that are far too large for anyone to materialize in full, so long as the table satisfies the *decomposability* condition mentioned earlier.

Lookup Every Instruction! Say $\mathcal{P}$ claims to have run a certain computer program for m steps, and that the program is written in the assembly language for a VM. Today, front-ends produce a circuit that, for each step of the computation: (1) identifies what instruction to execute at that step, and then (2) executes that instruction. The second step is essentially a switch statement with a case for each instruction, as the circuit should handle any possible instruction in the ISA. This leads to a wasteful blowup in circuit size. Jolt's core idea is to replace step 2 with a single lookup. For each instruction f, the table stores the entire evaluation table of f: that is, if f operates on two 64-bit inputs, this table stores $f(x, y)$ for every pair of inputs $(x, y) \in \{0, 1\}^{64} \times \{0, 1\}^{64}$. This table has size 2^{128}. In this work, we show that all RISC-V instructions are *decomposable*.

In a research forum post in 2022, Barry Whitehat articulated a goal of designing front-ends that produce circuits that *only* perform lookups [Whi], terming it the *lookup singularity*. Circuits that only perform lookups are much simpler to understand and formally verify than circuits consisting of many gates that are often hand-optimized. Arguably, Jolt realizes the vision of the lookup singularity. The bulk of the prover work in Jolt lies in the lookup argument, Lasso. On top of this, the Jolt front-end only performs simple logic to handle memory reads and writes. These are very basic and overall captured in fewer that 50 R1CS constraints!

1.3 Costs of Jolt

Polynomial Commitments and MSMs. A central component of most SNARKs is a cryptographic protocol called a *polynomial commitment scheme* (see Definition 8). Such a scheme allows an untrusted prover to succinctly commit to a polynomial p and later reveal an evaluation $p(r)$ for a point r chosen by the verifier along with a *proof* that the claimed evaluation is correct. In Jolt, as with most SNARKs, the bottleneck for the prover is the polynomial commitment scheme.

Many popular polynomial commitments are based on multi-exponentiations (also known as multi-scalar multiplications, or MSMs). This means that the commitment to a polynomial p (with n coefficients $c_0, \ldots, c_{n-1}$ over an appropriate basis) is $\prod_{i=0}^{n-1} g_i^{c_i}$, for some public generators $g_1, \ldots, g_n$ of a multiplicative group $\mathbb{G}$. Examples include KZG [KZG10], Bulletproofs/IPA [BCC+16, BBB+18], Hyrax [WTS+18], and Dory [Lee21].[5]

The naive MSM algorithm performs n group exponentiations and n group multiplications (note that each group exponentiation is about 400× slower than a group multiplication). But Pippenger's MSM algorithm saves a factor of about $\log(n)$ relative to the naive algorithm. This factor can be well over 10× in practice.

Working Over Large Fields, But Committing to Small Elements. If all exponents appearing in the multi-exponentiation are "small", one can save another factor of 10× relative to applying Pippenger's algorithm to an MSM involving random exponents. This is analogous to how computing $g_i^{2^{16}}$ is 10× faster than computing $g_i^{2^{160}}$: the first requires 16 squaring operations, while the second requires 160 such operations. In other words, if one is promised that all field elements (i.e., exponents) to be committed via an MSM are in the set $\{0, 1, \ldots, K\} \subset \mathbb{F}$, the number of group operations required to compute the MSM depend only on K and not on the size of $\mathbb{F}$.[6]

Quantitatively, if all exponents are upper bounded by some value K, with $K \ll n$, then Pippenger's algorithm only needs (about) one group *operation* per term in the multi-exponentiation. More generally, with any MSM-based commitment scheme, Pippenger's algorithm allows the prover to commit to roughly $k \cdot \log(n)$-bit field elements (meaning field elements in $\{0, 1, \ldots, n\}$) with only k group *operations* per committed field element. So for size-n MSMs, one can commit to $\log(n)$ bits with a *single* group operation.

Polynomial Evaluation Proofs. For many polynomial commitment schemes, the evaluation proof computation is a low-order cost [WTS+18, BBHR18, Lee21].

[5] In Hyrax and Dory, the prover does $\sqrt{n}$ MSMs each of size $\sqrt{n}$.

[6] Of course, the cost of each group operation depends on the size of the group's base field, which is closely related to that of the scalar field $\mathbb{F}$. However, the *number* of group operations to compute the MSM depends only on K, not on $\mathbb{F}$.

Moreover, evaluation proofs exhibit excellent batching properties, whereby the prover can commit to many polynomials and only produce a single evaluation proof across all of them [BGH19, Lee21, KST22, BDFG20]. So in many contexts, computing opening proofs is not a bottleneck even when a scheme such as Bulletproofs/IPA is employed. For these reasons, our accounting in this work ignores the cost of polynomial evaluation proofs.

The Ultimate Cost of Jolt. For RISC-V instructions on 64-bit data types supporting both the base integer instructions and the multiplication extension, the Jolt prover commits to about 80 field elements per step of the RISC-V CPU, with only a dozen being as large as 2^{64}. Table 2 provides the complete distribution. With an MSM-based polynomial commitment, the Jolt prover costs are roughly that of committing to under eleven arbitrary (256-bit) field elements per CPU step.

1.4 Comparison of Prover Costs to Prior Works

This section compares Jolt's commitment cost with other proof systems and zkVM protocols.

Plonk [GWC19] is a popular backend that can prove statements about certain generalizations of arithmetic circuit satisfiability. When Plonk is applied to an arithmetic circuit (i.e., consisting of addition and multiplication gates of fan-in two), the Plonk prover commits to 11 field elements per gate of the circuit, and 7 of these 11 field elements are random. Thus, the Jolt prover costs are roughly equivalent to applying the Plonk backend to an arithmetic circuit with only about one gate per step of the RISC-V CPU.

A more apt comparison is to the RISC Zero project[7], which currently targets the RISC-V ISA on 32-bit data types. A direct comparison is complicated, in part because RISC Zero uses FRI as its (univariate) polynomial commitment scheme, which is based on FFTs and Merkle-hashing, avoiding the use of elliptic curve groups. Still, a crude comparison can be made by using how many field elements the RISC Zero prover commits to, which is at least 275 31-bit field elements per CPU step [Sol23]. At least on small instances, the prover bottleneck is Merkle-hashing the result of various FFTs [Sol23], and one can hash 8 different 31-bit field elements with the same cost as hashing one 256-bit field element. This is roughly equivalent to committing to about $275 \cdot 1/8 \approx 34$ different 256-bit field elements per CPU step. Thus, Jolt commits to significantly fewer elements per CPU step (11 versus 34 in 256-bit equivalents) while also supporting 64-bit architectures.

A final comparison point is to the SNARK for the Cairo-VM described in the Cairo whitepaper [GPR21]. The prover in that SNARK commits to about 50 field

[7] https://www.risczero.com/.

elements per step of the Cairo Virtual Machine, using FRI as the polynomial commitment scheme. StarkWare currently works over a 251-bit field.[8] This field size may be larger than necessary (it is chosen to match the field used by certain ECDSA signatures), but the provided arithmetization of Cairo-VM *requires* a field of size at least 2^{63}. So the commitment costs for the prover are at least equivalent to committing to $50 \cdot 64/256 \approx 13$ 256-bit field elements.[9] Jolt's prover costs per CPU compare favorably to this, despite the RISC-V instruction set being vastly more complicated than the Cairo-VM (and with the Cairo-VM instruction set specifically designed to be ostensibly "SNARK-friendly").

Verifier Costs of Jolt. For RISC-V programs running for at most T steps, the dominant costs for the Jolt verifier are performing $O(\log(T) \log\log(T))$ hash evaluations and field operations,[10] plus checking one evaluation proof from the chosen polynomial commitment scheme (when applied to a multilinear polynomial over at most $O(\log T)$ variables). Verifier costs can be further reduced, and the SNARK rendered zero-knowledge, via composition with a zero-knowledge SNARK with smaller proof size.

1.5 Technical Details: CPU Instructions as Structured Polynomials

As mentioned, Lasso is most efficient when applied to lookup tables satisfying a property called *decomposability*. Intuitively, this refers to tables t such that one lookup into t of size N can be answered with a small number (say, about c) of lookups into much smaller tables $t_1, \ldots, t_\ell$, each of size $N^{1/c}$. Furthermore, if a certain polynomial $\widetilde{t}_i$ associated with each t_i can be evaluated at any desired point r using, say, $O(\log(N)/c)$ field operations,[11] then no one needs to cryptographically commit to any of the tables (neither to t itself, nor to $t_1, \ldots, t_\ell$). Specifically, $\widetilde{t}_i$ can be any so-called *low-degree extension* polynomial of t_i. In Jolt, we will exclusively work with a specific low-degree extension of t_i, called the *multilinear extension*, and denoted $\widetilde{\mathsf{t}}_i$.

[8] See, for example, https://github.com/starkware-libs/starkex-contracts/blob/master/audit/EVM_STARK_Verifier_v4.0_Audit_Report.pdf.

[9] Furthermore, in order to control proof size, StarkWare currently uses a "FRI blowup factor" of 16, compared to RISC Zero's choice of 4. This adds at least an extra factor of 4 to the prover time per field element committed, relative to RISC Zero's.

[10] As described in Appendix G.3 of the full version [AST23], Lasso can use any so-called *grand product argument*. The $O(\log(T) \log\log(T))$ verifier cost are due to the choice of grand product argument from [SL20, Section 6]. Other choices of lookup argument offer different tradeoffs between commitment costs for the prover, versus proof size and verifier time.

[11] The Lasso verifier has to evaluate $\widetilde{t}_i$ at a random point r on its own, so we need this computation to be fast enough that we are satisfied with the resulting verifier runtime. For all tables arising in Jolt, the verifier can compute all necessary $\widetilde{t}_i$ polynomial evaluations in $O(\log(N))$ total field operations.

Hence, to take full advantage of Lasso, we must show two things:

- The evaluation table t of each RISC-V instruction has is decomposable in the above sense. That is, one lookup into t, which has size N, can be answered with a small number of lookups into much smaller tables $t_1, \ldots, t_\ell$, each of size $N^{1/c}$. For most RISC-V instructions, ℓ equals one or two, and about c lookups are performed into each table.
- For each of the small tables t_i, the multilinear extension $\widetilde{t_i}$ is evaluatable at any point, using just $O(\log(N)/c)$ field operations.

Establishing the above is the main technical contribution of our work. It turns out to be quite straightforward for certain instructions (e.g., bitwise AND), but more complicated for others (e.g., bitwise shifts, comparisons).

1.6 Decomposable Instructions

Suppose that table t contains all evaluations of some primitive instruction $f\colon \{0,1\}^n \to \mathbb{F}$. Decomposability of the table t is equivalent to the following property of f: for any n-bit input x to f, x can be decomposed into c "chunks", $X_0, \ldots, X_{c-1}$, each of size n/c, and such that the following holds. There are ℓ functions $f_0, \ldots, f_{\ell-1}$ such that $f(x)$ can be derived in a relatively simple manner from $f_i(x_j)$ as i ranges over $0, \ldots, \ell-1$ and j ranges over $0, \ldots, c-1$. Then the evaluation table t of f is decomposable: one lookup into t can be answered with c total lookups into $\ell \cdot c$ lookups into the evaluation tables of $f_0, \ldots, f_{\ell-1}$.

Bitwise AND is a clean example by which to convey intuition for why the evaluation tables of RISC-V instructions are decomposable. Suppose we have two field elements x and y in $\mathbb{F}$, both in $\{0, \ldots, 2^{64}-1\}$. We refer to x and y as 64-bit field elements (we clarify here that "64 bits" does *not* refer to the size of the *field* $\mathbb{F}$, which may, for example, be a 256-bit field. Rather to the fact that x and y are both in the much smaller set $\{0, \ldots, 2^{64}-1\} \subset \mathbb{F}$, no matter how large $\mathbb{F}$ may be).

Our goal is to determine the 64-bit field element z whose binary representation is given by the bitwise AND of the binary representations of x and y. That is, if $x = \sum_{i=0}^{63} 2^i \cdot x_i$ and $y = \sum_{i=0}^{63} 2^i \cdot y_i$ for $(x_0, \ldots, x_{63}) \in \{0,1\}^{64}$ and $(y_0, \ldots, y_{63}) \in \{0,1\}^{64}$, then $z = \sum_{i=0}^{63} 2^i \cdot x_i \cdot y_i$.

One way to compute z is as follows. Break x and y into 8 chunks of 8 bits each compute the bitwise AND of each chunk, and concatenate the results to obtain z. Equivalently, we can express

$$z = \sum_{i=0}^{7} 2^{8 \cdot i} \cdot \mathsf{AND}(X_i, Y_i), \tag{1}$$

where each $X_i, Y_i \in \{0, \ldots, 2^8 - 1\}$ is such that $x = \sum_{i=0}^{7} 2^{8 \cdot i} \cdot X_i$ and $y = \sum_{i=0}^{7} 2^{8 \cdot i} \cdot Y_i$. These X_i's and Y_i's represent the decomposition of x and y into 8-bit limbs.[12]

In this way, one lookup into the evaluation table of bitwise-AND, which has size 2^{128}, can be answered by the prover providing $X_1, \ldots, X_8, Y_1, \ldots Y_8 \in \{0, \ldots, 2^8 - 1\}$ as untrusted advice, and performing 8 lookups into the size-2^{16} table t_1 containing all evaluations of bitwise-AND over pairs of 8-bit inputs. The results of these 8 lookups can easily be collated into the result of the original lookup, via Eq. (1). No party has to commit to the size-2^{16} table t_1 because for any input $(r'_0, \ldots, r'_7, r''_0, \ldots, r''_7) \in \mathbb{F}^{16}$, the multilinear extension $\widetilde{t}_1(r'_0, \ldots, r'_7, r''_0, \ldots, r''_7) = \sum_{i=0}^{7} 2^i \cdot r'_i \cdot r''_i$, can be evaluated directly by the verifier with only 14 field multiplications and 6 field additions.

Challenges for Other Instructions. One may initially expect that correct execution of RISC-V operations capturing 64-bit addition and multiplication would be easy prove, because large prime-order fields come with addition and multiplication operations that behave like integer addition and multiplication until the result of the operation overflows the field characteristic. Unfortunately, the RISC-V instructions capturing addition and multiplication have specified behavior upon overflow (beyond 64 bits, not 256 bits!) that differs from that of field addition and multiplication. Resolving this discrepancy is one key challenge that we overcome.

2 Technical Preliminaries

2.1 Multilinear Extensions

An ℓ-variate polynomial $p\colon \mathbb{F}^\ell \to \mathbb{F}$ is said to be *multilinear* if p has degree at most one in each variable. Let $f\colon \{0,1\}^\ell \to \mathbb{F}$ be any function mapping the ℓ-dimensional Boolean hypercube to a field $\mathbb{F}$. A polynomial $g\colon \mathbb{F}^\ell \to \mathbb{F}$ is said to *extend* f if $g(x) = f(x)$ for all $x \in \{0,1\}^\ell$. It is well-known that for any $f\colon \{0,1\}^\ell \to \mathbb{F}$, there is a unique *multilinear* polynomial $\widetilde{f}\colon \mathbb{F} \to \mathbb{F}$ that extends f. The polynomial $\widetilde{f}$ is referred to as the *multilinear extension* (MLE) of f.

Multilinear Extensions of Vectors. Given a vector $u \in \mathbb{F}^m$, we will often refer to the *multilinear extension of* u and denote this multilinear polynomial by $\widetilde{u}$. Assuming for simplicity that m is a power of two, $\widetilde{u}$ is obtained by viewing u as a function mapping $\{0,1\}^{\log m} \to \mathbb{F}$ in the natural way[13]: the function interprets its $(\log m)$-bit input $(i_0, \ldots, i_{\log m - 1})$ as the binary representation of an integer i

[12] Just as "digits" refers to a base-10 decomposition of an integer or field element, "limbs" refer to a decomposition into a different base, in this case base 2^8.

[13] All logarithms in this paper are to base 2.

between 0 and $m-1$, and outputs u_i. $\widetilde{u}$ is defined to be the multilinear extension of this function.

Lagrange Interpolation. An explicit expression for the MLE of any function is given by the following standard lemma (see [Tha22, Lemma 3.6]).

Lemma 1. *Let $f\colon \{0,1\}^\ell \to \mathbb{F}$ be any function. Then the following multilinear polynomial $\widetilde{f}$ extends f:*

$$\widetilde{f}(x_0,\ldots,x_{\ell-1}) = \sum_{w\in\{0,1\}^\ell} f(w)\cdot\chi_w(x_0,\ldots,x_{\ell-1}), \tag{2}$$

where, for any $w=(w_0,\ldots,w_{\ell-1})$, $\chi_w(x_0,\ldots,x_{\ell-1}) := \prod_{i=0}^{\ell}(x_i w_i + (1-x_i)(1-w_i))$. Equivalently,

$$\chi_w(x_0,\ldots,x_{\ell-1}) = \widetilde{\mathsf{EQ}}(x_0,\ldots,x_{\ell-1},w_0,\ldots,w_{\ell-1}).$$

The polynomials $\{\chi_w\colon w\in\{0,1\}^\ell\}$ are called the *Lagrange basis polynomials* for ℓ-variate multilinear polynomials. The evaluations $\{\widetilde{f}(w)\colon w\in\{0,1\}^\ell\}$ are sometimes called the coefficients of $\widetilde{f}$ *in the Lagrange basis*, terminology that is justified by Eq. (2).

Lasso can make use of any commitment schemes for *multilinear* polynomials g.[14] Here an ℓ-variate multilinear polynomial $g\colon \mathbb{F}^\ell \to \mathbb{F}$ is a polynomial of degree at most one in each variable.

We employ standards definitions of SNARKs, Polynomial Commitments, Polynomial IOPs, and R1CS constraints and provide them in Appendix A of the full version [AST23] for completeness.

2.2 Lookup Arguments

Lookup arguments allow a prover to commit to two vectors $a\in\mathbb{F}^m$ and $b\in\mathbb{F}^m$ (with a polynomial commitment scheme) and prove that each entry a_i of vector a resides in index b_i of a pre-determined lookup table $T\in\mathbb{F}^N$. That is, for each $i=1,\ldots,m$, $a_i = T[b_i]$. Here, to emphasize the interpretation of T as a table, we use square brackets $T[i]$ to denote the i'th entry of T. Here, if $b_i\notin\{1,\ldots,N\}$, then $t[b_i]$ is undefined, and hence $a_i \neq T[b_i]$. We refer to a as the vector of *looked-up values* and b as the vector of *indices*.

Definition 1 (Lookup arguments, indexed variant). *Let PC = (Gen, Commit, Open, Eval) be an extractable polynomial commitment scheme for*

[14] Any univariate polynomial commitment scheme can be transformed into a multilinear one, though the transformations introduce some overhead (see, e.g., [CBBZ23, BCHO22, ZXZS20]).

multilinear polynomials over $\mathbb{F}$. *A lookup argument (for* indexed lookups*) for table* $T \in \mathbb{F}^N$ *is a SNARK for the relation*

$$\big\{(\mathsf{pp}, \mathcal{C}_1, \mathcal{C}_2, w = (a,b)) \colon a, b \in \mathbb{F}^m \wedge a_i = T[b_i] \forall i \in \{1, \ldots, n\} \\ \wedge \mathit{Open}(\mathsf{pp}, \mathcal{C}_1; \widetilde{a}) = 1 \wedge \mathit{Open}(\mathsf{pp}, \mathcal{C}_2; \widetilde{b}) = 1\big\}.$$

Here $w = (a, b) \in \mathbb{F}^m \times \mathbb{F}^m$ *is the witness, while* pp, $\mathcal{C}_1$, *and* $\mathcal{C}_2$ *are public inputs.*

Definition 1 captures so-called *indexed* lookup arguments (this terminology was introduced in our companion work, Lasso [STW23]. Other works consider *unindexed* lookup arguments, in which only the vector vector $a \in \mathbb{F}^m$ of looked-up values is committed, and the prover claims that *there exists* a vector b of indices such that $a_i = T[b_i]$ for all $i = 1, \ldots, m$.

Definition 2 (Lookup arguments, unindexed variant). *Let* $PC = (\mathsf{Gen}, \mathit{Commit}, \mathit{Open}, \mathit{Eval})$ *be an extractable polynomial commitment scheme for multilinear polynomials over* $\mathbb{F}$. *A lookup argument (for* indexed lookups*) for table* $T \in \mathbb{F}^N$ *is a SNARK for the relation*

$$\big\{(\mathsf{pp}, \mathcal{C}_1, \mathcal{C}_2, a) \colon a \in \mathbb{F}^m \wedge \forall i \in \{1, \ldots, n\}, \exists b_i \text{ such that } a_i = T[b_i] \wedge \mathit{Open}(\mathsf{pp}, \mathcal{C}_1, \widetilde{a}) = 1\big\}.$$

Here $a \in \mathbb{F}^m \times \mathbb{F}^m$ *is the witness, while* pp *and* $\mathcal{C}_1$ *are public inputs.*

Jolt primarily requires indexed lookups. However, a few instructions require range checks, which are naturally handled by unordered lookups (to prove that a value is in the range $\{0, \ldots 2^L - 1\}$, perform an unordered lookup into the table T with $T[i] = i$ for $i = \{0, \ldots, 2^L - 1\}$).

There are natural reductions in both directions, i.e., unindexed lookup arguments can be transformed into index lookup arguments and vice versa.

A Companion Work: Lasso. Our companion work Lasso introduces a family of lookup arguments called Lasso. The lookup arguments in this family are the first that do not require any party to cryptographically commit to the table vector $T \in \mathbb{F}^N$, so long as T satisfies one of the two structural properties defined below.

Definition 3 (MLE-structured tables). *We say that a vector* $T \in \mathbb{F}^N$ *is* MLE-structured *if for any input* $r \in \mathbb{F}^{\log(N)}$, $\widetilde{\mathsf{T}}(r)$ *can be evaluated with* $O(\log N)$ *field operations.*

Definition 4 (Decomposable tables). *Let* $T \in \mathbb{F}^N$. *For a small constant* c, *we say that* T *is* c*-decomposable if there exist a constant* k *and constant* $\alpha \leq kc$ *tables* $T_1, \ldots, T_\alpha$ *each of size* $N^{1/c}$ *and each MLE-structured, as well as a multilinear* α*-variate polynomial* g *such that the following holds. As in Sect. 2.1, let us view* T *as a function mapping* $\{0,1\}^{\log N}$ *to* $\mathbb{F}$ *in the natural way, and view*

each T_i *as a function mapping* $\{0,1\}^{\log(N)/c} \to \mathbb{F}$. *Then for any* $r \in \{0,1\}^{\log N}$, *writing* $r = (r_1, \ldots, r_c) \in \{0,1\}^{\log(N)/c}$,

$$T[r] = g(T_1[r_1], \ldots, T_k[r_1], T_{k+1}[r_2], \ldots, T_{2k}[r_2], \ldots, T_{\alpha-k+1}[r_c], \ldots, T_\alpha[r_c]).$$

We refer to $T_1, \ldots, T_\alpha$ *as* sub-tables.

For any constant $c > 0$ and any c-decomposable table, our companion paper gives a lookup argument called Lasso, in which the prover commits to roughly $3cm + cN^{1/c}$ field elements. Moreover, all of these field elements are *small*, meaning that they are all in $\{0, \ldots, m\}$ (specifically, they are counts for the number of times each entry of each subtable is read), or are elements of the subtables $T_1, \ldots, T_\alpha$. The verifier performs $O(\log(m) \log\log(m))$ hash evaluations and field operations, processes one evaluation proof from the polynomial commitment scheme applied to a multilinear polynomial in $\log m$ variables, and evaluates $\widetilde{T}_1, \ldots, \widetilde{T}_\alpha$ each at a single randomly chosen point.

The Relationship Between MLE-Structured and Decomposable Tables. For any decomposable table $T \in \mathbb{F}^N$, there is some low-degree extension $\hat{T}$ of T (namely, an extension of degree at most k in each variable) that can be evaluated in $O(\log N)$ time. Specifically, the extension polynomial is

$$\hat{T}(r) = g(\widetilde{T}_1(r_1), \ldots, \widetilde{T}_\alpha(r_c)).$$

In general, $\hat{T}$ is not necessarily multilinear, so a table being decomposable does not necessarily imply that it is MLE-structured. In Jolt, we show *all* lookup tables used are *both* c-decomposable (for any integer $c > 0$) as well as MLE-structured.

Lasso with Small Tables ($c = 1$). A special case of Lasso used throughout this work is with a lookup table of small size (say, under 2^{22}) that does not need to be decomposed. Equivalently, this can be thought of as a decomposition with $c = 1$. Such tables are used to range-check small values like bytes, chunks and timestamps.

Remark 1. To show that a value x is in a table of size under 2^{22}, the lookup proof requires the prover to commit to only *one* additional element, the value of which is bounded by the total number of lookup queries made (specifically the *access count* of the subtable).

2.3 Memory Checking

Any SNARK for VM execution has to perform *memory-checking*. This means that the prover must be able to commit to an execution trace for the VM (that is, a step-by-step record of what the VM did over the course of its execution),

and the verifier has to find a way to confirm that the prover maintained memory correctly throughout the entire execution trace. In other words, the value purportedly returned by any read operation in the execution trace must equal the value most recently written to the appropriate memory cell. We use the term *memory-checking argument* to refer to a SNARK for the above functionality. Note that a lookup table $T \in \mathbb{F}^N$ can be viewed as a read-only memory of size N, with memory cell i initialized to $T[i]$. Hence, a lookup argument for indexed lookups (Definition 1) is equivalent the a memory-checking argument for read-only memories.

A variety of memory-checking arguments have been described in the research literature [ZGK+18, BCG+18, BFR+13, BSCGT13] (with the underlying techniques rediscovered multiple times). The most efficient are based on lightweight fingerprinting techniques for the closely related problem of *offline memory checking* [Lip89, BEG+91]. In this work, we use such an argument due to Spice [SAGL18], but optimize it using Lasso. For completeness, we an provide overview of other memory-checking arguments in Appendix G, and Spice's in particular in Appendix G.3 of the full version [AST23].

3 An Overview of RISC-V and Jolt's Approach

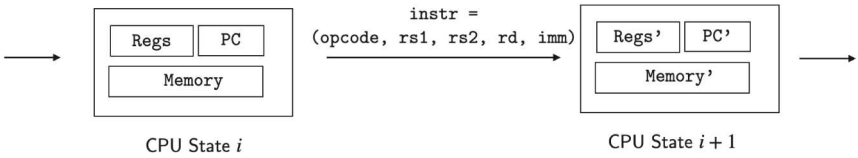
This section first provides a brief overview of the RISC-V instruction set architecture considered in this work. Our goal is to convey enough about the architecture that readers who have not previously encountered it can follow this paper. A complete specification can be found at [WA17].[15] We also stick to regular control flow and do not support external events and other unusual run-time conditions like exceptions, traps, interrupts and CSR registers.

Informally, the RISC-V ISA consists of a CPU and a read-write memory, collectively called the *machine*.

Definition 5 (Machine state). *The machine state consists of* $(\mathsf{PC}, \mathcal{R}, \mathcal{M})$. $\mathcal{R}$ *denotes the 32 integer registers, each of* W *bits, where* W *is 32 or 64.* $\mathcal{M}$ *is a linear read-write byte-addressable array consisting of a fixed number of total locations with each location storing one byte. The* PC*, also of* W *bits, is a separate register that stores the memory location of the instruction to be executed.*

Assembly programs consist of a sequence of instructions, each of which operate on the machine state. The instruction to be executed at a step is the one stored at the address pointed to by the PC . Unless specified by the instruction, the PC is advanced to the next memory location after executing the instruction. The RISC-V ISA specifies that all instructions are 32 bits long (i.e., 4 bytes), so advancing the PC to the next memory location entails incrementing PC by 4.

[15] Another helpful resource for interested readers is Lectures 5–8 at https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/.



(a) The CPU state and instruction formats.

CPU step transition:

1. Read the instruction at location PC in the program code.

   Parse instruction as `[opcode, rs1, rs2, rd, imm]`.
2. Read the W-bit values stored in registers `rs1`, `rs2`.
3. If required, write to or read from memory.

   The value written and memory location accessed are derived from the values stored in `rs1`, `rs2`, `imm`.
4. Perform the instruction's function on the values read from registers and `imm` to get the `output`.

   Examples of functions are arithmetic, logical and comparison operations.
5. Store `output` to register `rd`.

   Only a few instructions, like STOREs, do not involve `rd`.
6. Update PC.

   PC is usually incremented by 4, but instructions like jumps and branches update PC in other ways.

(b) The broad stages of a CPU step transition.

Fig. 1. A model of RISC-V's CPU state and transition function. Note that the transition function is deterministic and all information required, such as the location of memory accessed, is derived from the CPU state and `instr`.

While RISC-V uses multiple formats to store instructions in memory, we can abstract away the details and represent all instructions in the following 5-tuple format.

Definition 6 (5-tuple RISC-V instruction format). *Any RISC-V instruction can be written in the following format: `[opcode, rs1, rs2, rd, imm]`. That is, each instruction specifies an operation code uniquely identifying its function, at most two* source registers *`rs1`, `rs2`, a* destination register *`rd`, and a constant value `imm` (standing for "immediate") provided in the program code itself.*

Figure 1 provides a schematic of the CPU state change and instruction format. Operations read the source registers, perform some computation, and can do any or all of the following: read from memory, write to memory, store a value in `rd`,

or update the PC. For example, the logical left-shift instruction "(SLL, `r5, r8, r2, -`)" reads the value stored in the fifth register, performs a logical left shift on the value by the length stored in the eighth register, and stores the result in second register (and does not involve any immediates). As another example, the branch instruction "(BEQ, `r5, r8, -, imm`)" sets PC to be PC + `imm` if the values stored in the fifth and eighth registers are equal, or routinely increments PC by 4, otherwise (and does not involve the destination register).

Unsigned and Signed Data Types. For the RISC-V ISA, data in registers has no type. A register simply stores W bits. However, different instructions can be conceptualized as interpreting register values in different ways. Specifically, some instructions operate upon unsigned data types, while others operate over signed data types. All RISC-V instructions involving signed data types interpret the bits in a register as an integer via two's complement representation.[16] For many instructions (such as ADD and SUB), the use of two's complement has the consequence that the instruction operates identically regardless of whether or not the inputs are interpreted as signed or unsigned. For some instructions, like multiplication (MUL and MULU) and integer comparison (SLT and SLTU), there will be two different RISC-V instructions, one for signed and one for unsigned. See Appendix D of the full version [AST23] for more information on two's complement notation and arithmetic.

Let z be a W-bit data type with constituent bits $[z_{W-1}, \ldots, z_0]$ such that $z = \sum_{i=0}^{W-1} 2^i \cdot z_i$. When discussing instructions interpreting their W-bit inputs as signed data types represented in twos-complement format, we refer to z_{W-1} as the sign bit of z, and denote this by z_s. (Concretely, the sign bit of a 64-bit register value z will be $z_s = z_{63}$.) We use $z_{<s}$ to refer to $[z_{W-2}, \ldots, z_0] \in \{0,1\}^{W-1}$.

Sign and Zero Extensions. A "sign-extension" of an L-bit value z to W bits (where $L < W$) is the W-bit value $z_{\text{sign-ext}}$ with bits $[z_s, \ldots, z_s, z_{L-1}, \ldots, z_0]$. That is, the sign bit of z is replicated to fill the higher-order bits of z until it reaches length W. A "zero-extension" is when, instead of the sign bit, the 0 bit is used. This results in W-bit $z_{\text{zero-ext}}$ with bits $[0, \ldots, 0, z_{L-1}, \ldots, z_0]$.

3.1 Performing Instruction Logic Using Lookups

As described in Sect. 2.2, the Jolt paradigm avoids the complexity of implementing each instruction's logic as constraints in a circuit by encapsulating instruction execution into a lookup table. Specifically, we identify an "evaluation table" for each operation opcode, $T_{\text{opcode}}[x \parallel y] = r$, that contains the required result for all possible inputs x, y. Jolt combines the tables for all instructions into

[16] See https://en.wikipedia.org/wiki/Two%27s_complement for an overview of how two's complement maps bit vectors in $\{0,1\}^L$ to integers in $\{-2^L, \ldots, 2^L - 1\}$ and vice versa.

one table and thus makes only one lookup query per step to this table as $T_{\text{risc-v}}[\texttt{opcode} \parallel x \parallel y] = r$. Given a processor and instruction set, this table is fixed and independent of the program or inputs. The key contribution of Jolt is to design these enormous tables with a certain *decomposability* structure (see Definition 4) that allows for efficient lookup arguments using Lasso.

Preparing Operands and the Lookup Query. The main responsiblity of the constraint system is to prepare the appropriate operands x, y at each step before the lookup. This is efficient to do as the operands only come from the set {value in `rs1`, value in `rs2`, `imm`, PC}. This means, for example, that the instructions ADD and ADDI are expressed by the same lookup table as they only differ in whether the second operand comes from register `rs2` or is `imm`, respectively. With the operands prepared, the lookup query is then committed to by the prover and fed to the lookup argument for verification. The query is of the form `opcode` $\parallel$ `z` where z is generally $x \parallel y$ or $(x + y)$ or $(x \times y)$, making it either $2 \cdot W$ or $W + 1$ bits in length. The prover provides as advice the claimed entry, `result`, in the lookup table corresponding to the query.

The trace of all lookup queries and entries is sent to Lasso. As described in Definition 4, Lasso requires the query to be split into "chunks" which are fed into different subtables. The prover provides these chunks as advice, which are c in number for some small constant c, and hence approximately W/c or $2W/c$ bits long, depending on the structure of z. The constraint system must verify that the chunks correctly constitute z, but need not perform any range checks as the Lasso algorithm itself later implicitly enforces these on the chunks.

3.2 Using Memory-Checking

The machine state transition involves reading from and writing to three conceptually separate parts of memory: (1) the program code, (2) the registers and (3) the random access memory. As discussed in Sect. 2.3, the most efficient way to enforce correct reads and writes is by using the offline memory checking techniques. These techniques are used for reading from the program code, reading and writing to registrers, and performing load and store operations from the RAM. Unlike other operations, loads and stores do not involve lookups to a large table to perform their core function. As is standard in zkVM design, Jolt conceptualizes the memory-checking procedure as a black box that guarantees correctness of all the memory reads and writes required by the CPU execution, and hence the proof proceeds assuming these operations are correct.

At the start of the proof, the prover commits to the transcript of *all* memory accesses in the form of two sequences: the sequence of reads RS and that of writes WS. Each access is represented as a 3-tuple of field elements (a, v, t) where a is the address read from (or written to), v is the value read (or written) and t is the "timestamp". In writes, the timestamp is the current CPU step counter, and in reads, the timestamp is that of the preceding write to that address.

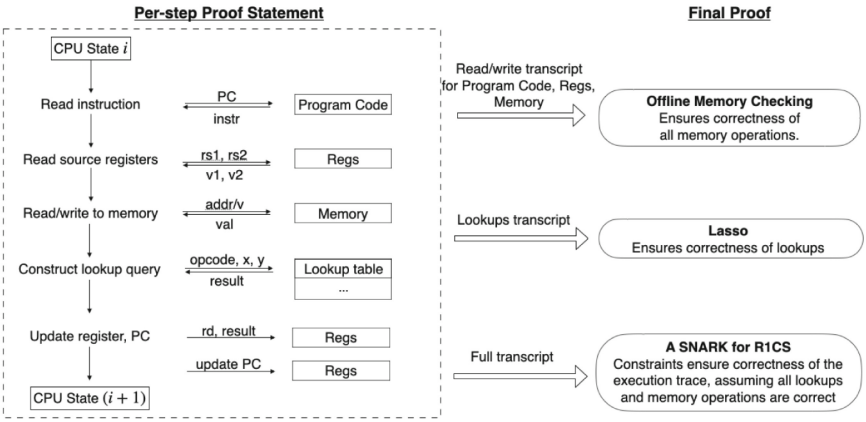


Fig. 2. Proving the correctness of CPU execution using offline memory checking (Sect. 3.2) and lookups (Sect. 3.1).

The offline memory-checking procedure takes these sequences and their commitments as inputs and convinces the verifier that they are consistent: that is, the value read from an address is always the latest value written to that address. The constraint system also takes the transcript of all reads and writes and performs various checks, such as ensuring the address read from or written to is the one deterministically computed by the corresponding step's operation and CPU state. See Appendix C.3 of the full version [AST23] for the list of constraints enforced.

Supporting Byte-Addressable Memory. RISC-V requires that memory be byte-addressable (as opposed to word-addressable). A load or store operation may read up to $W/8$ (which equals four and eight for 32-bit and 64-bit processors, respectively) bytes in a given instruction. Thus, when writing a W-bit value v, the prover must provide its byte-decomposition $[v_1 \dots v_{W/8}]$ as each byte is stored in a separate address in memory. Jolt enforces range-checks on the provided bytes through lookups performed using Lasso. Certain load instructions also require the values read from memory to be sign-extended to W bits. This is enforced using lookups to small tables to obtain the sign bit. See Appendix C.2 of the full version [AST23] for more details.

3.3 Formatting Assembly Code

Before the proof starts, the assembly code is formatted into the 5-tuple form of Definition 6: `(opcode, rs1, rs2, rd, imm)`. Instructions may need to sign-extend or zero-extend `imm` to W bits. This is a deterministic choice that depends only on the instruction (and is independent of the rest of the program or inputs).

Additionally, in our design, each instruction may also comes with a number of one-bit "flags" that guide the constraint system. For example, in our design, `opflag[5]` is 1 to indicate whether the instruction is a jump instruction, and `opflags[7]` is 1 if and only if the lookup's result is to be stored in `rd`. Note that these flags are fixed for any given instruction. See Appendix C.1 of the full version [AST23] for a list of all the fourteen flags used in Jolt.

Putting this together, before the proof starts, the prover and verifier convert the RISC-V assembly code to the 5-tuple format, along with the flags packed into one "packed_flags" value. These six elements are then stored in consecutive locations of a read-only section of memory and accessed using standard offline memory-checking techniques in the constraint system. Jolt thus performs six memory-checking reads per CPU step to read each element of the tuple. As the program code is read-only, the prover's cost involves committing to the elements of the tuple, along with the read timestamp for the address PC .

4 Analyzing MLE-Structure and Decomposability

This section illustrates the process of designing MLE-structured tables and decomposing them as per Definition 4 required by Lasso. We first establish notation and then design the tables for three important functions that are used as building blocks for the tables of many RISC-V instructions: equality, less than, and shifts. The MLE-structured tables and their decomposition for other instructions such as arithmetic ones (like ADD, SUB), logical ones (like AND, OR, XOR), and jumps and branches are described in Appendix B of the full version [AST23]. Load and store instructions do not perform any lookups.

Notation. Let z be a field element in $\{0, 1, \ldots, 2^W - 1\} \subset \mathbb{F}$. We denote the binary representation of z as $\mathsf{bin}(z) = [z_{W-1}, \ldots, z_0] \in \{0,1\}^W$. Here, z_0 is the least significant bit (LSB), while z_{W-1} is the most significant bit (MSB). That is, $z = \sum_{i=0}^{W-1} 2^i z_i$. We refer to the "sign-bit" of z as $z_s = z_{W-1}$. We use $z_{<i}$ and $z_{>i}$ to refer to the subsequences $[z_{i-1}, \ldots, z_0]$ and $[z_{W-1}, \ldots, z_{i+1}]$, respectively.[17]

Concatenation of Bit Vectors. Given two bit vectors $x, y \in \{0,1\}^W$, we use $x \parallel y$ to refer to the number whose binary representation is the concatenation $[x_{W-1}, \ldots, x_0 \parallel y_{W-1}, , \ldots, y_0]$. Under this definition, it holds that $\mathsf{int}(x \parallel y) = \mathsf{int}(x) \cdot 2^W + \mathsf{int}(y)$.

Decomposing Bit Vectors into Chunks. For a constant c, and any $x \in \{0,1\}^L$, we divide the bits of input x naturally into chunks

$$x = [x_{W-1} \ldots x_0] = X_{c-1} \parallel \ldots \parallel X_2 \parallel X_0, \tag{3}$$

[17] In the above paragraphs, we used an italicized z to denote both a field element in $\{0, \ldots, 2^W - 1\}$ and a vector in $\{0,1\}^W$. Throughout the paper, which of the two sets any variable z resides in will be clear from context.

with each $X_i \in \{0,1\}^{W/c}$. In the following discussions, we assume c divides W for simplicity. However, this is not necessary and is in fact more efficient to set $c = 3$ for $W = 32$ and $c = 6$ for $W = 64$, resulting in differing chunk lengths.

Three Instructive Functions and Associated Lookup Tables

Let field $\mathbb{F}$ be a prime order field of size at least 2^W. Let x and y denote field elements that are guaranteed to be in the set $\{0, 1, \ldots, 2^W - 1\}$.

4.1 The Equality Function

MLE-Structured. The equality function EQ takes as inputs two vectors $x, y \in \{0,1\}^W$ of identical length and outputs 1 if they are equal, and 0 otherwise. We will use a subscript to clarify the number of bits in each input to EQ, e.g., EQ_W denotes the equality function defined over domain $\{0,1\}^W \times \{0,1\}^W$. It is easily confirmed that the multilinear extension of EQ_W is as follow:

$$\widetilde{\mathsf{EQ}}_W(x, y) = \prod_{j=0}^{W-1} (x_j y_j + (1 - x_j)(1 - y_j)) \,. \tag{4}$$

Indeed, the right hand side is clearly a multilinear polynomial in x and y, and if $x, y \in \{0,1\}^W$, it equals 1 if and only if $x = y$. Hence, the right hand side must equal the unique multilinear extension of the equality function. Clearly, it can be evaluated at any point $(x, y) \in \mathbb{F}^W \times \mathbb{F}^W$ with $O(W)$ field operations.

Decomposability. To determine whether two W-bit inputs $x, y \in \{0,1\}^W$ are equal, one can decompose x and y into c chunks of length W/c, compute equality of each chunk, and multiply the results together.

Let $x = [X_{c-1}, \ldots, X_0]$ and $y = [Y_{c-1}, \ldots, Y_0]$ denote the decomposition of x and y into c chunks each, as per Eq. (3). Let EQ_W denote the "big" table of size $N = 2^{2W}$ indexed by pairs (x, y) with $x, y \in \{0,1\}^W$, such that $\mathsf{EQ}_W[x \parallel y] = \widetilde{\mathsf{EQ}}_W(x, y)$. Let $\mathsf{EQ}_{W/c}$ denote the "small" table of size $N^{2W/c}$ indexed by pairs (X, Y) of chunks $X, Y \in \{0,1\}^{W/c}$, such that $\mathsf{EQ}_{W/c}[X \parallel Y] = 1$ if $X = Y$ and $\mathsf{EQ}_{W/c}[X \parallel Y] = 0$ otherwise. The table below asserts that evaluating the equality function on x and y is equivalent to evaluating the equality function on each chunk $X_i \parallel Y_i$ and multiplying the results.

CHUNKS	SUBTABLES	FULL TABLE
$\mathsf{c}_i = X_i \parallel Y_i$	$\mathsf{EQ}_{W/c}[X_i \parallel Y_i] = \widetilde{\mathsf{EQ}}_{W/c}(X_i, Y_i)$	$\mathsf{EQ}_W[x, y] = \prod_{i=0}^{c-1} \mathsf{EQ}_{W/c}[X_i \parallel Y_i]$

The (lone) subtable $\mathsf{EQ}_{W/c}$ is MLE-structured by Eq. (4).

4.2 Less Than Comparision

To show that an L-bit value x is less than another L-bit value y, it suffices to enforce an L-bit range check on $y - x$. However, this doesn't work when x and y are treated as two-complement signed numbers and we thus use a lookup table. We explain below how this table is designed through unsigned less-than comparisons (LTU) and show how to adapt it to perform signed comparisons (LTS) in Appendix B.5 of the full version [AST23].

MLE-Structured. The comparison of two unsigned data types $x, y \in \{0, 1, \ldots, 2^{W-1}\}$ is involved in many instructions. For example, SLTU outputs 1 if $x < y$ and 0 otherwise, where the inequality interprets x and y as integers in the natural way. Note that the inequality computed here is strict. Consider the following $2W$-variate multilinear polynomial (LTU below stands for "less than unsigned"):

$$\widetilde{\mathsf{LTU}}_i(x, y) = (1 - x_i) \cdot y_i \cdot \widetilde{\mathsf{EQ}}_{W-i-1}(x_{>i}, y_{>i}). \tag{5}$$

Clearly, this polynomial satisfies the following two properties:

(1) Suppose $x \geq y$. Then $\widetilde{\mathsf{LTU}}_i(x, y) = 0$ for all i.

(2) Suppose $x < y$. Let k be the first index (starting from the MSB of x and y) such that $x_k = 0$ and $y_k = 1$. Then $\widetilde{\mathsf{LTU}}_k(x, y) = 1$ and $\widetilde{\mathsf{LTU}}_i(x, y) = 0$ for all $i \neq k$.

Based on the above properties, it is easy to check that

$$\widetilde{\mathsf{LTU}}(x, y) = \sum_{i=0}^{W-1} \widetilde{\mathsf{LTU}}_i(x, y). \tag{6}$$

Indeed, the right hand side is clearly multilinear, and by the two properties above, it equals $\widetilde{\mathsf{LTU}}(x, y)$ whenever $x, y \in \{0, 1\}^W$. It is not difficult to see that the right hand side of Eq. (6) can be evaluated at any point $(x, y) \in \mathbb{F}^W \times \mathbb{F}^W$ with $O(W)$ field operations as the set $\{\widetilde{\mathsf{EQ}}_{W-i}(x_{>i}, y_{>i})\}_{i=0}^{W-1}$ can be computed in $O(W)$ total steps using the recurrence relation

$$\widetilde{\mathsf{EQ}}_{W-i-1}(x_{>i}, y_{>i}) = \widetilde{\mathsf{EQ}}_{W-i-2}(x_{>(i+1)}, y_{>(i+1)}) \cdot \widetilde{\mathsf{EQ}}(x_i, y_i). \tag{7}$$

See [Tha22, Figure 3.3] for a depiction of this procedure.

Decomposing $\widetilde{\mathsf{LTU}}$. A similar reasoning to the derivation of Eq. (6) reveals the following. As usual, break x and y into c chunks, $X_{c-1} \parallel \cdots \parallel X_0$ and $Y_{c-1} \parallel \cdots \parallel Y_0$. Let $\mathsf{LTU}_{W/c}[X_i \parallel Y_i] = \widetilde{\mathsf{LTU}}_{W/c}(X_i, Y_i)$ denote the subtable with entry 1 if $X_i < Y_i$ when interpreted as unsigned (W/c)-bit data types, and

0 otherwise. Then

$$\begin{aligned}\mathsf{LTU}_W[x \parallel y] &= \sum_{i=0}^{c-1} \mathsf{LTU}_{W/c}[X_i \parallel Y_i] \cdot \mathsf{EQ}_{W/c}[X_{>i} \parallel Y_{>i}] \\ &= \sum_{i=0}^{c-1} \left(\mathsf{LTU}_{W/c}[X_i \parallel Y_i] \cdot \prod_{j<i} \mathsf{EQ}_{W/c}(X_j \parallel Y_j)\right)\end{aligned}$$

Thus, evaluating $\mathsf{LTU}(x, y)$ can be done by evaluating $\mathsf{LTU}_{W/c}$ and $\mathsf{EQ}_{W/c}$ on each chunk (X_i, Y_i) ($\mathsf{EQ}_{W/c}$ need not be evaluated on the lowest-order chunk (X_c, Y_c)). This is summarized in the table below.

CHUNKS	SUBTABLES	FULL TABLE
$\mathtt{c}_i = X_i \parallel Y_i$	$\mathsf{LTU}_{W/c}[X_i \parallel Y_i]$, $\mathsf{EQ}_{W/c}[X_i \parallel Y_i]$	$\mathsf{LTU}_W[x \parallel y] = \sum_{i=0}^{c-1} \mathsf{LTU}_{W/c}[X_i \parallel Y_i] \cdot \prod_{j<i} \mathsf{EQ}_{W/c}[X_j \parallel Y_j]$

The two subtables LTU and EQ are MLE-structured by Eqs. (4) and (6).

4.3 Shift Left Logical

MLE-Structured. SLL takes a W-bit integer x and a $\log(W)$-bit integer y, and shifts the binary representation of x to the left by length y. Bits shifted beyond the MSB of x are ignored, and the vacated lower bits are filled with zeros.[18] For a constant k, let

$$\widetilde{\mathsf{SLL}}_k(x) = \sum_{j=k}^{W-1} 2^j \cdot x_{j-k}. \tag{8}$$

It is straightforward to check that the right hand side of Eq. (8) is multilinear (in fact, linear) function in x, and that when evaluated at $x \in \{0,1\}^W$, it outputs the unsigned W-bit data type whose binary representation is the same as that of the output of the SLL instruction on inputs x and k, $\mathsf{SLL}(x, k)$.

Now consider

$$\widetilde{\mathsf{SLL}}(x, y) = \sum_{k \in \{0,1\}^{\log W}} \widetilde{eq}(y, k) \cdot \widetilde{\mathsf{SLL}}_k(x). \tag{9}$$

It is straightforward to check that the right hand side of Eq. (9) is multilinear in (x, y), and that, when evaluated at $x \in \{0,1\}^W \times \{0,1\}^{\log W}$, it outputs the unsigned W-bit data type $\mathsf{SLL}(x, y)$.

Decomposability. We split the value to be shifted, x, into c chunks, $X_1, \ldots, X_c$, each consisting of $W' = W/c$ bits. y has only one chunk, Y_0, consisting of the

[18] For $L = 32$-bit data types, the RISC-V manual says that the "shift amount is encoded in the lower $5 = \log(W)$ bits".

lowest order $\log W$ bits. As explained below, we decompose a lookup into the evaluation table of SLL into a lookup into c different subtables, each of size $2^{W'+\log W}$. For $W = 64$, a reasonable setting of c would be 4 (instead of the usual $c = 6$ for most other instructions), ensuring that $2^{W'+\log W} = 2^{20}$.

Conceptually, each chunk X_i of X needs to determine how many of its input bits goes "out of range" after the shift of length y. By out of range, we mean that shifting x left by y bits causes those bits to overflow the MSB of x and hence not contribute to the output of the instruction.

For chunks $i = 0, \ldots, (c-1)$ and shift length $k \in \{0,1\}^{\log W}$, define:

$$m_{i,k} = \min\{W', \max\{0, (\mathsf{int}(k) + W' \cdot (i+1)) - W\}\}$$

Here, $m_{i,k}$ equals the number of bits from the i'th chunk that go out of range. Let $m'_{i,k} = W' - m_{i,k} - 1$ denote the index of the highest-order bit within the i'th chunk that does *not* go out of range. Then the evaluation table of SLL decomposes into c smaller tables $\mathsf{SLL}_0, \ldots, \mathsf{SLL}_{c-1}$ as follows.

CHUNKS	SUBTABLES	FULL TABLE
$\mathsf{C}_i = X_i \parallel Y_0$	$\mathsf{SLL}_i[X_i \parallel Y_0] = \sum_{k \in \{0,1\}^{\log W}} \widetilde{\mathsf{EQ}}(Y_0, k) \cdot \left(\sum_{j=0}^{m'_{i,k}} 2^{j+\mathsf{int}(k)} \cdot X_{i,j} \right)$	$\mathsf{SLL}[x \parallel y] = \sum_{i=0}^{c-1} 2^{i \cdot W'} \cdot \mathsf{SLL}_i[X_i \parallel Y_c]$

Note that each SLL_i can be evaluated at any input $(x, y) \in \mathbb{F}^{W'} \times \mathbb{F}^{\log W}$ in $O(W')$ field operations. Indeed, the set $\{\widetilde{\mathsf{EQ}}(Y_0, k)\}_{k \in \{0,1\}^{\log W}}$ can be computed in $O(W)$ field operations via the recurrence in Eq. (7). Similarly, the set $\{2^{j+\mathsf{int}(k)}\}_{i \in \{0,\ldots,c-1\}, k \in \{0,1\}^{\log W}}$ can be computed with $O(W)$ field operations. It follows that $\mathsf{SLL}_0(x \parallel y), \ldots, \mathsf{SLL}_{c-1}(x \parallel y)$ can be evaluated in $O(W)$ field operations in total.

4.4 The Multiplication Extension

On top of the base integer instruction set, RISC-V supports various instructions to multiply, divide and find the remainder with two operands in an optional "M" extension. Jolt can handle all of these instructions, most with no additional overhead. The only caveat is that six of these are split into via several "pseudoinstructions". For example, division is handled by having P provide the quotient and remainder as untrusted advice in one pseudoinstruction, and they are checked for correctness by performing multiplication and addition with more pseudoinstructions. Appendix E of the full version [AST23] describes these techniques along with the tables for each new instruction.

5 Putting It All Together: A SNARK for RISC-V Emulation

The overall architecture of Jolt is depicted in Fig. 2. The Jolt prover executes the program to obtain the trace, and calls the provers for each module (memory-checking, lookups and constraint satisfaction) to obtain three proofs that together form the Jolt proof. This involves the prover cryptographically committing to the execution trace z of the VM on the appropriate input (or more precisely, its multilinear extension polynomial $\widetilde{z}$, using any multilinear polynomial commitment scheme). We leave a detailed discussion of the R1CS and memory-checking modules of Jolt to Appendix F of the full version [AST23], and focus on a particular aspect of the lookup argument here.

5.1 Combining Instruction Lookup Tables into One

The previous sections so far explained that the evaluation table of each individual RISC-V instruction is both MLE-structured and decomposable. But Lasso is a lookup argument for a single decomposable table. We now explain how to bridge this gap, and thus make Jolt use "just one lookup table".

Closely related issues have been addressed in earlier work on zkVMs. Specifically, the fact that the evaluation tables of different RISC-V instructions have different decompositions into subtables is analogous to the following issue dealt with in earlier approaches to front-end design for zkVMs: different instructions are computed by different circuits, and while only one instruction is executed per step of the VM, in general it is not known until runtime which instruction will be executed at any given step. Appendix F of the full version [AST23] discusses techniques used in existing works, such as the re-ordering approach of vRAM [ZGK+18], to solve this problem.

Conceptually, the approach we use expresses the concatenation of the evaluation tables of each instruction (and of which we have shown to be decomposable) as itself decomposable, analogous to how the concatenation of MLE-structured tables is itself MLE-structured. To this end, it is convenient to treat each instruction as leading to $2c-1$ different lookups into subtables ($2c-1$ here comes from the maximum number of subtable lookups across all instructions, namely due to SLT as described in Sect. 4). For instructions that require fewer than $2c-1$ lookups into subtables, the extraneous lookup results can be set to 0, thereby avoiding any cryptographic work on the part of the prover if using an MSM-based commitment scheme (we will explain below how to ensure that these extraneous subtable lookup results will be ignored by all subtables).

There will be a single collation polynomial g (Definition 4) for all instructions, but g will take as input not only the results of relevant subtable lookups, but also 8 additional variables that, when assigned values in $\{0,1\}$, are interpreted as the

bit-representation of the opcode. Denoting these variables as $w = (w_1, \ldots, w_8)$, and letting $g_i(z)$ denote the collation polynomial for the i'th instruction, and letting x denote a vector of $2c-1$ variables, interpreted as specifying the results of $2c-1$ subtable lookups, we define

$$g(w, x) = \sum_{y \in \{0,1\}^8} \widetilde{\mathsf{EQ}}(w, y) \cdot g_{\mathsf{int}(y)}(x).$$

This definition ensures that for any instruction i, $g(\mathsf{bin}(i), x) = g_i(x)$, i.e., collation for each instruction is performed correctly by g.

The core of Lasso is to invoke a grand product argument for each subtable (as explained in Appendix G.3 of the full version [AST23], this is due to the use of memory-checking techniques to verify that the sequences of reads from the subtable are consistent). We can modify the circuit used to compute these products for each subtable to take as input the bits of the opcode associated with each lookup. This way, the circuit can simply ignore any lookups associated with opcodes that do not access the subtable associated with the circuit.

To minimize the size of this circuit, rather than having the prover commit to the 8 bits of the opcode, it may be preferable to instead have the prover commit to some additional Boolean flags (beyond the Boolean circuit flags already described in Sect. 3.3), so that each subtable's circuit only needs to inspect fewer than 8 Boolean flags to determine whether or not a given lookup operation actually is intended to access the subtable.

6 Qualitative Cost Estimation

The overall architecture of Jolt is depicted in Fig. 2. In this section, we analyze the cost of a lookup and of the prover's per-step work in Jolt.

6.1 Cost of a Lookup

We first briefly state the costs incurred by the prover when making a lookup query in terms of the bit-lengths of the elements to be committed to. A Lasso lookup into a decomposed table involves committing to the following elements for some fixed Lasso parameter c:

1. Chunks of the operands x, y or of $x + y$ or $x \times y$. (Each $< 2W/c$ bits.)
2. The outputs of the subtables involved in the lookup. (Each $< W/c$ bits.)
3. Access counts of each subtable at that step. (Each $< \log T$ bits.)
4. Elements involved in the range checks of the chunks. (Each $< \log T$ bits.)

In the worst case (the LTU table, to be precise), each step involves at most $2c-1$ total elements. We use this scenario when reporting costs in Sect. 6.2. Note that the chunks need to be range-checked as it is possible for the prover to provide invalid chunks x_i, y_i but together form a valid lookup index $x_i \parallel y_i$. These range checks are a special case of Lasso with parameter $c = 1$ and, from Remark 1, each involve committing to a single element bounded by the step counter.

With parameters $(W = 32, c = 3)$, a lookup requires committing to 6 elements of at most 11 bits, 6 elements of at most 22 bits, and 12 elements that are equal to the step counter.

6.2 Overall Prover Costs in Jolt

From Fig. 2, the broad steps involved are committing to the transcripts and then proving satisfaction of the constraint system, lookup arguments and offline memory-checking procedures. The prover's field operation costs and the verifier's costs are presented in Table 1. = We note here that the constraint system is a uniform circuit consisting of under 50 R1CS constraints per CPU step. In Appendix C.3 and F of the full version [AST23], we describe the constraints and how Spartan is used to prove that they are satisfied. As discussed in Sect. 1.3, the dominating cost for the prover is in producing commitments.

Table 1. Field operation costs involved in Jolt for a program that runs in n steps with memory of size M. Cryptographic group operation costs are described in Tables 2 and 3. Lasso is run with parameter c and Jolt performs at most one Lasso lookup per step. We assume that memory-checking and Lasso protocols implement the optimized variant of the GKR protocol due to Thaler [Tha13]. We assume Spartan is instantiated with a polynomial commitment scheme with $O(\log n)$-verification time opening proofs.

Jolt field operation costs			
Module	Dominating Cost	$\mathcal{P}$ Cost	$\mathcal{V}$ Cost
Memory-checking	$13 \cdot n$ memory operations on an $(M + \lvert\mathsf{code}\rvert + 32)$-sized memory	$O(n + M)$	$O(\log^2(n + M))$
Lasso lookups	n lookups on a decomposable table of size $O(2^{128})$	$O(c^2 n)$	$O(\log^2 n)$
Constraint checking	Spartan proof on a uniform R1CS with $\approx 50 \cdot n$ constraints	$O(n)$	$O(\log n)$

The rest of this section analyzes the elements that are committed to in Jolt. Table 2 provides a upper bound on the elements committed per step grouped by their bit-lengths. We measure bit-length as that is the main factor determining the commitment cost when using Pippenger's multi-scalar multiplication algorithm.[19] We provide below a brief overview of the elements involved and leave a more detailed discussion to Appendix C of the full version [AST23].

[19] In Pippenger's multi-scalar multiplication algorithm to commit to elements, committing to an N-bit element costs roughly $\mathsf{ceil}(N/22)$ group operations. This makes committing to a 32-bit element cost two group operations while a 256-bit element costs 12 group operations.

Table 2. An approximate spread of the spread of elements committed to in Jolt in lookup-based operations (i.e., excluding loads and stores which do not involve lookups) by their bit-length. We assume that the program code is under 2^{22} bytes long, and the program finishes in under 2^{22} CPU steps. The Lasso parameter $c = 3$ when $W = 32$ and $c = 6$ when $W = 64$. We approximate the per-step committments costs in terms of the cost of committing to a 256-bit element when using Pippenger's MSM algorithm: an n-bit number involves $\lceil n/22 \rceil$ group operations to commit to.

Per-step Commitment Costs for Non-Memory Operations			
Bit-length	Number of Elements	In RV32 $W = 32, c = 3$	In RV64 $W = 64, c = 6$
1	22	22	22
$[2, 12]$	9	9	9
$(2W/c) \approx 22$	5c	15	30
$\log(T)$	9+4c	21	33
W	12	12	12
Total Elements	$52 + 9c$	79	106
In 256-bit commit equivalents:		< 8 elements	< 11 elements

Elements Involved in CPU Execution. First, let's look at the elements involved in satisfying the CPU step circuit's constraints before looking at the elements needed for the Lasso argument. The smallest of these are the 1-bit circuit flags (Sect. 3.3) and the opcode bits (Sect. 5), and the 5-bit elements indexing the source (`rs1`, `rs2`) and destination (`rd`) registers read from the instruction. Slightly larger elements are the PC (which could be as large as $\log |\texttt{program\_code}|$ bits) and the step counter, both of which we assume to be under 2^{22} to simplify our analysis. Finally, the largest elements involved are the W-bit ones specifying the values stored in the two source registers, the sign-extended `imm` read from the program code, the lookup output, (which is generally stored in the destination register), and the advice element involved (only) in division and reminder operations. However, an instruction uses at most 4 of these elements (specifically, this is because the division/remainder instructions do use advice never use `imm`). Beyond this, there are more (eight to be specific) W-bit values that arise as auxiliary witness values involved in constraint satisfaction. These can be thought of as the internal wires of the circuit representing each step's constraint checks.

6.3 Cost of Memory Operations

Load and store operations do not involve large lookups to perform the core instruction logic. Rather, the main cost here is performing memory-checking operations, one for each byte of memory involved in the load/store. This can be up to four for 32-bit processors and eight for 64-bit processors. The elements involved on top of the non-lookup elements of the non-memory instructions are

Table 3. The spread of elements committed per memory operation with the extra overhead elements per byte of load or store. See Appendix C.2 and Table 5 of the full version [AST23] for more details on the exact procedure and elements involved. Note that the per-step costs are independent of the total size of the memory . We approximate these costs in terms of the cost of committing to a 256-bit element when using Pippenger's MSM algorithm, assuming that the program code is under 2^{22} bytes long and the program finishes executing in under 2^{22} CPU steps.

Base Costs per Memory Instruction		**Overhead per Byte**	
Bit-length	Number of Elements	for Loads	for Stores
1	23	1	1
$[2, 12]$	4	1	1
$(2W/c) \approx 22$	2	1	1
$\log(T)$	8	2	3
W	10	–	–
Total Elements	47	5	6
In 256-bit equivalents (both RV32, RV64)	$\approx 5 - 6$ elements	≈ 0.5 elements	≈ 0.5 elements

the actual bytes read/written, the timestamps involved in memory-checking (one for each byte), and the cost of range checking these bytes and timestamps. Memory operations also commit to fewer W-bit elements as they don't involve computing the lookup query or reading the lookup output. Stores, which are memory "writes", require 8-bit range checks of the bytes written. These range-checks are again very efficient in Lasso (see Remark 1) and only involve committing to a single element of value at most the number of steps up to that point.

Acknowledgements and Disclosures. Justin Thaler was supported in part by NSF CAREER award CCF-1845125 and by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the United States Government or DARPA.

Disclosures. Thaler is a Research Partner at a16z crypto and is an investor in various blockchain-based platforms, as well as in the crypto ecosystem more broadly (for general a16z disclosures, see https://www.a16z.com/disclosures/.)

References

[AGL+22] Avigad, J., Goldberg, L., Levit, D., Seginer, Y., Titelman, A.: A verified algebraic representation of cairo program execution. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 153–165 (2022)

[AST23] Arun, A., Setty, S., Thaler, J.: Jolt: snarks for virtual machines via lookups. Cryptology ePrint Archive, Report 2023/1217 (2023)

[BBB+18] Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2018)

[BBHR18] Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Fast Reed-Solomon interactive oracle proofs of proximity. In: Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP) (2018)

[BCC+16] Bootle, J., Cerulli, A., Chaidos, P., Groth, J., Petit, C.: Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In: Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) (2016)

[BCG+18] Bootle, J., Cerulli, A., Groth, J., Jakobsen, S., Maller, M.: Arya: nearly linear-time zero-knowledge proofs for correct program execution. In: Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT) (2018)

[BCHO22] Bootle, J., Chiesa, A., Hu, Y., Orru, M.: Gemini: elastic snarks for diverse environments. In: Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) (2022)

[BDFG20] Boneh, D., Drake, J., Fisch, B., Gabizon, A.: Halo Infinite: Recursive zk-SNARKs from any Additive Polynomial Commitment Scheme. Cryptology ePrint Archive, Report 2020/1536 (2020)

[BEG+91] Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS) (1991)

[BFR+13] Braun, B., Feldman, A.J., Ren, Z., Setty, S., Blumberg, A.J., Walfish, M.: Verifying computations with state. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (2013)

[BFS20] Bünz, B., Fisch, B., Szepieniec, A.: Transparent SNARKs from DARK compilers. In: Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) (2020)

[BGH19] Bowe, S., Grigg, J., Hopwood, D.: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021 (2019)

[BGtR23] Bruestle, J., Gafni, P., the RISC Zero Team: Scalable, transparent arguments of RISC-V integrity, RISC Zero zkVM (2023)

[BSBHR19] Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable zero knowledge with no trusted setup. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 701–732. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_23

[BSCG+13a] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_6

[BSCG+13b] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: Tinyram architecture specification, vol. 991. en. In:(Aug. 2013), pp. 16 (2013)

[BSCGT13] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: Fast reductions from rams to delegatable succinct constraint satisfaction problems. In: Proceedings of the 4th conference on Innovations in Theoretical Computer Science, pp. 401–414 (2013)

[BSCTV14] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 276–294. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_16

[CBBZ23] Chen, B., Bünz, B., Boneh, D., Zhang, Z.: HyperPlonk: plonk with linear-time prover and high-degree custom gates. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023. LNCS, vol. 14005, pp. 499–530. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30617-4_17

[CHM+20] Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N.: Marlin: preprocessing zkSNARKs with universal and updatable SRS. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 738–768. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_26

[CMT12] Cormode, G., Mitzenmacher, M., Thaler, J.: Practical verified computation with streaming interactive proofs. In: Proceedings of the Innovations in Theoretical Computer Science (ITCS) (2012)

[ET18] Eberhardt, J., Tai, S.: Zokrates - scalable privacy-preserving off-chain computations. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp. 1084–1091 (2018)

[FS86] Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Proceedings of the International Cryptology Conference (CRYPTO), pp. 186–194 (1986)

[GKR08] Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. In: Proceedings of the ACM Symposium on Theory of Computing (STOC) (2008)

[GPR21] Goldberg, L., Papini, S., Riabzev, M.: Cairo–a Turing-complete stark-friendly CPU architecture. Cryptology ePrint Archive (2021)

[Gro16] Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_11

[GWC19] Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. ePrint Report 2019/953 (2019)

[KST22] Kothapalli, A., Setty, S., Tzialla, I.: Nova: recursive zero-knowledge arguments from folding schemes. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022. LNCS, vol. 13510, pp. 359–388. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15985-5_13

[KZG10] Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 177–194. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_11

[Lee21] Lee, J.: Dory: efficient, transparent arguments for generalised inner products and polynomial commitments. In: Nissim, K., Waters, B. (eds.) TCC 2021. LNCS, vol. 13043, pp. 1–34. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90453-1_1

[LFKN90] Lund, C., Fortnow, L., Karloff, H., Nisan, N.: Algebraic methods for interactive proof systems. In: Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS) (1990)

[Lip89] Lipton, R.J.: Fingerprinting sets. Princeton University, Department of Computer Science (1989)

[SAGL18] Setty, S., Angel, S., Gupta, T., Lee, J.: Proving the correct execution of concurrent services in zero-knowledge. In: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2018)

[Set20] Setty, S.: Spartan: efficient and general-purpose zkSNARKs without trusted setup. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12172, pp. 704–737. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56877-1_25

[SL20] Setty, S., Lee, J.: Quarks: quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275 (2020)

[Sol23] Solberg, T..: RISC Zero prover protocol & analysis (2023). https://github.com/ingonyama-zk/papers/blob/main/risc0_protocol_analysis.pdf

[STW23] Setty, S., Thaler, J., Wahby, R.S.: Lasso: Unlocking the lookup singularity. Cryptology ePrint Archive, Report 2023/1216 (2023)

[Tha13] Thaler, J.: Time-optimal interactive proofs for circuit evaluation. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 71–89. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_5

[Tha22] Thaler, J.: Proofs, arguments, and zero-knowledge. Found. Trends Priv. Secur. **4**(2–4), 117–660 (2022)

[WA17] Waterman, A., Asanovic, K.: The RISC-V instruction set manual (2017). https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[Whi] Whitehat, B.: Lookup singularity. https://zkresear.ch/t/lookup-singularity/65/7

[WTS+18] Wahby, R.S., Tzialla, I., Shelat, A., Thaler, J., Walfish, M.: Doubly-efficient zkSNARKs without trusted setup. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2018)

[ZGK+18] Zhang, Y., Genkin, D., Katz, J., Papadopoulos, D., Papamanthou, C.: vRAM: faster verifiable RAM with program-independent preprocessing. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2018)

[ZXZS20] Zhang, J., Xie, T., Zhang, Y., Song, D.: Transparent polynomial delegation and its applications to zero knowledge proof. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2020)