Dora: A Simple Approach to Zero-Knowledge for RAM Programs

Aarushi Goel* Mathias Hall-Andersen† Gabriel Kaptchuk‡

Abstract

Existing protocols for proving the correct execution of a RAM program in zero-knowledge are plagued by a *processor expressiveness tradeoff*: supporting fewer instructions results in smaller processor circuits (which improves performance), but may result in more program execution steps because non-supported instruction must be emulated over multiple processor steps (diminishing performance).

We present Dora, a very simple and concretely efficient zero-knowledge protocol for RAM programs that sidesteps this tension by making it (nearly) free to add additional instructions to the processor. The computational and communication complexity of proving each step of a computation in Dora, is *constant* in the number of supported instructions. Dora's approach is united by intuitive abstraction we call a ZKBag, a cryptographic primitive constructed from linearly homomorphic commitments that captures the properties of a physical bag. We implement Dora and demonstrate that on commodity hardware it can prove the correct execution of a processor with thousands of instruction, each of which has thousands of gates, in just a few milliseconds per step. Our evaluation shows that Dora has notably better end-to-end performance than concurrent work targeting the same problem.

*Purdue University aarushi@purdue.edu
†Galois, zkSecurity and Aarhus University mathias@hall-andersen.dk
‡University of Maryland, College Park kaptchuk@umd.edu.

Contents

1 Introduction

Zero-knowledge proofs and arguments[1] [GMR85, GMW86] empower a prover to demonstrate to a verifier that executing a public program p on some secret inputs x yields a particular output y, i.e., $p(x) = y$. A long line of work has demonstrated feasibility of practically efficient zero-knowledge systems [GS08, JKO13, GGPR13, BCC$^+$16, Gro16, KKW18, BBB$^+$18, BCR$^+$19, GWC19, Set20, HK20c, BMRS21, WYKW21, YSWW21, WYX$^+$21, WYY$^+$22, BBMHS22, YHKD22]. As a result, zero-knowledge proofs are now being integrated as a key component of deployed systems [BCG$^+$14, Zav20, se19].

Most existing zero-knowledge proof systems require that p is represented as an arithmetic or boolean circuit (or an equivalent algebraic constraint system). Most natural programs, however, are *RAM programs*, i.e., programs designed for von Neumann architectures. RAM programs capture the intuitive notion of computation used by most practitioners, in which a central processing unit (CPU) with a fixed set of instructions incrementally operates on a large, *random access memory* using load and store operations.

Closing the gap between the RAM programs about which provers and verifiers are interested and the circuit representations required by zero-knowledge provers is a key hurdle to making zero-knowledge more deployable. This gap is more than semantic, as some algorithms in the RAM model are more efficient than comparable circuit representations (e.g., sorting). In other circumstances, the prover and verifier might be particularly interested in an existing RAM program, e.g., proving knowledge of a software exploit against a deployed RAM program [HK20c, GHAH$^+$23, HYDK21, HK21] or proving that software is legally compliant [BCG$^+$22, BCGW22].

Zero-knowledge for RAM Programs. In order to facilitate proving executions of RAM programs in zero-knowledge, we use techniques that reduce RAM programs to the circuit representations used within zero-knowledge systems. One straightforward approach is to leverage *circuit compilers* [MGC$^+$16, Wan]: transform the source code describing a RAM program into a circuit capturing the same functionality. Correct evaluation of the resulting circuit can then be proven using any existing zero-knowledge system for circuits, as done in [HK20c]. This approach, however, introduces several, noteworthy inefficiencies: the resulting circuit must be input-independent, all loops must be unrolled for a fixed number of iterations, and all input-dependent conditional branches are part of the circuit description.[2] These constraints can significantly increase the size of the resulting circuit, and therefore increase the complexity of proving their evaluation in zero-knowledge.

Instead, the state-of-the-art approach to proving the execution of RAM programs in zero-knowledge [BCG$^+$13, BCTV14b, BCTV14a, HK20a, HYDK21, HK21, FKL$^+$21, DOTV22] emulates the execution of the RAM program on a custom processor. First, the prover and verifier agree upon a circuit representation of the CPU and RAM access module. Then, the prover demonstrates that a step of the computation was executed correctly by showing that the program state (comprising the state of memory and any additional registers in the CPU) is the outcome of a *valid state transition* from the previous program state, where the valid transition functions are determined by the processor's instruction set. This process is repeated until program execution has completed.

The Expressiveness Tradeoff. Designing an optimized processor to use when proving execution of RAM programs in zero-knowledge requires grappling with an *expressiveness tradeoff*. It is natural to want a very *small* processor circuit with very few instructions, as the prover must "pay" for all the instructions in the processor in each step of the proof—even unused instructions. This is because proving each state transition using modern zero-knowledge proof systems—including SNARKs—will have prover complexity proportional to total size of the processor. As such, minimizing processor size has become standard practice; Ben-Sasson et al. [BCG$^+$13, BCTV14b] introduced a processor called TinyRAM with only 29 instructions for this purpose, and recent works have created other processors with even fewer instructions [HK20a, HYDK21, FKL$^+$21]. This approach, however, results in *more* steps of program execution—potentially negating the value of a smaller circuit representation of the processor—because instructions not included in the processor must be *emulated* over multiple processor steps. Finding the right balance between processor expressiveness (i.e., how many instructions it supports) and program length is a highly nuanced engineering problem and will depend on the specific RAM program.

In this work, we propose a simple, new approach to RAM zero-knowledge that avoids the expressiveness tradeoff altogether. Our work leverages the observation that the processor circuit has a very specific structure; namely,

[1] We use the terms proofs and arguments interchangeably in this work, as is common in practically oriented work on zero-knowledge.
[2] As discussed below, some recent work has shown how to avoid the communication costs associated with branching.

that it is a *disjunction* of the supported instructions. A sequence of recent works on disjunctive zero-knowledge [HK20c, BMRS21, GGHAK22b, KST22, KS22, GHAKS23, KS23] have shown that it is possible to design zero-knowledge protocols with prover complexity proportional only to the size of the largest clause in the disjunction. Within the context of RAM zero-knowledge, this would allow adding additional instructions to the processor circuit for free, thereby increasing expressiveness.

1.1 Our Contributions

We present Dora, a conceptually simple and concretely efficient zero-knowledge proof system for RAM programs with a *non-succinct* proof size. We focus on designing a non-succinct proof system because they have been shown by several recent and groundbreaking works [DIO21, YSWW21, YHH$^+$23, BMRS21, WYKW21, DOTV22] to yield significantly better proof generation times as compared to succinct zero-knowledge (i.e., zkSNARKs).[3] Dora provides a new way out of the *expressiveness tradeoff* by supporting increased processor expressiveness for free (both in terms of computation and communication). Dora has the following desirable attributes:

- *Communication and Computation Complexity of $O(t + \ell)$,* where t is the number of steps of the computation and ℓ is the number of instructions supported by the processor.[4] The verifier sends just a single field element in each step of the computation and the prover's per-step communication and computation depends only on the size of the instruction being executed in that step. Note that naïve approaches would have prover and communication complexity $O(t\ell)$, making Dora a significant improvement.
- *Generic Approach and Fiat-Shamir Friendly:* Our approach combines new techniques with insights from recent work on disjunctive zero-knowledge [HK20c, GGHAK22b, GHAKS23] and incrementally verifiable computation [KST22, KS22]. Dora only assumes the existence of a linearly homomorphic commitment scheme, the optimal choice for which can be selected based on the deployment considerations. For example, if Dora was deployed in an interactive setting, VOLE-based techniques [BCGI18, YWL$^+$20, BMRS21, YSWW21, WYKW21, BBMH$^+$21] can be used, whereas Pedersen commitments [Ped92] can be substituted when non-interactivity is desirable. If the commitment scheme is post-quantum secure, then Dora will also be post-quantum secure. Finally, the verifier in Dora is public coin, making it Fiat-Shamir friendly [FS87].[5]
- *Concretely Efficient:* Dora is concretely efficient. We implement Dora and integrate it into the `swanky` [Gal19] framework. The marginal cost of proving an additional step of computation with Dora is on the order of milliseconds. For example, if each instruction has 2^9 gates, then Dora, when run on commodity hardware (slower than a typical laptop), can prove correct execution of a program at $>$2000 steps per second—no matter how expressive the processor instruction set.

Simple Approach. Our approach for efficiently realizing a zero-knowledge protocol for RAM programs is exceedingly simple. We identify a single abstraction through which we can unify our approach to proving that memory has been handled honestly and the processor circuit has been correctly applied. We call this abstraction a *zero-knowledge bag* (ZKBag). The natural *physical analogy* of the ZKBag is an opaque bag filled with identical envelopes. A prover can insert envelopes (i.e. commitments) into this bag and later remove envelopes. Because the bag's material is opaque and all envelopes are identical, an observer cannot determine when a removed envelope was initially inserted but knows that anything removed must have, at one point, been inserted. We construct Dora from two ZKBags as follows:

(1) To ensure that memory is treated consistently, the prover and the verifier keep the active state of memory within the first ZKBag. To manipulate a memory cell, the prover simply finds the envelope holding that cell within the bag and removes it, updates it appropriately, and returns it to the bag.

(2) We let the second bag hold the intermediary states for a set of ℓ *batch proof* protocols, each corresponding to an instruction supported by the processor. In each processor step, the prover removes the state for the appropriate instruction, adds another instance of the instruction into the state, and returns the updated state to the bag. Once all steps are complete, each of the ℓ batch proofs are verified.

[3] We leave the problem of designing a succinct zero-knowledge for RAM programs with similar concrete prover efficiency to future work.

[4] We assume that all instructions are of the same size. This assumption holds without loss of generality, as we can always pad smaller instructions to match the size of the largest instruction.

[5] We provide a more detailed discussion on the application of the Fiat-Shamir transform to Dora in 8.

Concurrent Work. Two recent works [YHH+23, YH23], developed concurrently with our own, focus on designing more efficient zero-knowledge random access memory [YH23] and for proving statements with processor-like structures [YHH+23]. Although not done, it is straightforward to combine these two works to achieve a RAM zero-knowledge protocol.[6]

A crucial difference between these works and Dora, lies in the relative simplicity of our approach. For instance, in [YHH+23], Heath et al. demonstrate how to adapt specific techniques from VOLE-based zero-knowledge proof systems [DIO21, YSWW21] for proving statements with processor-like structures with optimal asymptotic complexity. Similarly, they develop separate techniques for independently handling memory accesses. In contrast, our goal in this work was to identify the simplest fundamental approach for designing zero-knowledge for RAM programs with the desired asymptotic complexity. For this, as discussed above, we formalize a single unifying primitive called zkBag and demonstrate that it suffices for proving consistency of both processor execution and memory accesses.

We think that this clean and simple abstraction effectively highlights the main challenges that must be overcome for efficiently implementing zero-knowledge for RAM programs. This, in turn, may contribute to further enhancing the practical efficiency of zero-knowledge for RAM programs in future works—perhaps by combining our approaches. We include a concrete, best-effort comparison to these concurrently developed works in Appendix A.2. We find that Dora offers notably faster proving times (1.5x-10x)[7] for processor execution but is slower (≈ 2x) at updating memory.[8] In both approaches, proving the processor execution is the main performance bottleneck (proving correct execution of processor step takes milliseconds while memory update takes microseconds), meaning Dora may offer a more promising path to production quality systems.

1.2 Related Work

Zero-knowledge for RAM programs emerged as a problem of interest following the work of Ben-Sasson et al. [BCGT13, BCG+13, BCTV14b, BCTV14a], which demonstrated that it was feasible to prove the correct execution of real RAM programs. These works laid out the primary template from which we work (discussed in Section 2.1 below). Recent works have improved performance, including the work of Heath et al. [HK20a, HYDK21, HK21], Franzese et al. [FKL+21] and Delpech de Saint Guilhem et al. [DOTV22]. These works have demonstrated concrete efficiency, but still must pay the cost of the full processor circuit in each step. Another common approach to proving correct execution of RAM programs is to "unroll" the program into an explicit circuit which can be prover with generic zero-knowledge techniques, e.g., [CK18, YSWW21, WYKW21]. The demonstration that these approaches are efficient has led to studying new applications of zero-knowledge, e.g., proofs that a program can be exploited [HK20c, GHAH+23, CHP+23].

To reduce the complexity of executing one step of the processor to be independent of the number of instructions, we leverage the disjunctive structure of processors. Zero-knowledge that is optimized for disjunctions has been the focus of foundational work on zero-knowledge [CDS94, AOS02, GMY03] and a significant number of recent work [GK15, CPS+16, Kol18, HK20c, GGHAK22b, ACF21, BMRS21, GHAKS23]. Generally, these works exploit the observation that the *prover* knows which clause of the disjunction is satisfied, and therefore the work on the remaining clauses is "wasted." This means that protocols can be designed, e.g., [HK20c, GGHAK22b, ACF21, BMRS21], that have communication complexity the depends mostly on the size of the largest clause in the disjunctions (possibly with logarithmic overhead). Our work can be seen as developing specialized disjunctive zero-knowledge techniques that compose well with RAM access and have efficient computation time.

Incrementally Verifiable Computation. Our works builds on two recent results on building incrementally verifiable computation (IVC) from folding schemes, Nova [KST22], SuperNova [KS22], Protostar [BC23] which are a part of an emerging literature on concretely efficient IVC [BGH19, BCMS20, BDFG21, BCL+21]. In Nova [KST22] , Kothapalli et al. show how to build a folding scheme for NP using a generalization of R1CS called *Relaxed R1CS* and show how it can be used to build IVC. SuperNova [KS22] and Protostar [BC23] were then proposed as extensions of Nova that support *non-uniform* IVC for "free," and discuss how to apply their techniques to verifying processor

[6] In a follow-up work [YHH+24], the authors of [YHH+23, YH23] explore this approach.

[7] The conference version of this work had a typo in Figure 11 table which was propagated throughout the rest of the text. This has been fixed in this version.

[8] The initial version of this paper shared publicly reported that Dora offered slower proving times for both processor execution and updating memory that concurrent work. This was due to an error in our evaluation that compared the performance of Dora on an 128-bit field to concurrent work on a 61-bit field. The evaluation presented in this version is more accurate.

computations.

Zero-knowledge proofs for RAM program execution can be seen as a version of non-uniform IVC where the prover must also hide *which* instructions are applied to the state at each step of the computation, but also need not be fully succinct in the number of steps. Zero-knowledge is not a goal of SuperNova, and thus we require new techniques to leverage their approach into our setting. Additionally, SuperNova's IVC reasons over the entire contents of memory, which is not concretely efficient; instead, we couple our zero-knowledge IVC with a separate protocol for managing memory consistency. Kothapalli and Setty have also recently introduced HyperNova [KS23], which aims to develop new folding schemes for NP that can be used to build more efficient IVC.

Other SNARKs. There are other prior works [WSR$^+$15, ZGK$^+$18, KPPS20, BBHR18, lib18, gen20, hod21, GPR21, MAGABMMT23, DXNT23, CGG$^+$24] that focus on building concretely efficient zkSNARKs (zero-knowledge succinct non-interactive arguments of knowledge), where the prover cost grows only with the size of the program execution. For instance, Buffet [WSR$^+$15], vRAM [ZGK$^+$18], Mirage [KPPS20], MUX-Marlin [DXNT23] and Sublonk [CGG$^+$24] that consider an "a la carte" cost profile for the provers where the prover cost for proving a step of computation grow only with the size of the circuit representing the instruction invoked on that step, i.e. independent of the number of branches. However, these schemes require a trusted common reference string setup and make use of expensive public-key operations. Works building on zkSTARKs [BBHR18, lib18, gen20, hod21, GPR21, MAGABMMT23] use a transparent (i.e. untrusted) setup and require the prover to only do work proportional to the execution trace. However, they require making a non-black box use of cryptographic hash functions. Similarly, commit and prove style SNARKs that [CFQ19, Lip16, CFH$^+$15] that have similar prover computation times also make non-black box use of cryptographic commitments. Therefore, while all of these schemes have sublinear proof sizes, their prover computation times are significantly worse than those resulting from known techniques for zero-knowledge with non-sublinear sized proofs.

2 Technical Overview

We now give an overview of the key techniques we use to construct Dora. We first recall the basic template to achieving zero-knowledge for RAM programs before proceeding to Dora itself.

2.1 Background: Template for RAM Zero-knowledge

As discussed earlier, while zero-knowledge has primarily been studied in the circuit model (i.e., where the relation for the NP language is represented as a circuit over a finite field), a significant line of work has studied how to achieve zero-knowledge for RAM programs [BCGT13, BCG$^+$13, BCTV14b, HK20a, HYDK21, GHAH$^+$23]. The key idea in these works is to bootstrap from circuit zero-knowledge to RAM zero-knowledge by representing the RAM machine on which the program should be evaluated as an explicit circuit. The prover can then use this circuit as a state transition function, and show (in zero-knowledge) that repeatedly applying this circuit t times to some initial inputs, results in a desired final processor state.

More concretely, the prover and verifier represent the RAM machine using two components: (1) a *processor circuit* C_{proc}, and (2) a *memory checker circuit* C_{mem}. C_{proc} takes as input, values fetched from memory and implements a set of valid instructions $I = \{I_1, \ldots, I_\ell\}$, ensuring that only one of these is evaluated at every step over the inputs. For example, the I_i instruction might add values, test values for equality, or modify the processor state to affect control flow, etc… The result of this evaluation can then be stored back in memory.[9] The memory checker circuit C_{mem} enforces that memory is treated consistently—that is, when a value is read from a particular memory address, C_{mem} checks to make sure that the value corresponds exactly to the last value written to that memory address.

Because most approaches for instantiating zero-knowledge for RAM program relies on this bootstrapping approach, the key determinant of efficiency is the *size* of the circuits required to implement the functionality C_{proc} and C_{mem}.

- *Current Approaches to* C_{proc}. Prior work has emphasized the need for a *small* C_{proc}, at the expense of expressiveness. For example, Ben-Sasson et al. [BCG$^+$13] describe a minimal C_{proc} called TinyRAM, which contains 27

[9] Hardware architectures also have local memory, i.e., registers and program counter, within the processor circuit. For the purposes of this overview, we elide these low level details, but note that they can either be handled as *state* within the processor circuit or simply as a specially named memory region.

instruction that can be represented in ≤ 972 gates.[10] This is because the final circuit contains t copies of C_{proc}, and t can be very large (e.g. imagine t is in the hundreds of thousands, or more). Thus, if a particular instruction I_i is very rarely used (in an average program), the prover and verifier still *pay* for that instruction in each step of the program execution. It may be more efficient to instead *emulate* I_i using a sequential series of other instructions, increasing the value of t while effectively reducing the costs of each of the t steps. In practice, this emulation approach is concretely efficient—executing a RAM program on a TinyRAM only increases t by a multiplicative factor of 2-6x compared to x86, which contains hundreds of instructions.

– *Current Approaches to* C_{mem}. There are two primary approaches to checking the consistency of memory accesses discussed in prior works: (1) leverage an efficient oblivious RAM (ORAM) construction, or (2) use a *permutation proof*. In the former approach, the prover stores tuples of the form $(\textsc{address}, \textsc{value})$ within an ORAM (eg. [MRS17]), which is either maintained by the verifier (if the proof will be executed interactively) or represented in a non-black box manner within C_{mem}. Since ORAM constructions hide access patterns and can guarantee consistency, the verifier can be confident that memory has been treated honestly without learning anything about the program execution. The other approach has the prover generate a *memory trace* of all reads and writes during program execution. The prover then permutes this trace to be sorted by address (tie-broken by timestamp), and C_{mem} needs to only check that neighboring elements of the sorted trace are internally consistent. This latter approach has been found to be more efficient in practice, and is the primary approach used in work focused on concrete efficiency [FKL+21, DOTV22, GHAH+23].

2.2 Zero-Knowledge Bag

At the heart of our construction is a new, unifying primitive that we introduce called a *zero-knowledge bag* (or ZKBag). We begin by describing this building block and then show how it can be used to instantiate Dora. We require that a ZKBag—the digital equivalent of a physical, opaque bag—provides the following (informal) guarantees:

1. *Unique Removal*: Once an element has been retrieved from the ZKBag, it cannot be retrieved again (unless, of course, it is re-inserted).
2. *Ordered Binding*: Every element that is retrieved from the bag is exactly one of the elements that was previously inserted into the ZKBag.
3. *Order Hiding*: The act of retrieving an element from the ZKBag reveals nothing about when that element was inserted.

Clearly, in order to realize the *order hiding* property, elements cannot be inserted into the ZKBag in the clear, or else a verifier could trivially link insertions and retrievals based on the value itself. As such, we insert and remove cryptographic *commitments*; when the prover wants to remove a value, it creates a *new, fresh* commitment to the value and convinces the verifier that the value therein corresponds to a value currently within the bag. This process should also remove the committed value from the bag.

Looking ahead, ZKBag provides the right combination between binding and pattern hiding required to construct zero-knowledge for RAM programs. The relationship between ZKBag and memory consistency should be clear: writing to memory corresponds exactly to inserting a $(\textsc{address}, \textsc{value})$ tuple into a ZKBag, and reading from memory corresponds exactly to retrieving a $(\textsc{address}, \textsc{value})$ tuple from a ZKBag. We will also use a ZKBag to hold the instruction set I for the processor, and have the prover pick out one instruction to be evaluated in each processor step (before reinserting it).

Constructing a ZKBag. It is clear to see that ZKBag is closely reminiscent of many existing cryptographic primitives. If *unique removal* were not required, ZKBag could be realized directly with *set membership proofs*, a concretely efficient primitive that has been the subject of immense recent study (eg. [RST01, CCs08, BCF+21, GGHAK22a, CGT23]). To achieve *unique removal*, it is clear that some kind of oblivious revocation is required, a technique that has been used in multiple other contexts, eg. ZCash [MGGR13]. However, a set membership based approach will require that the

[10]For simplicity, we do not yet make a distinction between the number of gates needed to *compute* the instructions and the number of gates needed to *verify* that a claimed evaluation is correct. In practice, we always mean the latter.

statement for each retrieval *grows* as the protocol continues.[11] Ideally, we want each insertion and retrieval to require a *constant* amount of communication and computation, as these interfaces will be called many (ie. $\mathcal{O}(t)$) times.

To achieve constant overhead, we batch checks required for *ordered binding* and *unique removal* across all insertions and retrievals, deferring the verification until the end of the protocol. In more detail, the prover and verifier maintain two lists of commitments: a list of insertions $\mathcal{I}$ and a list of retrievals $\mathcal{R}$. Each time the prover wants to insert a value v_i into the ZKBag, the verifier provides a uniformly random *tag* tag_i to the prover. The prover forms a *hiding commitment* $\mathsf{com}_{\mathsf{v}_i}$ as $\mathsf{com}_{\mathsf{v}_i} = \mathsf{Com}(v_i)$ and the parties jointly form a *public/non-hiding commitment* $\mathsf{com}_{\mathsf{tag}_i}$ as $\mathsf{com}_{\mathsf{tag}_i} = \mathsf{Com}(\mathsf{tag}_i)$ with shared randomness. Both parties add $(\mathsf{com}_{\mathsf{tag}_i}, \mathsf{com}_{\mathsf{v}_i})$ to their respective insertion list $\mathcal{I}$. When retrieving a value v_j from the ZKBag, the prover recalls the tag tag_j generated during insertion, creates the *hiding* commitment tuple $(\mathsf{com}_{\mathsf{tag}_j} = \mathsf{Com}(\mathsf{tag}_j), \mathsf{com}_{\mathsf{v}_j} = \mathsf{Com}(v_j))$ using fresh randomness and both parties add $(\mathsf{com}_{\mathsf{tag}_j}, \mathsf{com}_{\mathsf{v}_j})$ to the their respective retrieval list $\mathcal{R}$.

When the protocol ends, the prover retrieves any remaining values from the bag (i.e., it empties the bag) and gives a permutation proof demonstrating that there exists a permutation ϕ such that $\mathcal{I} = \phi(\mathcal{R})$. It is easy to see that *read-only access* to the ZKBag can be accomplished by removing a tuple $(\mathsf{com}_{\mathsf{tag}}, \mathsf{com}_{\mathsf{v}})$ from the bag and immediately re-inserting the same (non-rerandomized) value commitment with a freshly generated tag (ie. the tuple $(\mathsf{com}'_{\mathsf{tag}}, \mathsf{com}_{\mathsf{v}})$).

Intuitively, the use of hiding commitments provides the necessary *order hiding* property, and the tags provides both the *ordered binding* and *unique removal* properties. Specifically, a prover who wanted to remove an item that has not yet been inserted would need to predict the tag that the verifier would generate for that value in the future. Similarly, if an adversary removes the same value from the ZKBag twice, it must produce a *second* valid tag corresponding to the value. If the prover re-uses the same tag twice, there will be a mismatch in the tags in $\mathcal{I}$ and $\mathcal{R}$, and if it uses a new tag, it must predict a tag the verifier will generate in the future. This construction is highly efficient. Each insertion and removal requires preparing and sending only two commitments. The batched check can be done with constant communication and linear computation using a Neff-style commit-and-prove style permutation proof [Nef01] (which we describe in Section 3.5).

2.3 Constructing Dora using ZKBag

In our work, we approach the problem of constructing efficient zero-knowledge for RAM programs at the *protocol* level, rather than trying to optimize the choice of circuits C_{proc} and C_{mem}.

Expressiveness Comes Free in Zero-Knowledge. The result is Dora, a protocol for RAM zero-knowledge that transcends the seemingly inherent tradeoff between processor expressiveness (i.e., $|I| = \ell$) and execution trace length (i.e., t) altogether, and instead shows that processor expressiveness can come (nearly) *free*[12]—both in terms of computation and communication.

As with prior attempts, Dora can be decomposed into a memory component and a processor instruction handling component, each of which we realize with ZKBag. Before describing the techniques that we use in Dora, we briefly recall our efficiency goals for each component:

- *Efficiency Goals for Memory Component:* During each step of execution, the prover will fetch (1) the value stored at the address indicated by the program counter, and (2) fetch a single value from memory and write a single value to memory, as either (or both) might be necessary for the next instruction. We require that the computation and communication complexity of each fetch and write must be *constant*.
- *Efficiency Goals for Processor Instruction Component:* During each step of execution, the prover will evaluate a *single* instruction on the processor state, where the instruction is determined by the value fetched in (1) above. We require that the communication and computation complexity of each step of execution is *independent* of $|I|$.

We now discuss how to achieve both of our goals using ZKBag.

Handling Memory in Dora using ZKBag. As noted above, handling memory access with ZKBag is straightforward, as ZKBag's properties are virtually identical to those required for memory consistency. The prover and the verifier

[11] We note that there is a recent line of work showing the set membership—and disjunctive zero-knowledge more generally—can be achieved with very low overhead as the statement size grows. While it may be possible to construct ZKBag from these primitives, we instead pursue another approach discussed below.

[12] In particular, we do not need to pay the cost of processor expressiveness at each step of the processor execution.

begin by initializing the memory space by inserting public tuples (ADDRESS, VALUE) into ZKBag for every ADDRESS in the memory space, including the program code and the rest of the initial memory state (e.g. the initial stack and heap) of the execution. When proving a step of the computation, the prover interacts with the memory store three times[13]:

(1) The prover begins by reading the next instruction from memory and loading it into the processor state. This is a read-only operation, which the prover achieves by removing and re-inserting the same value (i.e. the same commitment).
(2) The prover also reads a value from memory into the processor state in case the instruction that will be run in the next instruction needs to read memory (e.g. for a LOAD instruction). Just as above, this read is read-only. Note that the prover must always perform this read in every step of the computation in order to hide any witness-dependent read patterns.
(3) Finally, the prover performs an update to one address in memory in case the instruction run in that step is a STORE instruction. This write instruction requires removing an element from the ZKBag and then rewriting to the same address with a new value from the processor state.[14] If the instruction does not require performing a write instruction, the prover can simply rewrite the initial value leaving memory functionally unchanged.

Soundness follows directly from the *unique removal* and *ordered binding* properties of the ZKBag (discussed above), as these properties guarantee that the verifier knows that each values read from memory must be "current." Zero-knowledge relies on the *order hiding* property to hide the memory addresses being manipulated.

Using this protocol, the total complexity of managing memory in Dora is only three tuple insertions and three tuple removals per step of the computation, but this can be reduced because the prover does not need to resend the same commitments multiple times.

Handling Processor Instructions in Dora using ZKBag. During each step of processor execution, the prover needs to convince the verifier that a processor state st_{i+1} is the result of applying *one* of the instructions in the instruction set to the previous processor state st_i, without revealing which instruction was applied. We begin by giving a baseline approach for achieving our goal before proceeding to optimize the approach to improve concrete performance.

Baseline Approach. A straightforward approach would be to use a *set membership* proof; the prover could generate a commitment to the executed instruction and then provide a proof that the contents of the commitment are a valid instruction. This commitment can then be added to the statement for another zero-knowledge proof that demonstrates the transition from st_i to st_{i+1}. This approach, while intuitive, has two primary downfalls:

(1) While there has been a tremendous amount of work on set membership proofs, state-of-the-art protocols have a *logarithmic* size in the number of elements in the set and a *linear* computation complexity in the size of the set (e.g., [HK20c, HK20b, GGHAK22a, GGHAK22b]). While in practice it might be acceptable to tolerate the communication overhead, linear computation complexity may be unreasonable for large instruction sets. Moreover, our aim in this work is to achieve *constant* overhead—both in terms of communication and computation. While SNARKs might be a way to achieve our goals for the verifier, given SNARK's succinctness and constant-time verification, there is not an obvious way to use this set membership approach to get constant overhead for the prover.
(2) Given a commitment to the step's instruction I, the prover must then prove that st_{i+1} is the result of applying I to st_i. Doing this efficiently is non-trivial, given that the statement of interest is in committed form. A very natural approach to would be to combine non-black box use of the commitment scheme and universal circuits (ie. prove that I is in the commitment and that $U(I, \mathsf{st}_i) = \mathsf{st}_{i+1}$, where U is a universal circuit of the appropriate size), and then prove the resulting statement using generic, circuit zero-knowledge. Unfortunately, both non-black box use of cryptography and universal circuits tend to be highly inefficient, making this approach unattractive. It might be possible to design very specific zero-knowledge proofs that naturally interoperate the chosen commitment scheme

[13]We assume that the processor here has a simple load store architecture and all instructions in the instruction set read and write at most a single value. In more complex architectures (eg. architectures that support indirect loads) additional interactions with memory may be necessary. Supporting these instructions is trivial.

[14]Ensuring that the read and write are to the same memory location can be easily ensured by reusing the address commitment retrieved during the removal.

to avoid the non-black box use of cryptography, but this approach would reduce the flexibility and modularity of our construction.

As such, the seemingly natural approach to handling processor instruction in Dora appears to be unfruitful. Instead, we investigate how ZKBag could be used to design a more efficient approach. As already demonstrated with memory management, ZKBag provides a highly efficient (ie. constant overhead) way to obliviously select elements from a set. As such, it seems natural to substitute the set-membership proof in the above template with ZKBag, resolving problem (1). However, using ZKBag in this way does nothing to resolve problem (2). As such, we require a slightly more nuanced approach to using ZKBag in order to achieve our result.

Combining ZKBag and Relaxed R1CS to Achieve Constant Overhead. Rather than store instructions in a ZKBag, we build on an approach from prior works on IVCs [KST22, KS22] and store a set of *accumulators* in the ZKBag—one accumulator for each instruction in the instruction set. Executing a step of the processor involves obliviously retrieving the appropriate accumulator from the ZKBag and updating it. The intuition behind this approach is to use these accumulators to iteratively update NP statements at each step, such that the prover can simultaneously verify the final accumulated set of $|\mathcal{I}|$ statements at the end of the protocol. These accumulators are carefully designed such that the prover's knowledge of a valid witness at the end of the protocol for each accumulated statement demonstrates that *each* step was correctly executed. The benefit of this approach is that the computationally expensive zero-knowledge proofs are deferred until the end of the protocol, requiring only a single zero-knowledge proof for each instruction rather than for each step, further improving Dora's concrete efficiency.

To instantiate these accumulators, we leverage *Relaxed R1CS folding*, an approach described by [KST22]. Relaxed R1CS is a natural extension to standard R1CS such that there can be additional error terms. A typical R1CS relation is constructed by matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and an instance $\vec{\mathsf{x}}$ is satisfied if there exists a witness $\vec{\mathsf{w}}$ such that $\mathbf{A} \cdot \vec{\mathsf{z}} \circ \mathbf{B} \cdot \vec{\mathsf{z}} = \mathbf{C} \cdot \vec{\mathsf{z}}$, where $\vec{\mathsf{z}} = \vec{\mathsf{w}} \| \vec{\mathsf{x}}$. A relaxed R1CS relation injects two additional *error* parameters, $\mathsf{u} \in \mathbb{F}$ and $\vec{\mathsf{e}} \in \mathbb{F}^m$, and is satisfied if there exists a $\vec{\mathsf{z}} = \vec{\mathsf{w}} \| \vec{\mathsf{x}} \| \mathsf{u}$ such that $(\mathbf{A} \cdot \vec{\mathsf{z}}) \circ (\mathbf{B} \cdot \vec{\mathsf{z}}) = \mathsf{u} \cdot (\mathbf{C} \cdot \vec{\mathsf{z}}) + \vec{\mathsf{e}}$. The power of relaxed R1CS is that it permits *folding*: given a fixed relation $\mathbf{A}, \mathbf{B}, \mathbf{C}$, and two instances $(\vec{\mathsf{x}}_1, \mathsf{u}_1, \vec{\mathsf{e}}_1)$ and $(\vec{\mathsf{x}}_2, \mathsf{u}_2, \vec{\mathsf{e}}_2)$, it is possible to combine the two into a new instance $(\vec{\mathsf{x}}, \mathsf{u}, \vec{\mathsf{e}})$ for the same relation $\mathbf{A}, \mathbf{B}, \mathbf{C}$. Importantly, a prover can only satisfy the new instance $(\vec{\mathsf{x}}, \mathsf{u}, \vec{\mathsf{e}})$ if they had valid witnesses $\vec{\mathsf{w}}_1, \vec{\mathsf{w}}_2$ to the initial instances (except with negligible probability). We defer the details of this folding procedure to Section 3.2.

Dora leverages this technique as follows: the prover and verifier initialize a ZKBag and (publicly) insert a relaxed R1CS instance (as defined by $\vec{\mathsf{e}}$ and $\vec{\mathsf{z}}$) for each instruction into the ZKBag that will be used as an accumulator. During each step of the computation, the prover retrieves the instance corresponding to the current instruction and prepares a new instance for the current instruction using the committed processor state and the values retrieved from memory. The prover then folds the state of the accumulator with the newly prepared instance, locally updating the witness required to satisfy the folded instance. Finally, the prover inserts the folded instance into the ZKBag and continues to the next step. After all the steps have been run, the prover removes the final accumulator for each instruction from the ZKBag and opens them to the verifier. The prover and verifier then engage in a generic zero-knowledge proof for the final relaxed R1CS instances. We note that there are several low-level details we have omitted in this description for clarity (e.g., the final instances must be randomized to satisfy zero-knowledge).

Putting It All Together. Dora is realized by combining the techniques described above for memory management and proving the correctness of instruction executions. In each step, the prover retrieves the appropriate values from memory and adds them to the (committed) processor state. The prover then uses the processor state to construct a relaxed R1CS instance that would prove correct execution of the instruction and folds it into the accumulator for the instruction executed in that step. Finally, the prover updates a memory location to emulate a store instruction. Once all of the steps have been completed, the prover opens all the accumulators and proves that it has a witness to each one.

We benchmark Dora in Section 8 and show that it is highly efficient. Because of its nice asymptotics, Dora can prove correct execution of RAM programs on massive processors (thousands of instruction with thousands of gates each) in milliseconds per step.

3 Preliminaries

In this section, we recall some prelimary definitions. In Section 3.1, we present a definition of linearly homomorphic commitments. In Section 3.3, we recall the definition of a commit-and-prove zero-knowledge protocol. In Section 3.2, we provide a formal overview of relaxed R1CS [KST22]. In Section 3.4, we recall a construction of commit-and-prove ZK for R1CS (implicit in [KST22]). Finally, in Section 3.5 we recall the construction of Neff-style [Nef01] multi-set equality proofs.

Notation. Let t be the number of steps in the program trace, ℓ be the number of instructions in the processor circuit, m be the number of addresses in memory.

3.1 Linearly Homomorphic Commitments

Our construction makes use of a standard linearly homomorphic commitment primitive, which we define below. We intentionally give a general enough definition of this primitive that can capture both *interactive* instantiations (eg. VOLE-based [BMRS21]) and *non-interactive* instantiations (eg. Pedersen [Ped92]).

Definition 1 (Linearly Homomorphic Commitments)**.** *Linearly homomorphic commitments comprise of a tuple of four interactive protocols* $\pi^{\mathsf{LCom}} = (\pi^{\mathsf{LCom}}_{\mathsf{Setup}}, \pi^{\mathsf{LCom}}_{\mathsf{Commit}}, \pi^{\mathsf{LCom}}_{\mathsf{Open}}, \pi^{\mathsf{LCom}}_{\mathsf{Comb}})$ *between a Sender* Sen *and receiver* Rec *and a PPT algorithm* $\mathsf{Equiv}^{\mathsf{LCom}}$ *defined as follows:*

- $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Setup}}$*: The setup protocol generates any needed public parameters* pp*, a sender key* skey *as output for the sender and a receiver key* rkey *as output for the receiver.*
- $((\mathsf{com}, \mathsf{op}), (\mathsf{com})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}$*: The commit protocol takes the value* val *to be committed as input from the sender and outputs a commitment* com *to both the sender and the receiver. It additionally outputs* op *to the sender.*
- $((b), (\mathsf{val}')) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Open}}$*: Both the sender and receiver invoke the opening protocol using a commitment* com *as input. The sender additionally inputs a value* val *committed inside this commitment and the associated opening information* op*. This protocol outputs a value* $\mathsf{val}' \in \{\mathsf{val}, \perp\}$ *to the receiver and a bit* $b \in \{0, 1\}$ *to the sender indicating whether or not* $\mathsf{val}' =^? = \mathsf{val}$.
- $((\mathsf{com}, \mathsf{op}), (\mathsf{com})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Comb}}$*: The linear combination protocol takes* $(\mathsf{pp}, \mathsf{skey}, f_{\mathsf{lin}}, \mathsf{com}_1, \mathsf{op}_1, \mathsf{com}_2, \mathsf{op}_2)$ *as input from the sender and* $(\mathsf{pp}, \mathsf{rkey}, f_{\mathsf{lin}}, \mathsf{com}_1, , \mathsf{com}_2)$ *as input from the receiver. It computes the function* f_{lin} *on* com_1 *and* com_2 *and outputs the resulting new commitment* com *and its corresponding opening information* op.
- $\mathsf{op} \leftarrow \mathsf{Equiv}^{\mathsf{LCom}}(\mathsf{pp}, \mathsf{rkey}, \mathsf{com}, \mathsf{val})$*: The equivocation algorithm and outputs the new opening information* op *corresponding to* com *and* val.

We require that the scheme satisfies standard hiding*. For* binding*, we assume that the commitment scheme has an extractor that can extract the val within a commitment. In addition to these standard properties, we assume that the* $\pi^{\mathsf{LCom}}_{\mathsf{Comb}}$ *algorithm allows the sender and receiver to perform linear operations over commitments and we assume that the receiver can always equivocate. Formally, these properties are defined as follows:*

1. ***Hiding:*** *Let* $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Setup}} \langle \mathsf{Sen}(1^\lambda), \mathsf{Rec}(1^\lambda) \rangle$ *be an honest execution of the setup protocol. For any* $\mathsf{val}_1, \mathsf{val}_2 \in \mathcal{V}$*, the view of* Rec *remains computationally indistinguishable in the following two executions:*

$$\pi^{\mathsf{LCom}}_{\mathsf{Commit}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{val}_1), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}) \rangle$$

$$\pi^{\mathsf{LCom}}_{\mathsf{Commit}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{val}_2), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}) \rangle$$

2. ***Equivocation:*** *Let* $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Setup}} \langle \mathsf{Sen}(1^\lambda), \mathsf{Rec}(1^\lambda) \rangle$ *be an honest execution of the setup protocol. The following holds* $\forall\ \mathsf{val} \in \mathcal{V}$ *and every honest execution of the commit protocol* $((\mathsf{com}, \mathsf{op}), (\mathsf{com})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{val}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}) \rangle$*: if* $(\mathsf{val}', \mathsf{op}') \leftarrow \mathsf{Equiv}^{\mathsf{LCom}}(\mathsf{pp}, \mathsf{rkey}, \mathsf{com})$*, then for an honest sender and receiver,*

$$\Pr[((1), (\mathsf{val}')) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Open}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{com}, \mathsf{op}', \mathsf{val}'), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{com}) \rangle] \geq 1 - \mathsf{neg}(\lambda)$$

3. ***Linear Homorphism:*** *Let* $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Setup}} \langle \mathsf{Sen}(1^\lambda), \mathsf{Rec}(1^\lambda) \rangle$ *be an honest execution of the setup protocol. The following holds for all* $\mathsf{val}_1, \mathsf{val}_2 \in \mathcal{V}$*, every linear function* $f_{\mathsf{lin}} : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ *and all honest executions of the commit protocol* $(\forall\, i \in [2])\ ((\mathsf{com}_i, \mathsf{op}_i), (\mathsf{com}_i)) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{val}_i), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}) \rangle$*: if*

$$((\mathsf{com}, \mathsf{op}), (\mathsf{com})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Comb}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, f_{\mathsf{lin}}, \mathsf{com}_1, \mathsf{op}_1, \mathsf{com}_2, \mathsf{op}_2), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, f_{\mathsf{lin}}, \mathsf{com}_1, , \mathsf{com}_2) \rangle ,$$

then for an honest sender and receiver,

$$\Pr[((1), (f_{\mathsf{lin}}(\mathsf{val}_1, \mathsf{val}_2))) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Open}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{com}, \mathsf{op}, f_{\mathsf{lin}}(\mathsf{val}_1, \mathsf{val}_2)), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{com}) \rangle] \geq 1 - \mathsf{neg}(\lambda)$$

4. ***Binding/Extraction:*** *Let* $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Setup}} \langle \mathsf{Sen}(1^\lambda), \mathsf{Rec}(1^\lambda) \rangle$ *be an honest execution of the setup protocol. There exists an extractor* $\mathcal{E}$*, such that for any PPT adversary* $\mathcal{A}$ *and for any* com *such that* $((\cdot), (\mathsf{com})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}} \langle \mathcal{A}(\mathsf{pp}, \mathsf{skey}, \cdot), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}) \rangle$*, then* $(\mathsf{val}) \leftarrow \mathcal{E}^{\mathcal{O}(\mathcal{A})}(\mathsf{pp})$ *such that for any honest receiver and* $\mathsf{val} \neq \mathsf{val}' \neq \perp$*, it holds that*

$$\Pr[((\cdot), (\mathsf{val}')) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Open}} \langle \mathcal{A}(\mathsf{pp}, \mathsf{skey}, \mathsf{com}, \cdot), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{com}) \rangle] \leq \mathsf{neg}(\lambda)$$

Short-Hand Notation. For simplicity, we use the notation $[\![v]\!]$ denotes a commitment to some value $\vec{v}$. We often abuse notation and use $[\![\vec{x}]\!]$ to denote a linearly homomorphic commitment to a vector of elements in $\vec{x} \in \mathbb{F}^*$. We use linear arithmetic operations as a short-hand for $\pi^{\mathsf{LCom}}_{\mathsf{Comb}}$, e.g., $[\![\mathsf{val}]\!] = c_1 \cdot [\![\mathsf{val}_1]\!] + [\![\mathsf{val}_2]\!]$, where c_1 is some public value. Finally, we remark that the by default, the above definition of $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ is presented for *private commitments*, i.e., it only takes the value to be committed as input from the sender. However, it can easily be adapted to allow for *public commitments*, where both the sender and receiver have access to the value being committed. It that case, we assume that in addition to taking val as input from both parties, $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ is run on shared randomness between the sender and receiver.

3.2 Relaxed R1CS

Exisiting proof systems works with different representations of the relation they are proving. The most popular representation amongst state-of-the-art proof systems is known as the rank 1 constrained system (or R1CS) that generalizes arithmetic circuits. In this work, we use *Relaxed R1CS*, a generalization of R1CS introduced by Kothapalli, Setty and Tzialla [KST22]:

Definition 2 (Relaxed R1CS, [KST22]). *A relaxed R1CS (Rank-1 Constraint System) [KST22] is defined by three matrixes* $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}^{m \times m}$*. A witness* w *satisfies an instance* $(\vec{\mathsf{e}}, \vec{\mathsf{x}}, \mathsf{u})$ *iff. the "extended witness"* $\vec{\mathsf{z}} = \vec{\mathsf{w}} \| \vec{\mathsf{x}} \| \mathsf{u} \in \mathbb{F}^m$ *satisfies:* $(\mathbf{A} \cdot \vec{\mathsf{z}}) \circ (\mathbf{B} \cdot \vec{\mathsf{z}}) = \mathsf{u} \cdot (\mathbf{C} \cdot \vec{\mathsf{z}}) + \vec{\mathsf{e}}$*. For ease of notation, refer to Relaxed R1CS instances by their extended witness* $\vec{\mathsf{z}}$ *and error term* $\vec{\mathsf{e}}$*, which in turn defines* $\vec{\mathsf{w}}$*,* $\vec{\mathsf{x}}$*, and* u*.*

One valuable feature of Relaxed R1CS instances, as noted by [KST22], is that they can be "folded." That is, given two Relaxed R1CS instances $(\vec{\mathsf{z}_1}, \vec{\mathsf{e}_1})$ and $(\vec{\mathsf{z}_2}, \vec{\mathsf{e}_2})$ and a randomly sampled $r \in \mathbb{F}$, we can define a new instance $(\vec{\mathsf{z}}, \vec{\mathsf{e}})$ as:

$$\vec{\mathsf{e}} = \vec{\mathsf{e}_1} + r \cdot \vec{T} + r^2 \cdot \vec{\mathsf{e}_2}, \quad \mathsf{u} = \mathsf{u}_1 + r \cdot \mathsf{u}_2, \quad \vec{\mathsf{z}} = \vec{\mathsf{z}_1} + r \cdot \vec{\mathsf{z}_2}, \text{ where}$$
$$\vec{T} = \mathbf{A} \cdot \vec{\mathsf{z}_1} \circ \mathbf{B} \cdot \vec{\mathsf{z}_2} + \mathbf{A} \cdot \vec{\mathsf{z}_2} \circ \mathbf{B} \cdot \vec{\mathsf{z}_1} - \mathsf{u}_1 \cdot \mathbf{C} \cdot \vec{\mathsf{z}_2} - \mathsf{u}_2 \cdot \mathbf{C} \cdot \vec{\mathsf{z}_1}$$

Importantly, this folding process is *sound*, in that if either $(\vec{\mathsf{z}_1}, \vec{\mathsf{e}_1})$ or $(\vec{\mathsf{z}_2}, \vec{\mathsf{e}_2})$ are not satisfied, then $(\vec{\mathsf{z}}, \vec{\mathsf{e}})$ is also unsatisfied with high probability (over the choice of r). An additional fact about the folding scheme above (not directly used in Nova [KST22]) is that the folding *only depends on the dimensions of* $\mathbf{A}$, $\mathbf{B}$ *and* $\mathbf{C}$. This means that we can have the verifier "fold" two committed instances pairs without revealing the relation these instances belong. This will be crucial as we will be executing the folder "obliviously," in that only the prover will know which instance is being considered.

Remark (*R1CS is a Special Case of Relaxed R1CS).* Note that regular R1CS is captured as the special case of Definition 2 where $\vec{\mathsf{e}} = \vec{0} \in \mathbb{F}^m$ and $\mathsf{u} = 1$. Throughout the section, to simplify notation, we will refer to relaxed R1CS instances by their error term $\vec{\mathsf{e}} \in \mathbb{F}^m$ and extended witness $\vec{\mathsf{z}} \in \mathbb{F}^m$; which define $\vec{\mathsf{w}}$, $\vec{\mathsf{x}}$, u.

3.3 Commit-and-Prove Zero-Knowledge

Both our final construction Dora and our subprotocol for handling processor instructions are custom-designed commit-and-prove style zero-knowledge for specific languages. In this section, we recall the definition of this primitive. We assume that the commitments in this definition were computed using inearly homomorphic commitments defined in Section 3.1.

Definition 3 (LinCom-Based Commit-and-Prove ZK)**.** *LinCom-based commit-and-prove zero-knowledge proof system for an NP-relation $\mathcal{R}$, comprises of a tuple of 3 interactive protocols $(\pi_{\mathsf{Setup}}, \pi_{\mathsf{Proof}}, \pi_{\mathsf{Verify}})$ between the sender and receiver defined as follows:*

- $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{ZK}}_{\mathsf{Setup}}$*:The setup protocol generates any needed public parameters* pp*, a sender key* skey *as output for the sender/prover and a receiver key* rkey *as output for the receiver/verifier.*
- $((\mathsf{Proof}^{\mathsf{ZK}}, \mathsf{st}), (\mathsf{Proof}^{\mathsf{ZK}})) \leftarrow \pi^{\mathsf{ZK}}_{\mathsf{Prove}}$*: The prove protocol takes as input* $(\mathsf{pp}, \mathsf{skey}, \overrightarrow{x}, \mathsf{com}, \overrightarrow{\mathsf{op}}, \overrightarrow{w})$ *from the sender/prover and* $(\mathsf{pp}, \mathsf{rkey}, \overrightarrow{x}, [\![\overrightarrow{w}]\!])$ *from the receiver/verifier. It outputs a proof* $\mathsf{Proof}^{\mathsf{ZK}}$ *that allows the prover/sender to convince the receiver/verifier that it knows* $\overrightarrow{w}, \overrightarrow{\mathsf{op}}$ *such that they are a valid opening for* $[\![\overrightarrow{w}]\!]$ *and* $\overrightarrow{w}$ *is a valid witness for statement* $\overrightarrow{x}$*. This protocol may additionally output some secret state* st *for the sender/prover.*
- $((b), (b)) \leftarrow \pi^{\mathsf{ZK}}_{\mathsf{Verify}}$*: The verify protocol takes as input* $(\mathsf{pp}, \mathsf{skey}, \mathsf{Proof}^{\mathsf{ZK}}, \mathsf{st}, \overrightarrow{x})$ *from the sender/prover and* $(\mathsf{pp}, \mathsf{rkey}, \mathsf{Proof}^{\mathsf{ZK}}, \overrightarrow{x})$ *from the receiver/verifier and outputs a bit* $b \in \{0, 1\}$*, based on whether or not the proof* $\mathsf{Proof}^{\mathsf{ZK}}$ *verifies.*

We require the above protocols to satisfy the standard notions of correctness*,* zero-knowledge *and* knowledge soundness.

3.4 Commit-and-Prove ZK for R1CS

Next, we recall a simple Σ-protocol for R1CS-satisfiability. This protocol is derived directly from the Nova [KST22] IVC scheme. This protocol satisfies all the properties that we need from a commit and prove zero-knowledge protocol defined in Section 3.3. Let $(\mathbf{A}, \mathbf{B}, \mathbf{C})$ be an R1CS instance. Given a commitment $[\![\overrightarrow{\mathsf{z}}]\!]$, computed using a linearly homomorphic commitment (see Section 3.1), the prover wants to convince the verifier that the value $\overrightarrow{\mathsf{z}} = \overrightarrow{\mathsf{w}} \| \overrightarrow{\mathsf{x}} \| \mathsf{u}$ committed inside this commitment is a valid extended witness for $(\mathbf{A}, \mathbf{B}, \mathbf{C})$. The setup algorithm $\pi^{\mathsf{ZK}}_{\mathsf{Setup}}$ of this proof system is the same as the setup of the above linearly homomorphic commitment scheme. We now describe the $\pi^{\mathsf{ZK}}_{\mathsf{Prove}}$ and $\pi^{\mathsf{ZK}}_{\mathsf{Verify}}$ protocols.

- Prover samples a random satisified relaxed R1CS instance as follows:
  - Sample $\overrightarrow{\mathsf{z}_0} \leftarrow\!\$\, \mathbb{F}^m$ and parse $\overrightarrow{\mathsf{z}_0} = \overrightarrow{\mathsf{w}_0} \| \overrightarrow{\mathsf{x}_0} \| \mathsf{u}_0$.
  - Set $\overrightarrow{L} \leftarrow \left(\mathbf{A} \cdot \overrightarrow{\mathsf{z}_0}\right) \circ \left(\mathbf{B} \cdot \overrightarrow{\mathsf{z}_0}\right)$, $\overrightarrow{R} \leftarrow \mathsf{u}_0 \cdot \left(\mathbf{C} \cdot \overrightarrow{\mathsf{z}_0}\right)$ and $\overrightarrow{\mathsf{e}_0} \leftarrow \overrightarrow{L} - \overrightarrow{R}$
- Prover then computes the cross terms:

$$\overrightarrow{t_1} \leftarrow \mathbf{A} \cdot \overrightarrow{\mathsf{z}} \circ \mathbf{B}_i \cdot \overrightarrow{\mathsf{z}_0} + \mathbf{A} \cdot \overrightarrow{\mathsf{z}_0} \circ \mathbf{B}_i \cdot \overrightarrow{\mathsf{z}}$$

$$\overrightarrow{t_2} \leftarrow \mathsf{u} \cdot \mathbf{C} \cdot \overrightarrow{\mathsf{z}_0} + \mathsf{u}_0 \cdot \mathbf{C} \cdot \overrightarrow{\mathsf{z}}$$

$$\overrightarrow{T} \leftarrow \overrightarrow{t_1} - \overrightarrow{t_2}$$

- Prover and verifier use $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ to compute commitment-opening pairs $(([\![T]\!], \mathsf{op}_T), ([\![T]\!]))$, $([\![\mathsf{z}_0]\!], \mathsf{op}_{\mathsf{z}_0})$ and $([\![\mathsf{z}_0]\!], \mathsf{op}_{\mathsf{e}_0})$.
- The verifier then samples and sends $r \leftarrow\!\$\, \mathbb{F}$.
- Prover uses $\pi^{\mathsf{LCom}}_{\mathsf{Comb}}, \pi^{\mathsf{LCom}}_{\mathsf{Open}}$ to open the following linear combinations of the two instances:
  - Let $\overrightarrow{\mathsf{e}'}$ be the opened value associated with the commitment $\left(r \cdot \left[\!\!\left[\overrightarrow{T}\right]\!\!\right] + r^2 \cdot [\![\overrightarrow{\mathsf{e}_0}]\!]\right)$

– Let $\overrightarrow{\mathsf{z}'}$ be the opened value associated with the commitment $\left(\overrightarrow{\mathsf{z}} + r \cdot \overrightarrow{\mathsf{z}_0}\right)$

- Finally, if the above openings are valid, the verifier checks: $\left(\mathbf{A} \cdot \overrightarrow{\mathsf{z}'}\right) \circ \left(\mathbf{B} \cdot \overrightarrow{\mathsf{z}'}\right) =^? = \mathsf{u}' \cdot \mathbf{C} \cdot \overrightarrow{\mathsf{z}'} + \overrightarrow{\mathsf{e}'}$, where $\mathsf{u}' = \mathsf{u} + r \cdot \mathsf{u}_0$.

3.5 Multi-Set Equality Proofs

In our construction of ZKBag, we leverage an efficient set equality proof (also referred to as a permutation proof). In our concrete instantiation of Dora, we use the simple Bayer-Groth style proof. To the best of our knowledge, this construction was first documented in [Nef01] and has subsequently been independently discovered in many works [BG12, FKL+21]. Given 2 sets of commitments, $\mathcal{S}_1 = (\llbracket \overrightarrow{a_1} \rrbracket, \ldots, \llbracket \overrightarrow{a_k} \rrbracket)$ and $\mathcal{S}_2 = \left(\left\llbracket \overrightarrow{b_1} \right\rrbracket, \ldots, \left\llbracket \overrightarrow{b_k} \right\rrbracket\right)$, the multi-set equality proof can be viewed as a commit-and-prove zero-knowledge protocol (say $\left(\pi^{\mathsf{ZKMultiSet}}_{\mathsf{Setup}}, \pi^{\mathsf{ZKMultiSet}}_{\mathsf{Prove}}, \pi^{\mathsf{ZKMultiSet}}_{\mathsf{Verify}}\right)$) for the following relation: *there exists a permutation* p*, such that* $p\left(\overrightarrow{a}_1, \ldots, \overrightarrow{a}_k\right) = \overrightarrow{b}_1, \ldots, \overrightarrow{b}_k$.

We now recall this well-known Bayer-Groth style [BG12] shuffle proof. We assume that all commitments were computed using linearly homomorphic commitments from Section 3.1. This is the only component in our construction that (black-box) relies on a general proof system – let $\left(\pi^{\mathsf{ZK}}_{\mathsf{Setup}}, \pi^{\mathsf{ZK}}_{\mathsf{Prove}}, \pi^{\mathsf{ZK}}_{\mathsf{Verify}}\right)$ be the commit and prove zero-knowledge protocol for general R1CS satisfiability from Section 3.4. The setup algorithm $\pi^{\mathsf{ZKMultiSet}}_{\mathsf{Setup}}$ of this proof system is the same as the setup of the above linearly homomorphic commitment scheme. We now describe the $\pi^{\mathsf{ZKMultiSet}}_{\mathsf{Prove}}$ and $\pi^{\mathsf{ZKMultiSet}}_{\mathsf{Verify}}$ protocols.

- Verifier samples random field elements $u, v \leftarrow\!\$\, \mathbb{F}$, and sends them to the prover.
- For each $i \in [k]$, both the prover and verifier use $\pi^{\mathsf{LCom}}_{\mathsf{Comb}}$ to compute

$$\llbracket \alpha_i \rrbracket = \left\langle \left(1, u^2, \ldots, u^{k-1}\right), \llbracket \overrightarrow{a_i} \rrbracket \right\rangle$$

$$\llbracket \beta_i \rrbracket = \left\langle \left(1, u^2, \ldots, u^{k-1}\right), \left\llbracket \overrightarrow{b_i} \right\rrbracket \right\rangle$$

- Finally, the prover uses $\left(\pi^{\mathsf{ZK}}_{\mathsf{Setup}}, \pi^{\mathsf{ZK}}_{\mathsf{Prove}}, \pi^{\mathsf{ZK}}_{\mathsf{Verify}}\right)$ to convince the verifier that $\prod_{i \in [k]} \left(v - \llbracket \alpha_i \rrbracket\right) = \prod_{i \in [k]} \left(v - \llbracket \beta_i \rrbracket\right)$.

4 Zero-Knowledge Bag

The heart of Dora is a zero-knowledge bag (ZKBag) protocol. This cryptographic object is analogous to a physical bag into which the prover and verifier place wrapped objects. The critical properties of the protocol are equivalent to the physical properties that such a bag would possess: only objects previously put into the bag can be removed, and the bag itself hides the correspondence between the order in which objects are inserted and removed. In some sense, the zero-knowledge bag can be seen as a "slow moving" shuffle proof augmented with a sense of time.

4.1 Defining ZKBag

Definition 4 (LinCom-Based Zero-Knowledge Bag)**.** *A ZKBag is parameterized by a linearly homomorphic commitment scheme, and as such we call the resulting cryptographic primitive a LinCom-Based ZKBag. A LinCom-Based ZKBag comprises of a tuple of 5 interactive protocols* $(\pi^{\mathsf{ZKBag}}_{\mathsf{Setup}}, \pi^{\mathsf{ZKBag}}_{\mathsf{Init}}, \pi^{\mathsf{ZKBag}}_{\mathsf{Insert}}, \pi^{\mathsf{ZKBag}}_{\mathsf{Remove}}, \pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}})$ *between the sender and receiver:*

– $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Setup}} \left\langle \mathsf{Sen}\left(1^\lambda\right), \mathsf{Rec}\left(1^\lambda\right)\right\rangle$*: The setup protocol generates any needed public parameters* pp*, generates a sender key* skey *as output for the sender and a receiver key* rkey *as output for the receiver.*

– $((\mathsf{bag}, \mathsf{state}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Init}} \left\langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\right\rangle$ *: The parties take the output of* $\pi^{\mathsf{ZKBag}}_{\mathsf{Setup}}$ *as input and initialize the ZKBag. The sender and receiver each maintain some joint information* bag *and the sender maintains some secret information* state.

– $\left(\left(\mathsf{bag}', \mathsf{state}'\right), \left(\mathsf{bag}'\right)\right) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Insert}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}, \mathsf{state}, \left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right], \mathsf{op}, \mathsf{val}\right), \mathsf{Rec}\left(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}, \left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right]\right)\right\rangle$: *The parties take in the current state of the bag* $((\mathsf{bag}, \mathsf{state}), (\mathsf{bag}))$ *and a commitment* $\left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right]$. *Additionally, the sender provides a valid opening to the commitment* $(\overrightarrow{\mathsf{val}}, \mathsf{op})$. *This updates the state of the bag held by both the sender and the receiver.*

– $\left(\left(\mathsf{bag}', \mathsf{state}'\right), \left(\mathsf{bag}'\right)\right) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Remove}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}, \mathsf{state}, \left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right], \mathsf{op}, \mathsf{val}\right), \mathsf{Rec}\left(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}, \left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right]\right)\right\rangle$: *The parties take in the current state of the bag* $((\mathsf{bag}, \mathsf{state}), (\mathsf{bag}))$ *and a commitment* $\left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right]$. *Additionally, the sender provides a valid opening to the commitment* $(\overrightarrow{\mathsf{val}}, \mathsf{op})$. *This updates the state of the bag held by both the sender and the receiver.*

– $((b), (b)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}, \mathsf{state}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag})\rangle$: *The parties take in the current state of the bag* $((\mathsf{bag}, \mathsf{state}), (\mathsf{bag}))$ *and check if the bag is empty. This outputs a bit* b *to the sender and the receiver.*

We define 3 properties of these algorithms: correctness, knowledge soundness, *and* zero-knowledge.

1. **Correctness:** *Correctness considers an interaction between the sender and receiver in which they run setup and initialize. After this first phase, the sender and receiver run an arbitrary sequence of inserts and removes. If there is a one-to-one correspondence between inserts and removes such that the remove always comes after the corresponding insert and the values in each corresponding pair are for the same values, then a call to* $\pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}$ *will return* 1 *w.h.p.*

   Formally speaking, let $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Setup}}\langle(\mathsf{Sen}(1^\lambda), \mathsf{Rec}(1^\lambda)\rangle$, $((\mathsf{bag}, \mathsf{state}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Init}}\langle\mathsf{Sen}(\mathsf{pp}, \mathsf{skey}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\rangle$ *be honest executions of the setup and initialization protocols. For any* $n \in \mathsf{poly}(\lambda)$, $\mathsf{val}_1, \ldots, \mathsf{val}_n \in \mathcal{V}$ *and any sequence of* $2n$ *executions of the insert and remove protocols such that for each* $i \in [n]$, *a protocol of the form* $\pi^{\mathsf{ZKBag}}_{\mathsf{Remove}}\langle\mathsf{Sen}(\cdots, \mathsf{com}_i, \mathsf{op}_i\mathsf{val}_i), \mathsf{Rec}(\cdots, \mathsf{com}_i)\rangle$ *only appears after* $\pi^{\mathsf{ZKBag}}_{\mathsf{Insert}}\langle\mathsf{Sen}(\cdots, \mathsf{com}'_i, \mathsf{op}'_i\mathsf{val}_i), \mathsf{Rec}(\cdots, \mathsf{com}'_i)\rangle$ *in the sequence and each of these appear exactly once, it holds that:*

   $$\Pr\left[((1), (1)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}\langle\mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}, \mathsf{state}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag})\rangle\right] \geq 1 - \mathsf{neg}(\lambda)$$

   Here for each $i \in [n]$, com_i *and* com'_i *are commitments of the form:*

   $$((\mathsf{com}_i, \mathsf{op}_i), (\mathsf{com}_i)) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}\langle\mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{val}_i), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\rangle \textit{ and}$$

   $$((\mathsf{com}'_i, \mathsf{op}'_i), (\mathsf{com}'_i)) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}\langle\mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{val}'_i), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\rangle.$$

2. **Knowledge Soundness:** *Knowledge soundness intuitively says that a malicious sender cannot (w.h.p.) convince the receiver that the bag is empty after an interaction unless all the restrictions on the interaction from* correctness *hold and the bag truly is empty. We formalize this by saying that there exists an extractor that can extract the values used in the insertions and removals, such that (as above) there is a one-to-one correspondence between inserts and removes such that the remove always comes after the corresponding insert and the values in each corresponding pair are for the same values.*

   Formally speaking, let $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Setup}}\langle(\mathsf{Sen}(1^\lambda), \mathsf{Rec}(1^\lambda)\rangle$ *be an honest execution of the setup protocol. There exists an extractor* $\mathcal{E}$ *such that, for any PPT adversary* $\mathcal{A}$, *any* $n \in \mathsf{poly}(\lambda)$, *any execution of the initialization protocol of the form* $((\cdots), (\mathsf{bag}_0)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Init}}\langle\mathcal{A}(\mathsf{pp}, \mathsf{skey}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\rangle$, *and any sequence of* $2n$ *protocol executions* $\left(((\cdots), (\mathsf{bag}_i)) \leftarrow \pi_{\mathsf{Update}_i}\langle\mathcal{A}(\mathsf{pp}, \mathsf{skey}, \mathsf{com}_i \cdots), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}_{i-1}, \mathsf{com}_i)\rangle\right)_{i \in [2n]}$ *where each* com_i *is the result of invoking*

   $$((\mathsf{com}_i, \mathsf{op}_i), (\mathsf{com}_i)) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}\left\langle\mathcal{A}(\mathsf{pp}, \mathsf{skey}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\right\rangle,$$

   and where for each $i \in [2n]$, $\mathsf{Update}_i \in \{\mathsf{Insert}, \mathsf{Remove}\}$, *if it holds that,*

   $$((1), (1)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}\langle\mathcal{A}(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}_{2n}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}_{2n})\rangle$$

then $(\mathsf{val}_1, \ldots, \mathsf{val}_{2n}) \leftarrow \mathcal{E}^{\mathcal{O}(\mathcal{A})}(\mathsf{pp})$*, such that if* $\mathsf{Index}_{\mathsf{Insert}}$ *and* $\mathsf{Index}_{\mathsf{Remove}}$ *denote the values of* i *corresponding to insertions and removals, then*

$$\Pr\left[\exists \text{ a bijection } f : \mathsf{Index}_{\mathsf{Insert}} \to \mathsf{Index}_{\mathsf{Remove}}, \text{ s.t., } \forall i \in \mathsf{Index}_{\mathsf{Insert}}, (f(i) > i) \wedge \left(\mathsf{val}_i = \mathsf{val}_{f(i)}\right)\right] \geq 1 - \mathsf{neg}(\lambda)$$

and for all $i \in [2n]$*, any honest receiver* Rec*, and computationally bounded adversary* $\mathcal{A}$*, and any* $\mathsf{val}_i \neq \mathsf{val}'_i \neq \bot$*, it holds that*

$$\Pr[((\cdot), (\mathsf{val}'_i)) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Open}}\langle \mathcal{A}(\mathsf{pp}, \mathsf{skey}, \mathsf{com}, \cdot), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{com})\rangle] \leq \mathsf{neg}(\lambda)$$

3. **Zero-Knowledge:** *Zero-knowledge says that the receiver learns nothing about the values inserted and removed, beyond the fact that the limitations from* correctness *are satisfied. We formalize this by saying that the view of the receiver in an honest interaction with the sender is computationally indistinguishable from an interaction with a simulator that does not know the values inserted or removed from the bag.*

   Formally speaking, the exists a simulator $\mathsf{Sim} = (\mathsf{Sim}_{\mathsf{Setup}}, \mathsf{Sim}_{\mathsf{Init}}, \mathsf{Sim}_{\mathsf{Insert}}, \mathsf{Sim}_{\mathsf{Remove}}, \mathsf{Sim}_{\mathsf{VerEmpty}})$*, such that for any* $n \in \mathsf{poly}(\lambda)$*, the the view of* Rec *in the following sequence of protocol executions*

$$((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Setup}}\left\langle \mathsf{Sen}(1^\lambda), \mathsf{Rec}(1^\lambda)\right\rangle$$

$$((\mathsf{bag}_0, \mathsf{state}_0), (\mathsf{bag}_0)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Init}}\left\langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\right\rangle$$

   For each $i \in [2n]$ *and arbitrary* val_i*:*

$$((\mathsf{com}_i, \mathsf{op}_i), (\mathsf{com}_i)) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}\left\langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{val}_i), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\right\rangle,$$

$$((\mathsf{bag}_i, \mathsf{state}_i), (\mathsf{bag}_i)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Update}_i}\left\langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}_{i-1}, \mathsf{state}_{i-1}, \mathsf{com}_i, \mathsf{op}_i, \mathsf{val}_i), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}_{i-1}, \mathsf{com}_i)\right\rangle,$$

   where $\mathsf{Update}_i \in \{\mathsf{Insert}, \mathsf{Remove}\}$. *And finally,*

$$((1), (1)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}\left\langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}_{2n}, \mathsf{state}_{2n}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}_{2n})\right\rangle$$

   is computationally indistinguishable from its view in the following sequence of protocol executions. For readability, we omit the state passing between the interactions, but assume that each part of the simulator and the receiver can pass arbitrary state:

$$\langle(\mathsf{Sim}_{\mathsf{Setup}}(1^\lambda) \leftrightarrow \mathsf{Rec}(1^\lambda)\rangle$$

$$\langle(\mathsf{Sim}_{\mathsf{Init}}(1^\lambda) \leftrightarrow \mathsf{Rec}(1^\lambda)\rangle$$

   For each $i \in [2n]$*:*

$$((\mathsf{com}_i, \mathsf{op}_i), (\mathsf{com}_i)) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}\left\langle \mathsf{Sim}(\mathsf{pp}, \mathsf{skey}, 0), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\right\rangle,$$

$$\langle(\mathsf{Sim}_{\mathsf{Update}_i}(1^\lambda, \mathsf{com}_i, \mathsf{op}_i) \leftrightarrow \mathsf{Rec}(1^\lambda, \mathsf{com}_i)\rangle$$

   $\mathsf{Update}_i \in \{\mathsf{Insert}, \mathsf{Remove}\}$. *And finally,*

$$\langle(\mathsf{Sim}_{\mathsf{VerEmpty}}(1^\lambda) \leftrightarrow \mathsf{Rec}(1^\lambda)\rangle$$

4.2 Realizing a ZKBag Protocol

We give a concrete implementation of ZKBag in Figure 1. At a high level the protocol is as follows: during setup, the parties run the setup algorithm of the underlying linearly homomorphic commitment scheme (if there is one) π^{LCom} (see Section 3.1). During initialization, the parties just initialize three empty sets: (1) a set of committed values that were inserted into the bag $\mathcal{I}$, (2) a set of committed values that were removed from the bag $\mathcal{R}$, and (3) some private state $\mathbb{B}$ for the sender that will hold plaintext information about the committed values. Each time a (committed) item $[\![\overrightarrow{v}]\!]$ is inserted into the bag, the receiver samples a random tag $\leftarrow\$\ \mathbb{F}$ and both parties add $([\![\mathsf{tag}]\!], [\![\overrightarrow{v}]\!])$ to the set of "input elements" $\mathcal{I}$. Additionally, the sender records the tag and values by adding $(\mathsf{tag}, \overrightarrow{v})$ to $\mathbb{B}$. Whenever the sender

$((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Setup}} \langle \mathsf{Sen}(1^\lambda), \mathsf{Rec}(1^\lambda) \rangle$

- Sen and Rec invoke $((\mathsf{pp}^{\mathsf{LCom}}, \mathsf{skey}^{\mathsf{LCom}}), (\mathsf{pp}^{\mathsf{LCom}}, \mathsf{rkey}^{\mathsf{LCom}})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Setup}} \langle \mathsf{Sen}(1^\lambda), \mathsf{Rec}(1^\lambda) \rangle$
- Output $(\mathsf{pp} = \mathsf{pp}^{\mathsf{LCom}}, \mathsf{skey} = \mathsf{skey}^{\mathsf{LCom}})$ to Sen and $(\mathsf{pp} = \mathsf{pp}^{\mathsf{LCom}}, \mathsf{rkey} = \mathsf{rkey}^{\mathsf{LCom}})$ to Rec.

$((\mathsf{bag}, \mathsf{state}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Init}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}) \rangle$

- Sen and Rec each initialize an empty list of inserted elements $\mathcal{I} \leftarrow \emptyset$, an empty list of removed elements $\mathcal{R} \leftarrow \emptyset$ and a counter $\mathsf{cnt} \leftarrow 0$. Additionally, Sen initializes a map $\mathbb{B} \leftarrow \emptyset$.
- Output $((\mathcal{I}, \mathcal{R}), \mathbb{B})$ to Sen and $(\mathcal{I}, \mathcal{R})$ to Rec.

$((\mathsf{bag}', \mathsf{state}'), (\mathsf{bag}')) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Insert}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}, \mathsf{state}, \left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right], \mathsf{op}, \mathsf{val}\right), \mathsf{Rec}\left(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}, \left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right]\right) \right\rangle$

- Rec samples $\mathsf{tag} \leftarrow\!\$\, \mathbb{F}$ and sends it to Sen.
- Sen and Rec invoke $(([\![\mathsf{tag}]\!], \cdot), ([\![\mathsf{tag}]\!])) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{tag}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{tag}) \rangle$ on shared randomness.
- They add the following tuple to the list of inserted elements: $\mathcal{I} \leftarrow \mathcal{I} \cup \left([\![\mathsf{tag}]\!] \parallel \left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right]\right)$
- Finally, Sen adds a new counter and tag for the value to the map $\mathbb{B}[\mathsf{val}].\mathsf{Push}(\mathsf{tag})$
- Output $((\mathcal{I}, \mathcal{R}), \mathbb{B})$ to Sen and $(\mathcal{I}, \mathcal{R})$ to Rec.

$((\mathsf{bag}', \mathsf{state}'), (\mathsf{bag}')) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Remove}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}, \mathsf{state}, \left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right], \mathsf{op}, \mathsf{val}\right), \mathsf{Rec}\left(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}, \left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right]\right) \right\rangle$

- Sen retrieves a tag for the value from the map as $\mathsf{tag} \leftarrow \mathbb{B}[\overrightarrow{v}].\mathsf{Pop}()$, and computes commitments to this tag $(([\![\mathsf{tag}]\!], \cdot), ([\![\mathsf{tag}]\!])) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{tag}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}) \rangle$
- Sen and Rec add to the set of removed elements $\mathcal{R} \leftarrow \mathcal{R} \cup \left([\![\mathsf{tag}]\!] \parallel \left[\!\left[\overrightarrow{\mathsf{val}}\right]\!\right]\right)$

$((b), (b)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}, \mathsf{state}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}) \rangle$

- Sen and Rec assert equality between the list of inserted and removed elements by invoking $\pi^{\mathsf{ZKMultiSet}}$ on $(\mathcal{I}, \mathcal{R})$

Figure 1: Zero-Knowledge Bag Protocol.

wants to remove an element $\overrightarrow{v}$, they recall the appropriate tag using $\mathbb{B}$, creates a fresh commitment to $(\mathsf{tag}, \overrightarrow{v})$, and then both sides add the fresh commitment to the set of "removed elements" $\mathcal{R}$. The final check is simply checking (set) equality of the inserted and removed elements using the $\pi^{\mathsf{ZKMultiSet}} := (\pi^{\mathsf{ZKMultiSet}}_{\mathsf{Setup}}, \pi^{\mathsf{ZKMultiSet}}_{\mathsf{Prove}}, \pi^{\mathsf{ZKMultiSet}}_{\mathsf{Verify}})$ protocol (see Section 3.5).

Finally, the intuition for why this simple interactive protocol achieves soundness, i.e., ensures that the sender cannot cheat by removing an element that was not previously inserted is the following: in order to do so, the sender would need to guess the appropriate tag that will be sampled in the future, which they are only able to do with negligible probability $1/|\mathbb{F}|$. Therefore, they are restricted to "recalling/retrieving" a previously inserted element, for which the tag is known. They are prevented from removing the element multiple times because the tags for each insertion should be unique (with high probability). $\pi^{\mathsf{ZKMultiSet}}$ ensures that insertion and removals are one-to-one. Formally, we prove the following theorem:

Theorem 4.1. *Assuming that π^{LCom} in a secure linearly homomorphic commitment scheme (see Section 3.1), and $\pi^{\mathsf{ZKMultiSet}}$ is a commit-and-prove style multi-set equality proof system(see Section 3.5), then π^{ZKBag}, shown in Figure 1, is a LinCom-Based Zero-Knowledge Bag, as defined in Definition 4.*

Correctness. By the correctness of $\Pi_{\mathsf{MultiSetEquality}}$, it is simple to see that Π_{ZKBag} is correct. Namely, if the pattern of insertions and removals is honest, ie. the insertions and removals are a permutation and each removal comes *after* its associated insertion, then Π_{ZKBag} will output 1 with high probability.

Knowledge Soundness. The extractor $\mathcal{E}$ runs by simply running the extractor of the linearly-homomorphic commitment scheme on each of $\{\mathsf{com}_i\}_{i\in[2n]}$. Denote the outputs of these extractors as $\mathsf{val}_1, \ldots, \mathsf{val}_{2n}$. Moreover, if Update_i is Remove, $\mathcal{E}$ runs the extractor of the linearly-homomorphic commitment scheme on the commitment to the tag created in that interaction. Denote the outputs of the extractors as $\mathsf{tag}_i^{\mathsf{Remove}}$. If any of these extractions fails, the extractors fails with error $\mathsf{Error}_{\mathsf{ComExtract}}$. Otherwise, $\mathcal{E}$ outputs $\mathsf{val}_1, \ldots, \mathsf{val}_{2n}$.

We now show that $\mathcal{E}$ will output a compliant set of values $\mathsf{val}_1, \ldots, \mathsf{val}_{2n}$ with high probability. Let NumInsert denote the number of insertions and NumRemove denote the number of insertions and removals in the interaction, respectively.

1. Note that the extractor only outputs $\mathsf{Error}_{\mathsf{ComExtract}}$ with $3n$ times the error rate of the extractor of the linearly-homomorphic commitment scheme, which, by the binding/extraction property of the linearly-homomorphic commitment scheme only happens with negligible probability.

2. Next, note that the probability of any two instances of Insert in the interaction sharing a value tag is $< \frac{n^2}{|\mathbb{F}|}$. Since this value might not be small enough, in Section 8 we discuss an optimization to our implementation, where we sample each tag uniformly at random from $\mathbb{F}^2$ instead of $\mathbb{F}$. As a result, the probability of any two instances sharing a tag is $< \frac{n^2}{|\mathbb{F}|^2}$. Looking ahead, for the remaining soundness analysis we will assume that each tag is samped as two field elements.

3. To fix notation, we create the following tuples for $i \in [2n]$:
   - If Update_i is Insert, then create the tuple $(i, \mathsf{tag}_i^{\mathsf{Insert}}, \mathsf{val}_i)$, where $\mathsf{tag}_i^{\mathsf{Insert}}$ is the tag generated during the execution of Update_i. Denote the set of all such tuples as $\{(\mathsf{timestamp}_j^{\mathsf{Insert}}, \mathsf{tag}_j^{\mathsf{Insert}}, \mathsf{val}_j^{\mathsf{Insert}})\}_{j\in[\mathsf{NumInsert}]}$
   - If Update_i is Remove, then create the tuple $(i, \mathsf{tag}_i^{\mathsf{Remove}}, \mathsf{val}_i)$, where $\mathsf{tag}_i^{\mathsf{Remove}}$ is the tag extracted above. Denote the set of all such tuples as $\{(\mathsf{timestamp}_j^{\mathsf{Remove}}, \mathsf{tag}_j^{\mathsf{Remove}}, \mathsf{val}_j^{\mathsf{Remove}})\}_{j\in[\mathsf{NumRemove}]}$

4. Next, note that by the soundness of the permutation check, $\mathsf{NumInsert} = \mathsf{NumRemove} = n$, and $\{(\mathsf{tag}_j^{\mathsf{Insert}}, \mathsf{val}_j^{\mathsf{Insert}})\}_{j\in[\mathsf{NumInsert}]}$ and $\{(\mathsf{tag}_j^{\mathsf{Remove}}, \mathsf{val}_j^{\mathsf{Remove}})\}_{j\in[\mathsf{NumRemove}]}$ are permutations of one another, except with negligible probability bounded by $\frac{n}{|\mathbb{F}|}$. Denote this permutation as f.

5. Next, we observe that for each $(\mathsf{timestamp}_j^{\mathsf{Remove}}, \mathsf{tag}_j^{\mathsf{Remove}}, \mathsf{val}_j^{\mathsf{Remove}})$, there exists a $(\mathsf{timestamp}_{j'}^{\mathsf{Insert}}, \mathsf{tag}_{j'}^{\mathsf{Insert}}, \mathsf{val}_{j'}^{\mathsf{Insert}})$ with $\mathsf{timestamp}_{j'}^{\mathsf{Insert}} < \mathsf{timestamp}_j^{\mathsf{Remove}}$ such that $\mathsf{tag}_{j'}^{\mathsf{Insert}} = \mathsf{tag}_j^{\mathsf{Remove}}$, and $\mathsf{val}_{j'}^{\mathsf{Insert}} = \mathsf{val}_j^{\mathsf{Remove}}$. If this were not the case, then it would imply that the tag for the insertion must have been sampled after the removal and the prover must have correctly guessed a tag before it was sampled. Clearly the probability that there exist j, j', such that this happens is at most $\frac{n^2}{|\mathbb{F}|^2}$.

We let the bijective map f be defined by a valid permutation between inserts and removals, which must exist with high probability, as described in (4). Note that this f is monotonically increasing by (5). Moreover, because we invoked the linearly homomorphic commitment scheme's extractor, for all computationally bounded adversaries $\mathcal{A}$ there is only a negligible probability that they could produce a valid equivocation to the commitments. Thus, with statistically small probability $\leq \frac{n^2}{|\mathbb{F}|^2} + \frac{n}{|\mathbb{F}|} \leq \frac{2n}{|\mathbb{F}|}$, the output of $\mathcal{E}$ is compliant with the definition.

Zero-knowledge. The simulator Sim simply follows the protocol executions described in Definition 4, and honestly follows the protocol at all steps. Note that the most significant difference is that the simulator commits to zero instead of other values, but otherwise the interactions are identical.

We now show that view of the receiver when interacting with the simulator is the computationally close to the view of the receiver interacting with the honest sender. We proceed with a hybrid argument. Let Hybrid_0 denote the interaction between the receiver and the honest sender.

- Hybrid_1 : Let Hybrid_1 be the same as Hybrid_0, but Sim simulates $\pi^{\mathsf{ZKMultiSet}}$ during $\pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}$. By the zero-knowledge property of $\pi^{\mathsf{ZKMultiSet}}$, the view of receiver in Hybrid_1 and Hybrid_0 are computationally close.
- $\mathsf{Hybrid}_2, \mathsf{Hybrid}_3, \ldots, \mathsf{Hybrid}_{2n+1}$: In each of these hybrids, instead of commiting to a real value, Sim commits to 0 instead. By the hiding property of the commitment scheme, the view of receiver in Hybrid_{i+1} and Hybrid_i are computationally close for $i \in [1, 2n+1]$.
- Hybrid_{2n+2} : Hybrid_{2n+2} is the same as Hybrid_{2n+1}, but Sim executes $\Pi_{\mathsf{MultiSetEquality}}$ honestly instead of simulating. Again, by the zero-knowledge property of $\Pi_{\mathsf{MultiSetEquality}}$, the view of receiver in Hybrid_{2n+1} and Hybrid_{2n+2} are computationally close.

Note that the view of the receiver in Hybrid_{2n+2} is distributed the same as the view of the receiver when interacting with the simulator above. Thus, we have concluded our proof.

5 Verifying Memory Consistency using ZKBag

When proving the correct execution of a RAM program, we need to ensure that each time an address is read from memory, only the value *last written* to that address must be returned. Importantly, because we require zero-knowledge, this must be done without revealing executed programs memory access patterns. We observe that this aligns perfectly with the properties guaranteed by ZKBag.

Recall that memory can be seen as a sequence of tuples $(\mathsf{addr}, \mathsf{val})$, where addr is a unique address within the memory space and val is the current value being stored at that address. We can use ZKBag as a *key-value store* by dedicating the first part of the inserted value to be the key and the second part to be the value. That is, we store tuples of the form $(\mathsf{addr}, \mathsf{val})$ within the bag. The state of the bag corresponds to the "current" state of memory. Updating the contents of memory can be handled by updating (i.e., inserting or removing) the contents of the ZKBag.

Rather than giving a formal definition for our protocol for handling memory π^{Memory}, we simply observe that the definitions are functionally equivalent to those of ZKBag, but the elements being inserted and removed from the bag now contain memory addresses. In order to make the semantics of our final construction easier to read, we provide a wrapper around the ZKBag with the names of common memory operations: $\mathsf{Init}, \mathsf{Read}, \mathsf{Update}, \mathsf{Verify}$:

- $((\mathsf{state}_\mathsf{P}, \mathsf{state}_\mathsf{V}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{Memory}}_{\mathsf{Init}} \langle \mathsf{P}(\{\mathsf{val}_{\mathsf{addr}}\}_{\mathsf{addr} \in 1\ldots,m}), \mathsf{V}(\{\mathsf{val}_{\mathsf{addr}}\}_{\mathsf{addr} \in 1\ldots,m})\rangle$: The prover and verifier take in a set of public values that will make up the initial contents of memory. The result will be state for each party.
- $((\mathsf{bag}, \mathsf{state}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{Memory}}_{\mathsf{Read}} \langle \mathsf{P}(\mathsf{state}_\mathsf{P}, (\llbracket \mathsf{addr} \rrbracket, \llbracket \mathsf{val} \rrbracket), (\mathsf{op}_{\mathsf{addr}}, \mathsf{op}_{\mathsf{val}}), (\mathsf{addr}, \mathsf{val})), \mathsf{V}(\mathsf{state}_\mathsf{V}, (\llbracket \mathsf{addr} \rrbracket, \llbracket \mathsf{val} \rrbracket))\rangle$: The prover and verifier take a commitment $(\llbracket \mathsf{addr} \rrbracket, \llbracket \mathsf{val} \rrbracket)$—where val is the value that the prover *claims* will result of reading from the address. Additionally, the prover takes in the actual value and opening to the commitments. The result is updated state for each party.
- $((\mathsf{bag}, \mathsf{state}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{Memory}}_{\mathsf{Update}} \langle \mathsf{P}(\mathsf{state}_\mathsf{P}, (\llbracket \mathsf{addr} \rrbracket, \llbracket \mathsf{val} \rrbracket, \llbracket \mathsf{val}' \rrbracket), (\mathsf{op}_{\mathsf{addr}}, \mathsf{op}_{\mathsf{val}}, \mathsf{op}_{\mathsf{val}'}), (\mathsf{addr}, \mathsf{val}, \mathsf{val}')), \mathsf{V}(\mathsf{state}_\mathsf{V}, (\llbracket \mathsf{addr} \rrbracket, \llbracket \mathsf{val} \rrbracket, \llbracket \mathsf{val}' \rrbracket))\rangle$: The prover and verifier take a commitment $(\llbracket \mathsf{addr} \rrbracket, \llbracket \mathsf{val} \rrbracket)$ along with a commitment to the new value $\llbracket \mathsf{val}' \rrbracket$. Additionally, the prover takes in the actual value and opening to the commitments. The result is an updated state for each party.
- $((b), (b)) \leftarrow \pi^{\mathsf{Memory}}_{\mathsf{Verify}} \langle \mathsf{P}(\mathsf{state}_\mathsf{P}, \{\mathsf{val}_{\mathsf{addr}}\}_{\mathsf{addr} \in 1\ldots,m}), \mathsf{V}(\mathsf{state}_\mathsf{V}, \{\mathsf{val}_{\mathsf{addr}}\}_{\mathsf{addr} \in 1\ldots,m})\rangle$: The prover and verifier take in their current state and a set of values (representing the current state of memory) and then output 1 if this is really the current state of memory and 0 otherwise. Optionally, the verifier can take any amount of these values in committed form (to maintain secrecy).

We provide a writeup of the memory checking protocol π^{Memory} in Figure 2. In brief, during $\pi^{\mathsf{Memory}}_{\mathsf{Init}}$, the parties initialize and setup the ZKBag, and then insert tuples with the address and values to the ZKBag. When invoking $\pi^{\mathsf{Memory}}_{\mathsf{Update}}$, the parties remove the old address-value tuple $(\llbracket \mathsf{addr} \rrbracket, \llbracket \mathsf{val} \rrbracket)$ from the ZKBag and insert the new tuple

$((\mathsf{state_P}, \mathsf{state_V}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{Memory}}_{\mathsf{Init}} \langle \mathsf{P}(\{\mathsf{val_{addr}}\}_{\mathsf{addr} \in 1\ldots,m}), \mathsf{V}(\{\mathsf{val_{addr}}\}_{\mathsf{addr} \in 1\ldots,m})\rangle$

- P and V initialize and setup a ZKBag by invoking both $\pi^{\mathsf{ZKBag}}_{\mathsf{Setup}}$ and $\pi^{\mathsf{ZKBag}}_{\mathsf{Init}}$
- For each $\mathsf{addr} \in 1 \ldots, m$:
  - P and V generate $[\![\mathsf{addr}]\!]$ and $[\![\mathsf{val_{addr}}]\!]$ by invoking $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ on shared randomness (to generate a public commitment).
  - P and V insert the tuple $([\![\mathsf{addr}]\!], [\![\mathsf{val_{addr}}]\!])$ into the ZKBag by invoking $\pi^{\mathsf{ZKBag}}_{\mathsf{Insert}}$

$((\mathsf{bag}, \mathsf{state}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{Memory}}_{\mathsf{Read}} \langle \mathsf{P}(\mathsf{state_P}, ([\![\mathsf{addr}]\!], [\![\mathsf{val}]\!]), (\mathsf{op_{addr}}, \mathsf{op_{val}}), (\mathsf{addr}, \mathsf{val})), \mathsf{V}(\mathsf{state_V}, ([\![\mathsf{addr}]\!], [\![\mathsf{val}]\!]))\rangle$

- P and V remove $([\![\mathsf{addr}]\!], [\![\mathsf{val}]\!])$ from the ZKBag by invoking $\pi^{\mathsf{ZKBag}}_{\mathsf{Remove}}$
- P and V insert $([\![\mathsf{addr}]\!], [\![\mathsf{val}]\!])$ into the ZKBag by invoking $\pi^{\mathsf{ZKBag}}_{\mathsf{Insert}}$

$((\mathsf{bag}, \mathsf{state}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{Memory}}_{\mathsf{Update}} \langle \mathsf{P}(\mathsf{state_P}, ([\![\mathsf{addr}]\!], [\![\mathsf{val}]\!], [\![\mathsf{val'}]\!]), (\mathsf{op_{addr}}, \mathsf{op_{val}}, \mathsf{op_{val'}}), (\mathsf{addr}, \mathsf{val}, \mathsf{val'})), \mathsf{V}(\mathsf{state_V}, ([\![\mathsf{addr}]\!], [\![\mathsf{val}]\!], [\![\mathsf{val'}]\!]))\rangle$

- P and V remove $([\![\mathsf{addr}]\!], [\![\mathsf{val}]\!])$ from the ZKBag by invoking $\pi^{\mathsf{ZKBag}}_{\mathsf{Remove}}$
- P and V insert $([\![\mathsf{addr}]\!], [\![\mathsf{val'}]\!])$ into the ZKBag by invoking $\pi^{\mathsf{ZKBag}}_{\mathsf{Insert}}$

$((b), (b)) \leftarrow \pi^{\mathsf{Memory}}_{\mathsf{Verify}} \langle \mathsf{P}(\mathsf{state_P}, \{\mathsf{val_{addr}}\}_{\mathsf{addr} \in 1\ldots,m}), \mathsf{V}(\mathsf{state_V}, \{\mathsf{val_{addr}}\}_{\mathsf{addr} \in 1\ldots,m})\rangle$

- For each $\mathsf{addr} \in 1 \ldots, m$:
  - P and V generate $[\![\mathsf{addr}]\!]$ and $[\![\mathsf{val_{addr}}]\!]$ by invoking $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ on shared randomness.
  - P and V remove the tuple $([\![\mathsf{addr}]\!], [\![\mathsf{val_{addr}}]\!])$ from the ZKBag by invoking $\pi^{\mathsf{ZKBag}}_{\mathsf{Remove}}$.
- Finally, P and V check that ZKBag is empty by invoking $\pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}$

Figure 2: A protocol for verifying memory consistency using ZKbag.

$([\![\mathsf{addr}]\!], [\![\mathsf{val'}]\!])$ into the ZKBag. Importantly, the commitment to the address $[\![\mathsf{addr}]\!]$ is consistent across the two protocol invocations. When invoking $\pi^{\mathsf{Memory}}_{\mathsf{Read}}$ the parties remove the address-value tuple $([\![\mathsf{addr}]\!], [\![\mathsf{val}]\!])$ and the reinsert the same tuple back into the ZKBag. Finally, when invoking $\pi^{\mathsf{Memory}}_{\mathsf{Verify}}$, the parties remove the remaining contents of the ZKBag and then call $\pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}$.

6 Verifying Processor Execution using ZKBag

When proving correct execution of a RAM program, the prover must convince the verifier that a "valid" instruction was executed at every step of the program. This constitutes: (1) the prover picked one of the instructions supported by the processor, (2) the picked instruction was executed honestly and (3) that the picked instruction is the "correct choice" based on the input-dependent execution. In this section, we present a zero-knowledge protocol using ZKBag (see Section 4) that helps enforce (1) and (2). In the next section, we demonstrate how to combine this protocol with the protocol for memory consistency (see Section 5) to obtain a zero-knowledge proof system for RAM programs that enforces all of the above guarantees.

Disjunctive Relation. Our zero-knowledge protocol for checking correct execution of processor instructions, is a custom LinCom based commit-and-prove style zero-knowledge protocol see Section 3.3) for the following relation $\mathcal{R}^{\mathsf{ZKDisj}}$:

> *Let $(\mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i)_{i\in[\ell]}$ be a set of ℓ R1CS instances[a]. Given t commitments $(\llbracket \overrightarrow{\mathsf{z}_j} \rrbracket)_{j\in[t]}$ computed using $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ (see Section 3.1), the prover/sender wants to convince the receiver/verifier that for each $j \in [t]$, it knows $\overrightarrow{\mathsf{z}_j}, \overrightarrow{\mathsf{op}_j}$ such that they form a valid opening for $\llbracket \mathsf{z}_j \rrbracket$ and an index $\mathsf{inst}_j \in [\ell]$, such that $\overrightarrow{\mathsf{z}_j}$ is a valid extended witness for $(\mathbf{A}_{\mathsf{inst}_j}, \mathbf{B}_{\mathsf{inst}_j}, \mathbf{C}_{\mathsf{inst}_j})$.*
>
> [a] We assume that each of these R1CS instances is of the same size. This can be achieved without loss of generality by appropriately padding the smaller instances.

Recall from Section 3.2, that for an R1CS relation, each extended witness is of the form $\overrightarrow{\mathsf{z}_j} = \overrightarrow{\mathsf{w}_j} \| \overrightarrow{\mathsf{x}_j} \| 1$, where $\overrightarrow{\mathsf{x}_j}$ is a part of the instance (which may or may not be known to the verifier), while $\overrightarrow{\mathsf{w}_j}$ is exclusively known only to the prover. Therefore, $\llbracket \overrightarrow{\mathsf{z}_j} \rrbracket$ can be parsed as $\llbracket \overrightarrow{\mathsf{w}_j} \rrbracket \| \llbracket \overrightarrow{\mathsf{x}_j} \rrbracket \| \llbracket 1 \rrbracket$. Here, we assume that $\llbracket \overrightarrow{\mathsf{w}_j} \rrbracket$ were computed traditionally, commitment $\llbracket 1 \rrbracket$ was computed using shared randomness, and the randomness used for $\llbracket \overrightarrow{\mathsf{x}_j} \rrbracket$ is either private or shared depending on the nature of $\overrightarrow{\mathsf{x}_j}$.

Commit-and-Prove ZK Proof System for $\mathcal{R}^{\mathsf{ZKDisj}}$. As discussed previously, we design a commit-and-prove zero-knowledge proof system for $\mathcal{R}^{\mathsf{ZKDisj}}$ using a ZKBag protocol π^{ZKBag} (see Section 4) and the folding scheme for relaxed R1CS from [KST22]. Given these tools, our protocol is straightforward and works as follows:

Setup: $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{ZKDisj}}_{\mathsf{Setup}}$

The parties run $\pi^{\mathsf{LCom}}_{\mathsf{Setup}}$ to obtain $(\mathsf{pp}, \mathsf{skey}, \mathsf{rkey})$.

Prove: $((\mathsf{Proof}^{\mathsf{ZK}}, \mathsf{st}), (\mathsf{Proof}^{\mathsf{ZK}})) \leftarrow \pi^{\mathsf{ZKDisj}}_{\mathsf{Prove}}$

This protocol proceeds in two phases:

I. Initialization Phase: The parties start by invoking $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ to create public commitments to trivially satisfied relaxed R1CS extended witnesses (i.e., just a vector of 0s) for each of the ℓ instructions. They then invoke $\pi^{\mathsf{ZKBag}}_{\mathsf{Init}}$ to initialize a ZKBag and $\pi^{\mathsf{ZKBag}}_{\mathsf{Insert}}$ to store each of these commitments in the ZKbag (see Figure 1).We refer to these commitments as accumulators.

II. Execution Phase: Then for each step $j \in [t]$:

i) Parties invoke $\pi^{\mathsf{ZKBag}}_{\mathsf{Remove}}$ to retrieve the accumulator for the satisfied instruction inst_j from the ZKbag.

ii) The prover computes cross terms $\overrightarrow{T}$ for the $\mathsf{inst}_j^{\mathrm{th}}$ instruction using (see Section 3.2) the retrieved accumulator and the new satisfied R1CS extended witness $\overrightarrow{\mathsf{z}_j}$ and uses $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ to compute a commitments to these cross terms.

iii) The verifier samples a random field element $r \leftarrow\$ \mathbb{F}$.

iv) The parties fold the retrieved accumulator onto the new satisfied R1CS extended witness $\overrightarrow{\mathsf{z}_j}$ using r. This forms the updated accumulator for the $\mathsf{inst}_j^{\mathrm{th}}$ instruction.

v) The parties invoke $\pi^{\mathsf{ZKBag}}_{\mathsf{Insert}}$ to store the updated accumulator.

Verify: $((b), (b)) \leftarrow \pi^{\mathsf{ZKDisj}}_{\mathsf{Verify}}$

Each accumulator is removed, randomized, and checked:

- For each $i \in [\ell]$, the prover samples a random relaxed R1CS instance and computes the corresponding error term. The prover recalls the accumulator for the i^{th} instruction and computes cross terms $\overrightarrow{T}$ for this instruction using the retrieved accumulator and the random R1CS extended witness and uses $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ to compute a commitments to these cross terms.

- The verifier samples a random field element $r \leftarrow\$ \mathbb{F}$.
- For each $i \in [\ell]$, the parties use $\pi^{\mathsf{ZKBag}}_{\mathsf{Remove}}$ to retrieve the accumulator for the i^{th} instruction and fold it onto the commitment of the randomly sampled R1CS instance for this instruction.
- The parties verify that the bag is empty with $\pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}$.
- Finally, for each $i \in [\ell]$, the prover opens the final accumulators and the verifier check that they are satisfying.

The parties start by creating public commitments to trivially satisfied relaxed R1CS extended witnesses (i.e., just a vector of 0s) for each of the ℓ instructions. They then initialize a ZKBag and store each of these commitments in the ZKbag (see Figure 1). We refer to these commitments as accumulators for the ℓ instructions.Then for each step $j \in [t]$ of the processor, the parties proceed as follows: i) Parties retrieve the accumulator for the satisfied instruction inst_j from the ZKbag. ii) The prover computes cross terms $\overrightarrow{T}$ for the $\mathsf{inst}_j^{\text{th}}$ instruction using the retrieved accumulator and the new satisfied R1CS extended witness $\overrightarrow{\mathsf{z}_j}$ and computes a commitment to these cross terms. iii) The verifier samples a random field element. iv) The parties fold the retrieved accumulator onto the new satisfied R1CS extended witness $\overrightarrow{\mathsf{z}_j}$ using this random value. This forms the updated accumulator for the $\mathsf{inst}_j^{\text{th}}$ instruction. v) Store the updated accumulator in the bag. At the end, every accumulator is extracted from the bag, randomized and checked naively.

We note that a naïve strategy to design a commit-and-prove protocol for this relation without zero-knowledge would be to simply commit to the extended witness $\overrightarrow{\mathsf{z}_j}$ at each step, reveal the associated instruction index inst_j use any generic commit-and-prove proof system (e.g. QuickSilver [YSWW21]) to prove correct execution of this step. Our protocol achieves the *zero-knowledge* property while only incurring a multiplicative overhead of 4 of this naïve protocol. This is because our protocol requires committing to 4 vectors proportional to the length of $\overrightarrow{\mathsf{z}_j}$ and the ZKBag operations are independent of the dimension of the extended witness or R1CS relation.

We include a formal description of this protocol in Figures 3 and 4. We now prove the following theorem:

Theorem 6.1. *Assuming that π^{LCom} in a secure linearly homomorphic commitment scheme (see Section 3.1), and π^{ZKBag} is a zero-knowledge bag (see Section 4), then π^{ZKDisj}, shown in Figures 3 and 4, is a LinCom-based commit-and-prove zero-knowledge as defined in Section 3.3 for $\mathcal{R}^{\mathsf{ZKDisj}}$.*

Proof. **Correctness.** Correctness follows from correctness of Linearly Homomorphic commitment, ZKBag and the folding property of relaxed R1CS (see Section 3.2).

Zero-Knowledge. Let $\mathsf{Sim}^{\mathsf{ZKBag}} = (\mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{Setup}}, \mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{Init}}, \mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{Insert}}, \mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{Remove}}, \mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}})$ be the simulator for ZKBag. We now describe the simulator Sim for our π^{ZKDisj} protocol.

1. *Setup:* Sim uses $\mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{Setup}}$ to simulate the setup protocol.

2. *Initialization Phase:* Sim uses $\mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{Init}}$ to simulate initializing a ZKBag. For each $i \in [\ell]$, it then honestly invokes $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ to compute public commitments to trivially satisfied relaxed-R1CS instances and uses $\mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{Insert}}$ to simulate inserting thses commitments in the simulated ZKBag. Sim also initializes the map $\mathbb{M}$ as described in the protocol.

3. *Execution Phase:* For each $j \in [t]$, Sim proceeds as follows: Set $\overrightarrow{\mathsf{w}_j} = \overrightarrow{0}$. Additioanlly, if $\overrightarrow{\mathsf{x}_j}$ is unknown to the receiver, set $\overrightarrow{\mathsf{x}_j} = \overrightarrow{0}$. Invoke $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ to compute a commitment to $\overrightarrow{\mathsf{z}_j} = \overrightarrow{\mathsf{w}_j} \| \overrightarrow{\mathsf{x}_j} \| 1$. Set $\overrightarrow{\mathsf{z}'} = \overrightarrow{\mathsf{e}'} = \overrightarrow{T} = \overrightarrow{0}$. It honestly invokes $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ to compute commitments to these values. Use $\mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{Remove}}$ to simulate removing $\left[\!\left[\overrightarrow{\mathsf{z}'}\right]\!\right]$ and $\left[\!\left[\overrightarrow{\mathsf{e}'}\right]\!\right]$ from the simulated bag. Finally, it samples $r \leftarrow\$ \mathbb{F}$, computes $[\![\overrightarrow{\mathsf{z}}]\!]$ and $[\![\overrightarrow{\mathsf{e}}]\!]$ using r and the above commitments as described in the protocol using $\pi^{\mathsf{LCom}}_{\mathsf{Comb}}$. Finally, it uses $\mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{Insert}}$ to simulate inserting $[\![\overrightarrow{\mathsf{z}}]\!]$ and $[\![\overrightarrow{\mathsf{e}}]\!]$ in the simulated ZKBag.

4. *Verification Protocol:* For each $i \in [\ell]$, the simulator sets $\overrightarrow{T_i} = \overrightarrow{\mathsf{z}_{(i,1)}} = \overrightarrow{\mathsf{z}_{(i,2)}} = \overrightarrow{\mathsf{e}_{(i,1)}} = \overrightarrow{\mathsf{e}_{(i,2)}} = \overrightarrow{0}$ and invokes $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ to compute commitments to these values. For each $i \in [\ell]$, it then samples $r \leftarrow\$ \mathbb{F}$ and uses $\mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{Remove}}$ to simulate removing $[\![\overrightarrow{\mathsf{z}_i}]\!]$ and $[\![\overrightarrow{\mathsf{e}_i}]\!]$ from the simulated ZKBag. Uses the above commitments along with r to compute $[\![\overrightarrow{\mathsf{z}_i}]\!]$ and $[\![\overrightarrow{\mathsf{e}_i}]\!]$ as described in the protocol using $\pi^{\mathsf{LCom}}_{\mathsf{Comb}}$. Then use $\mathsf{Sim}^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}$ to simulate demonstrating

that the ZKBag is empty. Finally, for each $i \in [\ell]$, it samples $\overrightarrow{\mathsf{z}_i}, \overrightarrow{\mathsf{e}_i}$ such that $\mathbf{A}_i \cdot \overrightarrow{\mathsf{z}_i} \circ \mathbf{B}_i \cdot \overrightarrow{\mathsf{z}_i} =^? = \mathsf{u}_i \cdot (\mathbf{C} \cdot \overrightarrow{\mathsf{z}_i}) + \mathsf{e}_i$. It uses these values and runs $\mathsf{Equiv}^{\mathsf{LCom}}$ to compute an equivocal opening for $\overrightarrow{\mathsf{z}_i}, \overrightarrow{\mathsf{e}_i}$ and invokes $\pi_{\mathsf{Open}}^{\mathsf{LCom}}$ using these openings.

$((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{ZKDisj}}_{\mathsf{Setup}} \left\langle \mathsf{Sen}\left(1^\lambda\right), \mathsf{Rec}\left(1^\lambda\right)\right\rangle$

- Sen and Rec invoke $\left(\left(\mathsf{pp}^{\mathsf{LCom}}, \mathsf{skey}^{\mathsf{LCom}}\right), \left(\mathsf{pp}^{\mathsf{LCom}}, \mathsf{rkey}^{\mathsf{LCom}}\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Setup}} \left\langle \mathsf{Sen}\left(1^\lambda\right), \mathsf{Rec}\left(1^\lambda\right)\right\rangle$
- Output $\left(\mathsf{pp} = \mathsf{pp}^{\mathsf{LCom}}, \mathsf{skey} = \mathsf{skey}^{\mathsf{LCom}}\right)$ to Sen and $\left(\mathsf{pp} = \mathsf{pp}^{\mathsf{LCom}}, \mathsf{rkey} = \mathsf{rkey}^{\mathsf{LCom}}\right)$ to Rec.

$\left(\left(\mathsf{Proof}^{\mathsf{ZK}}, \mathsf{st}\right), \left(\mathsf{Proof}^{\mathsf{ZK}}\right)\right) \leftarrow \pi^{\mathsf{ZKDisj}}_{\mathsf{Prove}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, (\mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i)_{i\in[\ell]}, \left(\llbracket \overrightarrow{\mathsf{z}_j} \rrbracket\right)_{j\in[t]}, \left(\overrightarrow{\mathsf{op}_{\mathsf{z}_j}}\right)_{j\in[t]}, \left(\overrightarrow{\mathsf{z}_j}\right)_{j\in[t]}\right), \mathsf{Rec}\left(\mathsf{pp}, \mathsf{rkey}, (\mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i)_{i\in[\ell]}, \left(\llbracket \overrightarrow{\mathsf{z}_j} \rrbracket\right)_{j\in[t]}\right)\right\rangle$

1. **Initialization Phase:**
   - Sen and Rec initialize a ZKBag $((\mathsf{bag}_0, \mathsf{state}_0), (\mathsf{bag}_0)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Init}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\rangle$.
   - For each $i \in [\ell]$, Sen and Rec invoke $\left(\left(\llbracket \overrightarrow{\mathsf{z}^0_i} \rrbracket, \mathsf{op}_{\mathsf{z}^0_i}\right), \left(\llbracket \overrightarrow{\mathsf{z}^0_i} \rrbracket\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \overrightarrow{0}\right), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\right\rangle$ and $\left(\left(\llbracket \overrightarrow{\mathsf{e}^0_i} \rrbracket, \mathsf{op}_{\mathsf{e}^0_i}\right), \left(\llbracket \overrightarrow{\mathsf{e}^0_i} \rrbracket\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \overrightarrow{0}\right), \mathsf{Rec}\left(\mathsf{pp}, \mathsf{rkey}, \overrightarrow{0}\right)\right\rangle$ on shared randomness, to compute public commitments to a trivially satisfied relaxed-R1CS instance and stores them in the ZKBag

$$((\mathsf{bag}_i, \mathsf{state}_i), (\mathsf{bag}_i)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Insert}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}_{i-1}, \mathsf{state}_i, \left(\llbracket \overrightarrow{\mathsf{z}^0_i} \rrbracket, \llbracket \overrightarrow{\mathsf{e}^0_i} \rrbracket\right), \left(\mathsf{op}_{\mathsf{z}^0_i}, \mathsf{op}_{\mathsf{e}^0_i}\right), \left(\overrightarrow{\mathsf{z}^0_i}, \overrightarrow{\mathsf{e}^0_i}\right)\right), \mathsf{Rec}\left(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}_{i-1}, \left(\llbracket \overrightarrow{\mathsf{z}^0_i} \rrbracket, \llbracket \overrightarrow{\mathsf{e}^0_i} \rrbracket\right)\right)\right\rangle$$

   - Sen initializes a local map $\mathbb{M}$ maintaining the state of each of the ℓ accumulators: $\forall i \in [\ell],\ \mathbb{M}[i] \leftarrow (\overrightarrow{\mathsf{z}_i}, \overrightarrow{\mathsf{e}_i})$.
2. **Execution Phase:** For each $j \in [t]$,
   - Given as input an index $\mathsf{inst}_j \in [\ell]$, Sen retrieves the state of the inst_j'th accumulator $\left(\overrightarrow{\mathsf{z}}', \overrightarrow{\mathsf{e}}'\right) \leftarrow \mathbb{M}[\mathsf{inst}_j]$, computes the cross terms

$$\overrightarrow{T} = \mathbf{A} \cdot \overrightarrow{\mathsf{z}}' \circ \mathbf{B} \cdot \overrightarrow{\mathsf{z}_j} + \mathbf{A} \cdot \overrightarrow{\mathsf{z}_j} \circ \mathbf{B} \cdot \overrightarrow{\mathsf{z}}' - \mathsf{u}_1 \cdot \mathbf{C} \cdot \overrightarrow{\mathsf{z}_j} - \mathsf{u}_2 \cdot \mathbf{C} \cdot \overrightarrow{\mathsf{z}}'$$

   - Sen and Rec invoke the following to compute commitments to the retrieved accumulator and these cross terms

$$\left(\left(\llbracket \overrightarrow{\mathsf{z}'} \rrbracket, \mathsf{op}_{\mathsf{z}'}\right), \left(\llbracket \overrightarrow{\mathsf{z}'} \rrbracket\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \overrightarrow{\mathsf{z}'}\right), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\right\rangle$$

$$\left(\left(\llbracket \overrightarrow{\mathsf{e}'} \rrbracket, \mathsf{op}_{\mathsf{e}'}\right), \left(\llbracket \overrightarrow{\mathsf{e}'} \rrbracket\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \overrightarrow{\mathsf{e}'}\right), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\right\rangle$$

$$\left(\left(\llbracket \overrightarrow{T} \rrbracket, \mathsf{op}_T\right), \left(\llbracket \overrightarrow{T} \rrbracket\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \overrightarrow{T}\right), \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey})\right\rangle$$

   - Sen and Rec remove the old accumulator corresponding to the inst_j'th index from the ZKBag. To simplify the notation, let $\rho = \ell + 2j - 1$.

$$\left(\left(\mathsf{bag}_\rho, \mathsf{state}_\rho\right), \left(\mathsf{bag}_\rho\right)\right) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Remove}} \left\langle \mathsf{Sen}\left(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}_{\rho-1}, \mathsf{state}_{\rho-1}, \left(\llbracket \overrightarrow{\mathsf{z}'} \rrbracket, \llbracket \overrightarrow{\mathsf{e}'} \rrbracket\right), (\mathsf{op}_{\mathsf{z}'}, \mathsf{op}_{\mathsf{e}'}), \left(\overrightarrow{\mathsf{z}'}, \overrightarrow{\mathsf{e}'}\right)\right), \mathsf{Rec}\left(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}_{\rho-1}, \left(\llbracket \overrightarrow{\mathsf{z}'} \rrbracket, \llbracket \overrightarrow{\mathsf{e}'} \rrbracket\right)\right)\right\rangle$$

   - Rec samples a random $r \leftarrow\$\, \mathbb{F}$ and sends it to Sen.
   - Sen and Rec update the inst_j'th accumulator

$$\llbracket \overrightarrow{\mathsf{e}} \rrbracket \leftarrow \llbracket \overrightarrow{\mathsf{e}}' \rrbracket + r \cdot \llbracket \overrightarrow{T} \rrbracket \qquad \llbracket \overrightarrow{\mathsf{z}} \rrbracket \leftarrow \llbracket \overrightarrow{\mathsf{z}}' \rrbracket + r \cdot \llbracket \overrightarrow{\mathsf{z}}_j \rrbracket$$

   and insert the updated accumulator in ZKBag. As before, let $\rho = \ell + 2j$.

$$\left(\left(\mathsf{bag}_\rho, \mathsf{state}_\rho\right), \left(\mathsf{bag}_\rho\right)\right) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Insert}} \langle \mathsf{Sen}(\mathsf{pp}, \mathsf{skey}, \mathsf{bag}_{\rho-1}, \mathsf{state}_{\rho-1}, (\llbracket \overrightarrow{\mathsf{z}} \rrbracket, \llbracket \overrightarrow{\mathsf{e}} \rrbracket), (\mathsf{op}_{\mathsf{z}}, \mathsf{op}_{\mathsf{e}}), (\overrightarrow{\mathsf{z}}, \overrightarrow{\mathsf{e}})) \; \mathsf{Rec}(\mathsf{pp}, \mathsf{rkey}, \mathsf{bag}_{\rho-1}, (\llbracket \overrightarrow{\mathsf{z}} \rrbracket, \llbracket \overrightarrow{\mathsf{e}} \rrbracket))\rangle$$

Figure 3: Part 1 of zero-knowledge protocol for verifying processor execution.

$$((b),(b)) \leftarrow \pi^{\mathsf{ZKDisj}}_{\mathsf{Verify}}\left\langle \mathsf{Sen}\left(\mathsf{pp},\mathsf{skey},\mathsf{Proof}^{\mathsf{ZK}},\mathsf{st},(\mathbf{A}_i,\mathbf{B}_i,\mathbf{C}_i)_{i\in[\ell]}\right),\mathsf{Rec}\left(\mathsf{pp},\mathsf{rkey},\mathsf{Proof}^{\mathsf{ZK}},(\mathbf{A}_i,\mathbf{B}_i,\mathbf{C}_i)_{i\in[\ell]}\right)\right\rangle$$

- Sen proceeds as follows for each $i \in [\ell]$
  1. Generate random relaxed R1CS instance $\mathsf{z}_{(i,2)} \leftarrow\$\ \mathbb{F}^m$, where $\overrightarrow{\mathsf{z}_{(i,2)}} = \overrightarrow{\mathsf{w}_{(i,2)}}\|\overrightarrow{\mathsf{x}_{(i,2)}}\|\overrightarrow{\mathsf{u}_{(i,2)}}$.
  2. Compute the corresponding error term
  $$\overrightarrow{L} \leftarrow (\mathbf{A}_i \cdot \overrightarrow{\mathsf{z}_{(i,2)}}) \circ (\mathbf{B}_i \cdot \overrightarrow{\mathsf{z}_{(i,2)}}) \qquad \overrightarrow{R} \leftarrow \mathsf{u}_{(i,2)} \cdot (\mathbf{C}_i \cdot \overrightarrow{\mathsf{z}_{(i,2)}}) \qquad \overrightarrow{\mathsf{e}_{(i,2)}} \leftarrow \overrightarrow{L} - \overrightarrow{R}$$
  3. Retrieve the i'th accumulator state $(\overrightarrow{\mathsf{z}_{(i,1)}},\overrightarrow{\mathsf{e}_{(i,1)}}) \leftarrow \mathbb{M}[i]$ and compute cross terms as
  $$\overrightarrow{\delta_1} \leftarrow \mathbf{A}_i \cdot \overrightarrow{\mathsf{z}_{(i,1)}} \circ \mathbf{B}_i \cdot \overrightarrow{\mathsf{z}_{(i,2)}} \quad \overrightarrow{\delta_2} \leftarrow \mathbf{A}_i \cdot \overrightarrow{\mathsf{z}_{(i,2)}} \circ \mathbf{B}_i \cdot \overrightarrow{\mathsf{z}_{(i,1)}} \quad \overrightarrow{\delta_3} \leftarrow \mathsf{u}_{(i,1)} \cdot \mathbf{C}_i \cdot \overrightarrow{\mathsf{z}_{(i,2)}} \quad \overrightarrow{\delta_4} \leftarrow \mathsf{u}_{(i,2)} \cdot \mathbf{C}_i \cdot \overrightarrow{\mathsf{z}_{(i,1)}}$$
  $$\overrightarrow{T_i} \leftarrow \overrightarrow{\delta_1} + \overrightarrow{\delta_2} - \overrightarrow{\delta_3} - \overrightarrow{\delta_4}$$
  4. Computes commitments to the two accumulators and the cross terms
  $$\left(\left(\left[\!\left[\overrightarrow{T_i}\right]\!\right],\mathsf{op}_{T_i}\right),\left(\left[\!\left[\overrightarrow{T_i}\right]\!\right]\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}\left\langle \mathsf{Sen}\left(\mathsf{pp},\mathsf{skey},\overrightarrow{T_i}\right),\mathsf{Rec}\left(\mathsf{pp},\mathsf{rkey}\right)\right\rangle$$
  $$\left(\left(\left[\!\left[\overrightarrow{\mathsf{z}_{(i,1)}}\right]\!\right],\mathsf{op}_{\mathsf{z}_{(i,1)}}\right),\left(\left[\!\left[\overrightarrow{\mathsf{z}_{(i,1)}}\right]\!\right]\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}\left\langle \mathsf{Sen}\left(\mathsf{pp},\mathsf{skey},\overrightarrow{\mathsf{z}_{(i,1)}}\right),\mathsf{Rec}\left(\mathsf{pp},\mathsf{rkey}\right)\right\rangle,$$
  $$\left(\left(\left[\!\left[\overrightarrow{\mathsf{z}_{(i,2)}}\right]\!\right],\mathsf{op}_{\mathsf{z}_{(i,2)}}\right),\left(\left[\!\left[\overrightarrow{\mathsf{z}_{(i,2)}}\right]\!\right]\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}\left\langle \mathsf{Sen}\left(\mathsf{pp},\mathsf{skey},\overrightarrow{\mathsf{z}_{(i,2)}}\right),\mathsf{Rec}\left(\mathsf{pp},\mathsf{rkey}\right)\right\rangle$$
  $$\left(\left(\left[\!\left[\overrightarrow{\mathsf{e}_{(i,1)}}\right]\!\right],\mathsf{op}_{\mathsf{e}_{(i,1)}}\right),\left(\left[\!\left[\overrightarrow{\mathsf{e}_{(i,1)}}\right]\!\right]\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}\left\langle \mathsf{Sen}\left(\mathsf{pp},\mathsf{skey},\overrightarrow{\mathsf{e}_{(i,1)}}\right),\mathsf{Rec}\left(\mathsf{pp},\mathsf{rkey}\right)\right\rangle,$$
  $$\left(\left(\left[\!\left[\overrightarrow{\mathsf{e}_{(i,2)}}\right]\!\right],\mathsf{op}_{\mathsf{e}_{(i,2)}}\right),\left(\left[\!\left[\overrightarrow{\mathsf{e}_{(i,2)}}\right]\!\right]\right)\right) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Commit}}\left\langle \mathsf{Sen}\left(\mathsf{pp},\mathsf{skey},\overrightarrow{\mathsf{e}_{(i,2)}}\right),\mathsf{Rec}\left(\mathsf{pp},\mathsf{rkey}\right)\right\rangle$$
- Rec samples a random $r \leftarrow\$\ \mathbb{F}$ and sends it to Sen.
- For each $i \in [\ell]$, Sen and Rec proceed as follows:
  1. Remove the i'th accumulator from ZKBag. To simplify notation, let $\tau = \ell + 2t + i$.
  $$((\mathsf{bag}_\tau,\mathsf{state}_\tau),(\mathsf{bag}_\tau)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Remove}}\left\langle \mathsf{Sen}\left(\mathsf{pp},\mathsf{skey},\mathsf{bag}_{\tau-1},\mathsf{state}_{\tau-1},\left(\left[\!\left[\overrightarrow{\mathsf{z}_{(i,1)}}\right]\!\right],\left[\!\left[\overrightarrow{\mathsf{e}_{(i,1)}}\right]\!\right]\right),\left(\mathsf{op}_{\mathsf{z}_i},\mathsf{op}_{\mathsf{e}_i}\right),\left(\overrightarrow{\mathsf{z}_i},\overrightarrow{\mathsf{e}_i}\right)\right),\right.$$
  $$\left.\mathsf{Rec}\left(\mathsf{pp},\mathsf{rkey},\mathsf{bag}_{\tau-1},\left(\left[\!\left[\overrightarrow{\mathsf{z}_{(i,1)}}\right]\!\right],\left[\!\left[\overrightarrow{\mathsf{e}_{(i,1)}}\right]\!\right]\right)\right)\right\rangle$$
  2. Accumulate with the blinding instance
  $$\left[\!\left[\overrightarrow{\mathsf{e}_i}\right]\!\right] \leftarrow \left[\!\left[\overrightarrow{\mathsf{e}_{(i,1)}}\right]\!\right] + r \cdot \left[\!\left[\overrightarrow{T_i}\right]\!\right] + r^2 \cdot \left[\!\left[\overrightarrow{\mathsf{e}_{(i,2)}}\right]\!\right] \qquad \left[\!\left[\overrightarrow{\mathsf{z}_i}\right]\!\right] \leftarrow \left[\!\left[\overrightarrow{\mathsf{z}_{(i,1)}}\right]\!\right] + r \cdot \left[\!\left[\overrightarrow{\mathsf{z}_{(i,2)}}\right]\!\right] \qquad \mathsf{u}_i = \mathsf{u}_{(i,1)} + r \cdot \mathsf{u}_{(i,2)}$$
- They check whether the ZKBag is empty $((1),(1)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}\left\langle \mathsf{Sen}\left(\mathsf{pp},\mathsf{skey},\mathsf{bag}_{2\ell+2t},\mathsf{state}_{2\ell+2t}\right),\mathsf{Rec}\left(\mathsf{pp},\mathsf{rkey},\mathsf{bag}_{2\ell+2t}\right)\right\rangle$.
- For each $i \in [\ell]$, Sen opens the following commitments to Rec
  $$((1),(\overrightarrow{\mathsf{e}_i})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Open}}\left\langle \mathsf{Sen}\left(\mathsf{pp},\mathsf{skey},\left[\!\left[\overrightarrow{\mathsf{e}_i}\right]\!\right],\mathsf{op}_{\overrightarrow{\mathsf{e}_i}},\overrightarrow{\mathsf{e}_i}\right),\mathsf{Rec}\left(\mathsf{pp},\mathsf{rkey},\left[\!\left[\overrightarrow{\mathsf{e}_i}\right]\!\right]\right)\right\rangle$$
  $$((1),(\overrightarrow{\mathsf{z}_i})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Open}}\left\langle \mathsf{Sen}\left(\mathsf{pp},\mathsf{skey},\left[\!\left[\overrightarrow{\mathsf{z}_i}\right]\!\right],\mathsf{op}_{\overrightarrow{\mathsf{z}_i}},\overrightarrow{\mathsf{z}_i}\right),\mathsf{Rec}\left(\mathsf{pp},\mathsf{rkey},\left[\!\left[\overrightarrow{\mathsf{z}_i}\right]\!\right]\right)\right\rangle$$
- Finally, for each $i \in [\ell]$, Rec verifies the extended witness
  $$\mathbf{A}_i \cdot \overrightarrow{\mathsf{z}_i} \circ \mathbf{B}_i \cdot \overrightarrow{\mathsf{z}_i} =^? = \mathsf{u}_i \cdot (\mathbf{C} \cdot \overrightarrow{\mathsf{z}_i}) + \mathsf{e}_i$$

Figure 4: Part 2 of zero-knowledge protocol for verifying processor execution.

We now show that view of the receiver when interacting with the simulator Sim is the computationally close to the view of the receiver interacting with the honest sender. We proceed with a hybrid argument. Let Hybrid_0 denote the interaction between the receiver and the honest sender.

- Hybrid_1 : Let Hybrid_1 be the same as Hybrid_0, but Sim simulates π^{ZKBag}. By the zero-knowledge property of π^{ZKBag}, the view of the receiver in Hybrid_1 and Hybrid_0 are computationally close.
- Hybrid_2 : This hybrid is similar to Hybrid_1, except that in the verification protocol in this hybrid, for each $i \in [\ell]$, Sim samples $\overrightarrow{\mathsf{z}_i}, \overrightarrow{\mathsf{e}_i}$ such that $\mathbf{A}_i \cdot \overrightarrow{\mathsf{z}_i} \circ \mathbf{B}_i \cdot \overrightarrow{\mathsf{z}_i} =^?= \mathsf{u}_i \cdot (\mathbf{C} \cdot \overrightarrow{\mathsf{z}_i}) + \mathsf{e}_i$. It uses these values and runs $\mathsf{Equiv}^{\mathsf{LCom}}$ to compute an equivocal opening for $\overrightarrow{\mathsf{z}_i}, \overrightarrow{\mathsf{e}_i}$ and invokes $\pi^{\mathsf{LCom}}_{\mathsf{Open}}$ using these openings. By equivocation property π^{LCom}, the view of the receiver in Hybrid_1 and Hybrid_2 are computationally close.
- Hybrid_3 This hybrid is the same as Hybrid_2, except that instead of computing commitments to honestly computed values, Sim computes commitments to $\overrightarrow{0}$. By the hiding property of π^{LCom}, view of receiver in Hybrid_2 and Hybrid_3 are computationally close.

Note that the view of the receiver in Hybrid_3 is distributed the same as the view of the receiver when interacting with the simulator above. Thus, we have concluded our proof.

Knowledge Soundness. Let $\mathcal{E}^{\mathsf{LCom}}$ be the extractor of the linearly homomorphic commitment scheme. Given a verifying proof transcript for π^{ZKDisj}, the extractor $\mathcal{E}$ for our π^{ZKDisj} protocol runs $\mathcal{E}^{\mathsf{LCom}}$ to simply extract the extended witness $\overrightarrow{\mathsf{z}_j}$ from $[\![\overrightarrow{\mathsf{z}_j}]\!]$, for each $j \in [t]$. The probability that the $\exists j^* \in [t]$, such that for each $i \in [\ell]$, the extracted $\overrightarrow{\mathsf{z}_j}$ is not a satisfying extended witness for $(\mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i)$ depends on the following:

- $\mathcal{E}^{\mathsf{LCom}}$ failed to extract the correct value, which only happens with negligible probability due to the binding property of π^{LCom}.
- The adversary succeeds in violating knowledge soundness of π^{ZKBag}. Recall from **??**, assuming the adversary cannot break the binding property of π^{LCom}, the statistical soundness error in our ZKBag with n insertions is $\leq \frac{2n}{|\mathbb{F}|}$. The ZKBag used in $\Pi_{\mathsf{ProveDisj}}$ has $n = \ell + t$ insertions, and therefore the probability that the adversary succeeds in breaking knowledge soundness of the underlying ZKBag is $\leq \frac{2\ell+2t}{|\mathbb{F}|}$, which is negligibly small.
- The adversary manages to cheat by successfully guessing at least one of the $t + \ell$ random challenges sampled by the verifier. Since the verifier samples these challenges uniformly at random from $\mathbb{F}$, this probability is $\frac{t+\ell}{|\mathbb{F}|}$, which is exponentially small for a large field.

Therefore, from binding property of π^{LCom}, it follows that this extractor fails to extract a satisfying set of extended witnesses from a verifying transcript with a statistically small probability $\leq \frac{3t+3\ell}{|\mathbb{F}|}$. □

7 Dora: Zero-Knowledge for RAM Programs

We construct Dora using our protocols from Section 5 and Section 6. A processor with a Von Neumann architecture consists of instructions $I = \{I_1, \ldots, I_\ell\}$ and maintains a local state $\overrightarrow{\mathsf{st}} = (\mathsf{pc}, \mathsf{Reg}_1, \ldots, \mathsf{Reg}_k)$, where pc denotes the program counter and we use $\mathsf{Reg}_1, \ldots, \mathsf{Reg}_k$ to refer to values stored in its local registers.

NP Relation $\mathcal{R}^{\mathsf{zkRAM}}$. To prove correct execution of a RAM program, we design a LinCom based commit-and-prove style zero-knowledge proof system (see 3.3) for $\mathcal{R}^{\mathsf{zkRAM}}$, defined as follows:

Definition 5 ($\mathcal{R}^{\mathsf{zkRAM}}$). *Let $\overrightarrow{M_0}$ denote the public initial state of the memory and $\overrightarrow{\mathsf{st}_0}$ denote the initial state of the processor. For each processor step $j \in [t]$, given commitments $[\![\overrightarrow{\mathsf{st}_j}]\!]$, $[\![\mathsf{inst}_j]\!]$, $[\![\overrightarrow{\mathsf{ReadVal}_j}]\!]$, $[\![\overrightarrow{\mathsf{OldWriteVal}_j}]\!]$, where $[\![\overrightarrow{\mathsf{st}_j}]\!]$ is a concatenation of commitments to the program counter $[\![\overrightarrow{\mathsf{pc}_j}]\!]$ and values stored in the registers including (but not limited to) $[\![\overrightarrow{\mathsf{ReadAddr}_j}]\!]$, $[\![\overrightarrow{\mathsf{WriteAddr}_j}]\!]$, $[\![\overrightarrow{\mathsf{WriteVal}_j}]\!]$, the prover wants to convince the verifier that it knows the corresponding values and opening information such that:*

– inst_j *is the value stored in the memory at location* $\overrightarrow{\mathsf{pc}_{j-1}}$.

- $\overrightarrow{\mathsf{ReadAddr}_j}$ *is stored in the appropriate registers in* $\overrightarrow{\mathsf{st}_{j-1}}$ *and* $\overrightarrow{\mathsf{ReadVal}_j}$ *is the value stored in memory at* $\overrightarrow{\mathsf{ReadAddr}_j}$.
- $\overrightarrow{\mathsf{st}_j}$ *(containing* $\overrightarrow{\mathsf{WriteAddr}_j}$, $\overrightarrow{\mathsf{WriteVal}_j}$ *in the appropriate registers) is the outcome of evaluating* I_{inst_j} *on input* $\overrightarrow{\mathsf{st}_{j-1}}$, $\overrightarrow{\mathsf{ReadVal}_j}$.
- *Old value* $\overrightarrow{\mathsf{OldWriteVal}_j}$ *at location* $\overrightarrow{\mathsf{WriteAddr}_j}$ *in the memory is replaced with* $\overrightarrow{\mathsf{WriteVal}_j}$.

For each $i \in [\ell]$, let $(\mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i)$ be an R1CS relation for a predicate checking if $\overrightarrow{\mathsf{st}'}$ (containing $\overrightarrow{\mathsf{WriteAddr}}$ and $\overrightarrow{\mathsf{WriteVal}}$) is the result of applying I_i on input $(\overrightarrow{\mathsf{st}}, \overrightarrow{\mathsf{ReadVal}})$.

Commit-and-Prove ZK Proof System for $\mathcal{R}^{\mathsf{zkRAM}}$. Let π^{LCom} be a linearly homomorphic commitment scheme, π^{zkDisj} be the protocol from Section 6 and π^{Memory} be the protocol from Section 5. Dora works as follows:

Setup: $\pi^{\mathsf{zkRAM}}_{\mathsf{Setup}}$:

Sen and Rec invoke $\pi^{\mathsf{LCom}}_{\mathsf{Setup}}$ to obtain pp, skey and rkey.

Prove: $\pi^{\mathsf{zkRAM}}_{\mathsf{Prove}}$:

We divide the prover protocol into an initialization phase and an execution phase:

I. Initialization Phase: Sen and Rec proceed as follows:

- Invoke $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ on $\overrightarrow{\mathsf{st}_0}$, $\overrightarrow{M_0}$ to get $\left[\!\left[\overrightarrow{\mathsf{st}_0}\right]\!\right]$ and $\left[\!\left[\overrightarrow{M_0}\right]\!\right]$.
- Invoke $\pi^{\mathsf{Memory}}_{\mathsf{Init}}$ on $\left[\!\left[\overrightarrow{M_0}\right]\!\right]$ to initialize the memory.
- Run the Initialization Phase of $\pi^{\mathsf{ZKDisj}}_{\mathsf{Prove}}$.

II. Execution Phase: For each $j \in [t]$:

- Invoke $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ to compute commitments $\left[\!\left[\overrightarrow{\mathsf{st}_j}\right]\!\right]$, $[\![\mathsf{inst}_j]\!]$, $\left[\!\left[\overrightarrow{\mathsf{ReadVal}_j}\right]\!\right]$, $\left[\!\left[\overrightarrow{\mathsf{OldWriteVal}_j}\right]\!\right]$, where $\left[\!\left[\overrightarrow{\mathsf{st}_j}\right]\!\right]$ includes commitments to the program counter $\left[\!\left[\overrightarrow{\mathsf{pc}_j}\right]\!\right]$ and register values including $\left[\!\left[\overrightarrow{\mathsf{ReadAddr}_j}\right]\!\right]$, $\left[\!\left[\overrightarrow{\mathsf{WriteAddr}_j}\right]\!\right]$, $\left[\!\left[\overrightarrow{\mathsf{WriteVal}_j}\right]\!\right]$. use $\overrightarrow{\mathsf{ReadVal}_j}$, $\overrightarrow{\mathsf{st}_{j-1}}$, and $\overrightarrow{\mathsf{st}_j}$ to compute a commitment to the extended witness $\left[\!\left[\overrightarrow{\mathsf{z}_{\mathsf{inst}_j}}\right]\!\right]$ for the relation $(\mathbf{A}_{\mathsf{inst}_j}, \mathbf{B}_{\mathsf{inst}_j}, \mathbf{C}_{\mathsf{inst}_j})$
- Invoke $\pi^{\mathsf{Memory}}_{\mathsf{Read}}$ to read $[\![\mathsf{inst}_j]\!]$ from address $\left[\!\left[\overrightarrow{\mathsf{pc}_{j-1}}\right]\!\right]$ and to read $\left[\!\left[\overrightarrow{\mathsf{ReadVal}_j}\right]\!\right]$ from address $\left[\!\left[\overrightarrow{\mathsf{ReadAddr}_j}\right]\!\right]$.
- Invoke $\pi^{\mathsf{Memory}}_{\mathsf{Update}}$ to replace $\left[\!\left[\overrightarrow{\mathsf{OldWriteVal}_j}\right]\!\right]$ with $\left[\!\left[\overrightarrow{\mathsf{WriteVal}_j}\right]\!\right]$ at the location $\left[\!\left[\overrightarrow{\mathsf{WriteAddr}_j}\right]\!\right]$.
- Finally, run the j^{th} step in the execution phase in $\pi^{\mathsf{ZKDisj}}_{\mathsf{Prove}}$ using $\left[\!\left[\overrightarrow{\mathsf{z}_{\mathsf{inst}_j}}\right]\!\right]$ and instruction index inst_j.

Verify: $\pi^{\mathsf{zkRAM}}_{\mathsf{Verify}}$:

Sen and Rec invoke $\pi^{\mathsf{Memory}}_{\mathsf{Verify}}$, $\pi^{\mathsf{ZKDisj}}_{\mathsf{Verify}}$ and $\pi^{\mathsf{LCom}}_{\mathsf{Open}}$ on $[\![\mathsf{st}_t]\!]$ and $[\![M_t]\!]$. Output 1, if all these checks verify.

We now prove the following Theorem.

Theorem 7.1. *Assuming that* π^{LCom} *in a secure linearly homomorphic commitment scheme (see Section 3.1),* π^{Memory} *is a protocol for checking memory consistency (see Section 5) and* π^{ZKDisj} *be a commit-and-prove zero-knowledge for* $\mathcal{R}^{\mathsf{ZKDisj}}$ *as defined in Section 6. Then the above protocol* $\pi^{\mathsf{zkRAM}} = (\pi^{\mathsf{zkRAM}}_{\mathsf{Setup}}, \pi^{\mathsf{zkRAM}}_{\mathsf{Prove}}, \pi^{\mathsf{zkRAM}}_{\mathsf{Verify}})$ *is a LinCom-based commit-and-prove zero-knowledge (Section 3.3) for* $\mathcal{R}^{\mathsf{zkRAM}}$ *with statistical soundness error bounded by* $\frac{2m+9t+3\ell}{|\mathbb{F}|}$.

Proof. **Correctness.** Correctness follows from correctness of Linearly Homomorphic commitment π^{LCom}, memory consistency protocol π^{Memory} and protocol for verifying processor execution π^{ZKDisj}.

Zero-Knowledge. Let $\mathsf{Sim}^{\mathsf{Memory}} = (\mathsf{Sim}^{\mathsf{Memory}}_{\mathsf{Setup}}, \mathsf{Sim}^{\mathsf{Memory}}_{\mathsf{Init}}, \mathsf{Sim}^{\mathsf{Memory}}_{\mathsf{Insert}}, \mathsf{Sim}^{\mathsf{Memory}}_{\mathsf{Remove}}, \mathsf{Sim}^{\mathsf{Memory}}_{\mathsf{VerEmpty}})$ be the simulator for π^{Memory} and let $\mathsf{Sim}^{\mathsf{ZKDisj}}$ be the simulator for π^{ZKDisj}. The simulator Sim for our π^{zkRAM} protocol proceeds like the Sen in an honest execution of π^{zkRAM}, except that: (1) Instead of running π^{Memory} honestly, it uses $\mathsf{Sim}^{\mathsf{Memory}}$ to simulate operations in π^{Memory}, (2) instead of running π^{ZKDisj} honestly, it uses $\mathsf{Sim}^{\mathsf{ZKDisj}}$ to simulate operations in π^{ZKDisj} and (3) whenever the parties invoke $\pi^{\mathsf{LCom}}_{\mathsf{Commit}}$ (except when committing to M_0), Sen computes a commitment to $\overrightarrow{0}$. Commitment to M_0 is computed honestly as described in the protocol.

We now show that view of the receiver when interacting with the simulator Sim is the computationally close to the view of the receiver interacting with the honest sender. We proceed with a hybrid argument. Let Hybrid_0 denote the interaction between the receiver and the honest sender.

- Hybrid_1 : Let Hybrid_1 be the same as Hybrid_0, but Sim simulates π^{Memory}. By the zero-knowledge property of π^{Memory}, the view of the receiver in Hybrid_1 and Hybrid_0 are computationally close.
- Hybrid_2 : This hybrid is similar to Hybrid_1, except that Sim simulates π^{ZKDisj}. By the zero-knowledge property of π^{ZKDisj}, the view of the receiver in Hybrid_1 and Hybrid_2 are computationally close.
- Hybrid_3 This hybrid is the same as Hybrid_2, except that instead of computing commitments to honestly computed (private) values, Sim computes commitments to $\overrightarrow{0}$. By the hiding property of π^{LCom}, view of receiver in Hybrid_2 and Hybrid_3 are computationally close.

Note that the view of the receiver in Hybrid_3 is distributed the same as the view of the receiver when interacting with the simulator above. Thus, we have concluded our proof.

Knowledge Soundness. Let $\mathcal{E}^{\mathsf{LCom}}$ be the extractor of the linearly homomorphic commitment scheme. Given a verifying proof transcript for π^{zkRAM}, the extractor $\mathcal{E}$ for our π^{zkRAM} protocol runs $\mathcal{E}^{\mathsf{LCom}}$ to simply extract the values from all commitments computed during the protocol. The probability that extracted values do not satisfy the relation $\mathcal{R}^{\mathsf{zkRAM}}$ as described in Section 7, depends on the following:

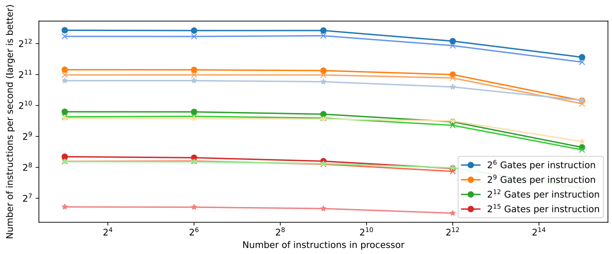
- $\mathcal{E}^{\mathsf{LCom}}$ failed to extract the correct value, which only happens with negligible probability due to the binding property of π^{LCom}.
- The adversary succeeds in violating knowledge soundness of π^{Memory}. Recall from **??**, assuming that the adversary cannot break the binding property of π^{LCom}, the statistical soundness error in our ZKBag with n insertions is $\leq \frac{2n}{|\mathbb{F}|}$. π^{Memory} is essentially a ZKBag with $n = m + 3t$ insertions and therefore, the probability that the adversary succeeds in breaking knowledge soundness of the underlying ZKBag is $\leq \frac{2m+6t}{|\mathbb{F}|}$, which is negligibly small.
- The adversary succeeds in violating knowledge soundness of π^{ZKDisj}, which as discussed in **??** happens probability $\leq \frac{3t+3\ell}{|\mathbb{F}|}$.

Therefore, assuming that the binding property of π^{LCom} holds, the overall probability that this extractor fails to extract a satisfying set of extended witnesses from a verifying transcript is $\leq \frac{2m+9t+3\ell}{|\mathbb{F}|}$, which is negligbly small. □

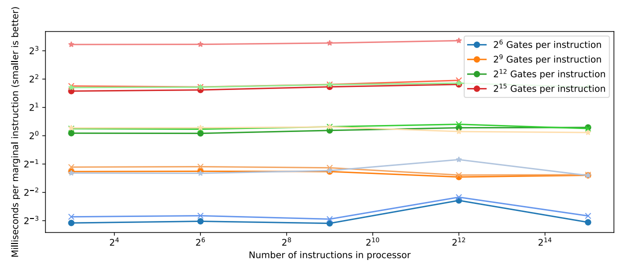
8 Implementation and Evaluation

We implement Dora and provide thorough micro-benchmarks for the sub-protocols described in Section 6 and Section 5.

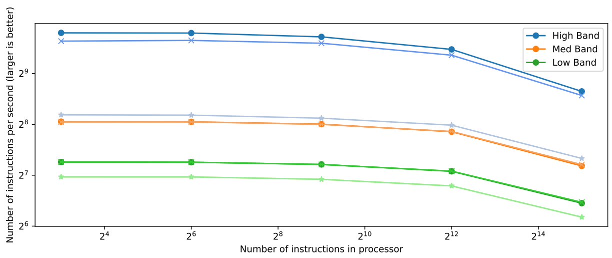
Optimizations. When implementing Dora, we integrate several minor optimizations. Namely, because of the high number of rounds in the ZKbag protocol we apply Fiat-Shamir to compute tag challenges, but we do not use Fiat-Shamir in the final consistency check. We note that, in general, the use of Fiat-Shamir in multi-round protocols can exponentially degrade security. This soundness loss comes from rewinding during extraction. Specifically, rewinding produces a tree of protocol transcripts, whose leaves grow exponentially in the number of rounds in a multi-round protocol (see, e.g., [AFK22]). This, however, is not a problem in Dora because we do not rewind during extraction; we rely on the underlying commitment scheme's extractor, circumventing the exponential loss.



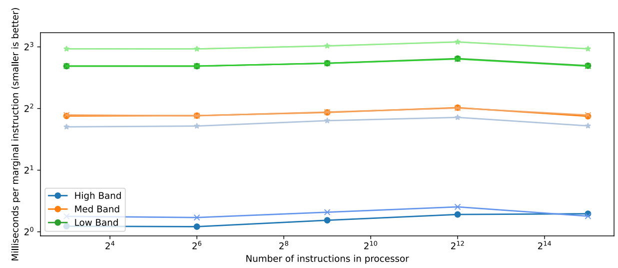
(a) Instructions/sec with Dora (over 50,000 steps). Larger is better.



(b) Marginal milliseconds per additional step. Smaller is better.



(c) Instructions/sec Dora can execute with instructions of size 2^{12} under different bandwidth configurations. Larger is better.



(d) Marginal milliseconds per additional step with instructions of size 2^{12} under different bandwidth configurations. Smaller is better.

Figure 5: Evaluations of Dora's performance with varying instruction sizes and network configurations. In graphs (a) and (b), the line's color represents the number of multiplication gates in each instruction used in the test (see legends). For each color there are three lines, each representing a different network latency: (1) the darkest hue, marked with $\bullet$, is 0ms latency, (2) the middle hue, marked with $\times$, is 10ms latency, and (3) the lightest hue, marked with $\star$, is 100ms. For these experiments, our setup had insufficient memory to evaluate a processor with 2^{15} instructions consisting of 2^{15} gates each. In graphs (c) and (d) we illustrate the impact of constrained bandwidth using the case of instructions with size 2^{12} gates. "High Band" represents a 1Gbit connection, "Med Band" represents a 100Mbit connection, and "Low Band" represents a 50Mbit connection. Differing marks and hues have consistent meanings regarding latency as in (a) and (b).

We note that the extractor of the underlying commitment scheme may, itself, rely on rewinding. As a result rewinding may be used during the full Dora extractor—which might seem to raise the specter of exponential security loss again. However, observe that we can extract each committed value immediately as the commitments are formed (before any other non-commitment specific messages are sent). As such, the *only* messages that are rewinded are those used to form commitments and we never rewind a Dora-specific message. Thus, we avoid the exponential transcript tree.

In our approach, the random oracle is only used to argue intractability. Within processor checks, we only require that for a commitment $c = Com(f)$ to a constant degree polynomial $f(X)$, if $\texttt{rand}$ is the output of the random oracle evaluated on c, then $f(\texttt{rand}) \neq 0$ with high probability, which is guaranteed by Schwartz-Zippel. When sampling tags within the ZKBag protocol, we only require that the random oracle does not output the same tag twice within a polynomial number of samples. Thus, because the security of ZKBag is statistical in the size of the field (a 61-bit prime field), we need to sample two field elements for each tag. The resulting protocol has ≈ 122 bits of computation security and remains designated verifier, as we still use VOLE for all commitments.

Concretely, the statistical soundness error of our largest tested parameters is $\approx 2^{-40}$. To see this, recall that Dora's soundness error is bounded by $\frac{2m+9t+3\ell}{|\mathbb{F}|}$ (Theorem 7.1) and we set $m = 2^{20}$, $t = 50,000 \approx 2^{16}$ and $\ell = 2^{15}$ for our largest tests. Note that the dominating factor here by *far* is m, so t could be dramatically increased without significantly degrading soundness.

Implementation and Benchmark Configuration. We implement Dora in Rust on top of Galois' swanky [Gal19]

Communication (MB) Gates in Instruction	Instructions in Processor 2^3	2^6	2^9	2^{12}	2^{15}
2^6	232.1	232.9	239.2	262.3	378.7
2^9	597.8	599.8	615.8	669.0	980.3
2^{12}	1734.3	1740.0	1788.3	1948.3	2864.7
2^{15}	4844.9	4860.7	4992.5	5437.7	-

Figure 6: Total communication for verifying the correct application of 50,000 processor steps, measured in MB.

Communication (KB) Gates in Instruction	Instructions in Processor 2^3	2^6	2^9	2^{12}	2^{15}
2^6	4.4KB	4.4KB	4.5KB	4.9KB	4.4KB
2^9	11.8KB	11.8KB	12.0KB	12.7KB	11.7KB
2^{12}	34.5KB	34.5KB	35.3KB	37.4KB	34.4KB
2^{15}	96.7KB	96.9KB	98.7KB	104.7KB	-

Figure 7: Marginal communication for verifying an *additional* processor step (in KB). Calculated by interpolating between 25,000 and 50,000 steps.

framework, a suite of secure computation and zero-knowledge tools.[15] Our code is intentionally designed to be interoperable with the emerging SIEVE intermediary representation (IR) [sie] standard such that it can interface with other emerging zero-knowledge techniques. To instantiate our linearly homomorphic commitments, we use vector oblivious linear evaluation (VOLE) base commitments, like other state-of-the-art interactive zero-knowledge protocols (e.g. QuickSilver [YSWW21]). `swanky` generates the prerequisite VOLE correlations using KOS OT-extension protocol [KOS15]. These correlations are computed "just-in-time," rather than in a pre-processing phase; the resulting interaction introduces a non-trivial overhead in our implementation which is included in all benchmarks. We also include all setup costs in our benchmarks. Our evaluation is done over a 61-bit prime field. We run the benchmarks on a single server (both prover and verifier are singled threaded) with 16 cores of AMD `Milan EPYC 7003` @ 2.4 Ghz with 64 GB of RAM. We note that while this machine has a lot of RAM, it is very slow compared to consumer grade laptops. We simulate network conditions using `tc(8)` and `netem(8)`.

8.1 Verifying Processor Execution

We begin by benchmarking our disjunctive zero-knowledge protocol that ensures each application of the processor circuit is done correctly (Section 6). We realize this protocol as a custom plugin for the SIEVE IR [sie] which takes in a set of functions (ie. the instructions) over which to do the disjunction. The result is a plugin that can be called with the appropriate number of inputs and outputs.

In order to benchmark this construction, we generate uniformly random instruction circuits over $\mathbb{F}_{2^{61}}$ with a prescribed number of multiplication gates. We do this by repeatedly sampling a random addition/multiplication gate with probability $1/2$ until the desired number of multiplication gates is reached. To connect these gates, we sample random input wires for each new gate from the set of previous output wires. The result is circuits with random topology, a good approximation for the worst case for efficiency.

Speed Benchmarks. We present our results in Figure 5:

(1) In Figure 5a, we show how many processor steps Dora proves per second for processors of varying complexity. These values are computed by proving 50,000 steps of the processor circuit, where a random instruction is chosen in each step. We vary the number of multiplication gates in each instruction in the set $\{2^6, 2^9, 2^{12}, 2^{15}\}$ and vary

[15]Code available at `https://github.com/rot256/research-dora/`

the number of instructions supported by the processor in the set $\{2^3, 2^6, 2^9, 2^{12}, 2^{15}\}$. Note that the overhead of setup and verifying the final R1CS instances grows as the number of instructions grows and the size of the instructions grows. When the number of instructions reaches 2^{15}, this overhead becomes non-trivial (compared to the fixed 50,000 steps) and begins to become visible in the benchmarks. We note that our machine ran out of memory for 2^{15} instructions of size 2^{15} simply because the overhead for holding the descriptions of the instructions was too high.

(2) In Figure 5b, we illustrate that the marginal cost of proving each additional step of the processor is constant in the number of instructions. To do this, we run the same experiment as in (1), but for 25,000 processor steps, interpolate between the two points and compute the time taken to prove each of the *additional* 25,000 steps. In this figure, the asymptotic characteristic of Dora becomes very clear: the marginal cost per-step is constant as the number of instructions in the processor increases.

(3) In Figure 5c and Figure 5d we replicate the above experiments while varying the bandwidth in the connection between the prover and the verifier (full benchmarks in Appendix A). Specifically, we test three bandwidth configurations: (i) a data-center-to-data-center"High Band" connection at 1Gbit, (ii) a standard consumer-grade "Med Band" connection at 100Mbit, and (iii) a low-quality "Low Band" connection at only 50Mbit. To illustrate the impact of constraining bandwidth on Dora's performance, we run 50,000 steps of a processor with a variable number of instructions (in the set $\{2^3, 2^6, 2^9, 2^{12}, 2^{15}\}$), each of size 2^{12}. Dropping bandwidth by a factor of 10 only cuts performance by a factor of 2, indicating that Dora is CPU bound.

Our benchmarks show that despite Dora's simple design, Dora is highly efficient—able to prove a marginal step of computation in less than 10ms, even with high network latency and large instructions.

Communication Benchmarks. We measure the total communication between the prover and the verifier—effectively the proof size—during the experiments described above. In Figure 6 we provide the *total* communication for running 50,000 steps of the processor with varying configuration, measured in MB. We additionally provide measurements of the additional communication incurred for each additional step of the proof in Figure 7. These are computed by interpolating between performance measurements for 25,000 and 50,000 steps of the relevant processor. Notice that each row of Figure 7 is largely constant, with jitter due to OT batching.

8.2 Verifying Memory Consistency

Recall that the total cost of proving a step of a processor in Dora is a single invocation of this protocol, plus several memory access operations. In this subsection we show that the costs of these memory operations are marginal compared to checking the processor instructions. As such, we use the benchmarks provided in Figure 5 as a good approximation of the overall performance of Dora.

Benchmarks. We present our memory benchmarks in two tables:

(1) In Figure 8 we present the average number of memory operations (READ/WRITE) per second, computed as an average over 2^{23} operations, when considering different network configurations (both bandwidth and latency). To illustrate that the size of the memory does not meaningfully impact performance, we run each experiment with ally memory space sizes in $\{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$. As above, our bandwidth configurations capture a 1Gbit High Bandwidth setting (data-center-to-data-center), a 100Mbit Medium Bandwidth setting (consumer-grade), and a 50Mbit Low Bandwidth setting (poor quality).

(2) In Figure 9 we show the marginal cost of each additional memory operation to highlight the asymptotic behavior of our construction. We do this by computing the difference in runtime for performing 2^{22} operations and 2^{23} operations.

(3) At the end of each table, we show the communication associated with memory accesses. In Figure 8, we show the *total* communication required to evaluate a large number of operations (in $\{2^{22}, 2^{23}, 2^{24}\}$) for different memory sizes. Then, at the end of Figure 9 we show that the additional communication associated with an additional memory operation is only ≈ 50 bytes. This is computed by interpolating between the communication measure for 2^{22} and 2^{23} memory operations.

Evaluation. Our results show that each memory operation takes only several microseconds, even with very high network latency. Given that these operations are several orders of magnitude faster than the processor instruction

High Bandwidth: Network Latency	Memory Space Size 2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
0 ms	331,946	334,580	334,167	325,316	310,861
10 ms	319,846	319,116	310,459	306,904	296,532
100 ms	173,626	173,705	172,179	168,968	164,134

Medium Bandwidth: Network Latency	Memory Space Size 2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
0 ms	174,985	174,446	173,569	171,304	162,658
10 ms	168,483	167,990	166,609	164,737	156,445
100 ms	122,165	122,960	121,115	120,394	114,766

Low Bandwidth: Network Latency	Memory Space Size 2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
0 ms	105,578	105,679	105,343	103,743	97,459
10 ms	103,335	102,497	99,842	98,703	93,011
100 ms	83,570	83,876	82,873	81,937	77,244

Communication (MB): Memory Operations	Memory Space Size 2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
2^{22}	214.2	214.6	216.6	224.5	258.6
2^{23}	423.6	424.1	426.0	433.9	465.4
2^{24}	839.7	840.3	842.2	850.1	881.5

Figure 8: Number of memory operations (READ/WRITE) per second, averaged over 2^{23} operations when running on `Intel i7-11800H`. Larger is better.

checks benchmarked above, we conclude that memory operations have a marginal impact on Dora's overall performance. We note that these results also show that performance starts to degrade when the network latency hits 100ms. This is an artifact of the on-demand nature of the VOLE computation in `swanky`. Because correlations are not computed upfront, the computation must pause in order to generate more VOLE correlations. Because this correlation generation protocol is a multi-round protocol, when the latency increases VOLE correlation generation dominates the overall cost. We emphasize that this is not a fundamental limitation of the protocol but rather a limitation of `swanky`, as OT correlations could be computed offline. Additionally, dropping bandwidth by a factor of 10 results in approximately twice the per-operation time, demonstrating the CPU is Dora's key resource.

8.3 Comparison with Other Approaches

We compare Dora with alternative approaches to proving the correct execution of RAM programs. For each of these comparisons, we lift performance figures from the associated papers.

Linear-sized Proofs. A direct alternative to Dora is using a linear-sized, linear-time zero-knowledge proof, which has seen significant recent breakthroughs e.g., [BMRS21, WYKW21, DOTV22, YSWW21, WYY$^+$22, BBMHS22], etc. The downside of this approach is that the proof must be over *entire* processor (as was done in [HK20a, HYDK21, YHKD22, GHAH$^+$23]). As the processor's branching factor increases, Dora 's performance improves relative to this approach. We estimate that Dora becomes more efficient than a naïve proof strategy (e.g., a commit-and-prove protocol using QuickSilver [YSWW21]) for processors with 4 or more instructions. This is because Dora computes commitments to 4 vectors (z, z', e, e') of instruction size, and its ZKBag operations are independent of the extended witness or R1CS relation.

Succinct Proofs. While succinct proofs have fast verification and small proof sizes, they suffer from slow prover

	Network Latency	Memory Space Size				
High Bandwidth		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
	0 ms	$3.09\mu s$	$2.95\mu s$	$3.05\mu s$	$3.07\mu s$	$2.61\mu s$
	10 ms	$3.16\mu s$	$3.06\mu s$	$3.29\mu s$	$3.28\mu s$	$2.73\mu s$
	100 ms	$5.68\mu s$	$5.63\mu s$	$5.68\mu s$	$5.80\mu s$	$4.84\mu s$
Medium Bandwidth	Network Latency	Memory Space Size				
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
	0 ms	$5.70\mu s$	$5.68\mu s$	$5.72\mu s$	$5.71\mu s$	$5.08\mu s$
	10 ms	$5.87\mu s$	$5.93\mu s$	$5.98\mu s$	$5.97\mu s$	$5.32\mu s$
	100 ms	$8.12\mu s$	$8.03\mu s$	$8.14\mu s$	$8.07\mu s$	$7.16\mu s$
Low Bandwidth	Network Latency	Memory Space Size				
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
	0 ms	$9.56\mu s$	$9.41\mu s$	$9.42\mu s$	$9.38\mu s$	$8.88\mu s$
	10 ms	$9.65\mu s$	$9.76\mu s$	$9.96\mu s$	$9.90\mu s$	$9.22\mu s$
	100 ms	$11.93\mu s$	$11.78\mu s$	$11.93\mu s$	$11.81\mu s$	$11.00\mu s$
Comm. (B)		Memory Space Size				
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
		49.9B	49.9B	49.9B	49.9B	49.3B

Figure 9: Marginal time for an *additional* memory operation, evaluated on `Intel i7-11800H`. Smaller is better. Notice that marginal cost is independent of memory size.

times. In contrast, Dora offers significantly faster prover times. To compare, we estimate the runtime for proving correct execution of processor instructions using state-of-the-art succinct proofs. Extending SNARKs to handle updatable memory efficiently would likely lead to non-trivial challenges (see [BCG+13]).

A naïve approach would represent the processor as a large circuit and prove it with a SNARK like Orion [XZS22], which has linear prover time. Orion's authors estimate proving an R1CS instance with 2^{20} constraints takes 3.09 seconds. Note that a single step ($t = 1$) of a processor circuit with 2^9 instructions, each with 2^{11} constraints *already* has 2^{20} constraints. In contrast, Dora proves one step of a similar processor in under 4ms (even with 100ms latency), making it ≈ 280 times faster.

Disjunction-optimized SNARKs like MuxProofs [DXNT23] and Sublonk [CGG+24] offer prover times proportional to the largest instruction. Sublonk, for example, proves $t = 16$ steps of a processor with $n = 2^{10}$ instructions, each of size 2^{16} in 20.04 seconds, which is ≈ 1.25 seconds per step, while Dora takes under 10ms, making it ≈ 125 times faster. The authors of Subplonk [CGG+24, Figure 3] claim that it takes ≈ 11 seconds to prove $t = 128$ steps of a processor with $n = 2^3$ instructions, each of size 2^{12}. Extrapolating naively, it will take $\approx 4,300$ seconds to prove $t = 50,000$ steps. In contrast, Dora proves $t = 50,000$ steps of the equivalent processor in only < 50 seconds with 100ms latency, making Dora ≈ 85 times faster.

Disjunction-Optimized Linear-sized Proofs. As discussed in Section 1.1, two works [YHH+23, YH23], developed concurrently with our own, also focus on designing zero-knowledge for proving correct execution of RAM programs. In Appendix A.2,we give a best-effort, apples-to-apples comparison between our approaches. We find that our memory approach is slightly slower (≈2x) than [YH23] while our processor approach is faster (1.5x-10x) than [YHH+23]. We note, however, that the comparison reduces to concrete constants, and thus even minor engineering choices could influence this comparison. More importantly, we believe that Dora's incredibly *simple* and intuitive design makes it independently interesting regardless of performance.

Acknowledgements

The authors would like to thank Yibin Yang for providing benchmarks by correspondence to help clarify the comparison with concurrent work provided in Appendix A.2. Majority of this work was done while the first author was at NTT Research and the third author was at Boston University. The second author is funded by Concordium Blockchain Research Center, Aarhus University, Denmark, implementation work was undertaken while at Galois partially funded by the FROMAGER (SIEVE) grant. The third author is supported by the National Science Foundation under Grant #2030859 to the Computing Research Association for the CIFellows Project and is supported by DARPA under Agreement No. HR00112020021. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

[ACF21] Thomas Attema, Ronald Cramer, and Serge Fehr. Compressing proofs of k-out-of-n partial knowledge. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 65–91, Virtual Event, August 2021. Springer, Cham. 5

[AFK22] Thomas Attema, Serge Fehr, and Michael Klooß. Fiat-shamir transformation of multi-round interactive proofs. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part I*, volume 13747 of *LNCS*, pages 113–142. Springer, Cham, November 2022. 27

[AOS02] Masayuki Abe, Miyako Ohkubo, and Koutarou Suzuki. 1-out-of-n signatures from a variety of keys. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 415–432. Springer, Berlin, Heidelberg, December 2002. 5

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018. 3

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. 6

[BBMH+21] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoît Razet, and Peter Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and Z2k. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 192–211. ACM Press, November 2021. 4

[BBMHS22] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-ole and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 329–358, Cham, 2022. Springer Nature Switzerland. 3, 31

[BC23] Benedikt Bünz and Binyi Chen. Protostar: Generic efficient accumulation/folding for special-sound protocols. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part II*, volume 14439 of *LNCS*, pages 77–110. Springer, Singapore, December 2023. 5

[BCC+16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, Berlin, Heidelberg, May 2016. 3

[BCF+21] Daniel Benarroch, Matteo Campanelli, Dario Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: Efficient, succinct, modular. In Nikita Borisov and Claudia Díaz, editors, *FC 2021, Part I*, volume 12674 of *LNCS*, pages 393–414. Springer, Berlin, Heidelberg, March 2021. 7

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Berlin, Heidelberg, August 2013. 3, 5, 6, 32

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014. 3

[BCG+22] Kenneth A. Bamberger, Ran Canetti, Shafi Goldwasser, Rebecca Wexler, and Evan Joseph Zimmerman. Verification dilemmas, law, and the promise of zero-knowledge proofs. *Berkeley Technology Law Journal*, 37(1), 2022. 3

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018. 4

[BCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: extended abstract. In Robert D. Kleinberg, editor, *ITCS 2013*, pages 401–414. ACM, January 2013. 5, 6

[BCGW22] Dor Bitan, Ran Canetti, Shafi Goldwasser, and Rebecca Wexler. Using zero-knowledge to reconcile law enforcement secrecy and fair trial rights in criminal cases. In *Proceedings of the 2022 Symposium on Computer Science and Law*, CSLAW '22, page 9–22, New York, NY, USA, 2022. Association for Computing Machinery. 3

[BCL+21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 681–710, Virtual Event, August 2021. Springer, Cham. 5

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive proof composition from accumulation schemes. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 1–18. Springer, Cham, November 2020. 5

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 103–128. Springer, Cham, May 2019. 3

[BCTV14a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Berlin, Heidelberg, August 2014. 3, 5

[BCTV14b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 781–796. USENIX Association, August 2014. 3, 5, 6

[BDFG21] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Proof-carrying data from additive polynomial commitments. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 649–680, Virtual Event, August 2021. Springer, Cham. 5

[BG12] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 263–280. Springer, Berlin, Heidelberg, April 2012. 14

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. 5

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 92–122, Virtual Event, August 2021. Springer, Cham. 3, 4, 5, 11, 31

[CCs08] Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 234–252. Springer, Berlin, Heidelberg, December 2008. 7

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 174–187. Springer, Berlin, Heidelberg, August 1994. 5

[CFH+15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 253–270. IEEE Computer Society, 2015. 6

[CFQ19] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2075–2092. ACM Press, November 2019. 6

[CGG+24] Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. Sublonk: Sublinear prover plonk. *Proc. Priv. Enhancing Technol.*, 2024(3):314–335, 2024. 6, 32

[CGT23] Alishah Chator, Matthew Green, and Pratyush Ranjan Tiwari. Sok: Privacy-preserving signatures. Cryptology ePrint Archive, Paper 2023/1039, 2023. https://eprint.iacr.org/2023/1039. 7

[CHP+23] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth:{Zero-Knowledge} proofs of real world vulnerabilities, 2023. 5

[CK18] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1687–1700. ACM Press, October 2018. 5

[CPS+16] Michele Ciampi, Giuseppe Persiano, Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. Online/offline OR composition of sigma protocols. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 63–92. Springer, Berlin, Heidelberg, May 2016. 5

[DIO21] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In Stefano Tessaro, editor, *2nd Conference on Information-Theoretic Cryptography, ITC 2021, July 23-26, 2021, Virtual Conference*, volume 199 of *LIPIcs*, pages 5:1–5:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 4, 5

[DOTV22] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of RAM programs from any public-coin zero-knowledge system. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22*, volume 13409 of *LNCS*, pages 615–638. Springer, Cham, September 2022. 3, 4, 5, 7, 31

[DXNT23] Zijing Di, Lucas Xia, Wilson Nguyen, and Nirvan Tyagi. Muxproofs: Succinct arguments for machine computation from tuple lookups. Cryptology ePrint Archive, Paper 2023/974, 2023. https://eprint.iacr.org/2023/974. 6, 32

[FKL+21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for RAM programs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 178–191. ACM Press, November 2021. 3, 5, 7, 14

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Berlin, Heidelberg, August 1987. 4

[Gal19] Galois, Inc. swanky: A suite of rust libraries for secure computation. https://github.com/GaloisInc/swanky, 2019. 4, 28

[gen20] genSTARK. https://github.com/guildofweavers/genstark, 2020. 6

[GGHAK22a] Aarushi Goel, Matthew Green, Mathias Hall-Andersen, and Gabriel Kaptchuk. Efficient set membership proofs using MPC-in-the-head. *PoPETs*, 2022(2):304–324, April 2022. 7, 9

[GGHAK22b] Aarushi Goel, Matthew Green, Mathias Hall-Andersen, and Gabriel Kaptchuk. Stacking sigmas: A framework to compose Σ-protocols for disjunctions. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 458–487. Springer, Cham, May / June 2022. 4, 5, 9

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Berlin, Heidelberg, May 2013. 3

[GHAH+23] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *PoPETs*, 2023(1):627–640, January 2023. 3, 5, 6, 7, 31

[GHAKS23] Aarushi Goel, Mathias Hall-Andersen, Gabriel Kaptchuk, and Nicholas Spooner. Speed-stacking: Fast sublinear zero-knowledge proofs for disjunctions. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part II*, volume 14005 of *LNCS*, pages 347–378. Springer, Cham, April 2023. 4, 5

[GK15] Jens Groth and Markulf Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 253–280. Springer, Berlin, Heidelberg, April 2015. 5

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985. 3

[GMW86] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design (extended abstract). In *27th FOCS*, pages 174–187. IEEE Computer Society Press, October 1986. 3

[GMY03] Juan A. Garay, Philip D. MacKenzie, and Ke Yang. Strengthening zero-knowledge protocols using signatures. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 177–194. Springer, Berlin, Heidelberg, May 2003. 5

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Report 2021/1063, 2021. 6

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Berlin, Heidelberg, May 2016. 3

[GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 415–432. Springer, Berlin, Heidelberg, April 2008. 3

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. 3

[HK20a] David Heath and Vladimir Kolesnikov. A 2.1 KHz zero-knowledge processor with BubbleRAM. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 2055–2074. ACM Press, November 2020. 3, 5, 6, 31

[HK20b] David Heath and Vladimir Kolesnikov. Stacked garbling - garbled circuit proportional to longest execution path. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 763–792. Springer, Cham, August 2020. 9

[HK20c] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 569–598. Springer, Cham, May 2020. 3, 4, 5, 9

[HK21] David Heath and Vladimir Kolesnikov. PrORAM - fast $P(\log n)$ authenticated shares ZK ORAM. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part IV*, volume 13093 of *LNCS*, pages 495–525. Springer, Cham, December 2021. 3, 5

[hod21] hodor. https://github.com/matter-labs/hodor, 2021. 6

[HYDK21] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast ZK processor with cached ORAM for ANSI C programs. In *2021 IEEE Symposium on Security and Privacy*, pages 1538–1556. IEEE Computer Society Press, May 2021. 3, 5, 6, 31

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013. 3

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018. 3

[Kol18] Vladimir Kolesnikov. Free `IF`: How to omit inactive branches and implement S-universal garbled circuit (almost) for free. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 34–58. Springer, Cham, December 2018. 5

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Berlin, Heidelberg, August 2015. 29

[KPPS20] Ahmed E. Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, and Dawn Song. MIRAGE: Succinct arguments for randomized algorithms with applications to universal zk-SNARKs. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 2129–2146. USENIX Association, August 2020. 6

[KS22] Abhiram Kothapalli and Srinath Setty. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Report 2022/1758, 2022. 4, 5, 10

[KS23] Abhiram Kothapalli and Srinath Setty. Hypernova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Paper 2023/573, 2023. https://eprint.iacr.org/2023/573. 4, 6

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 359–388. Springer, Cham, August 2022. 4, 5, 10, 12, 13, 21

[lib18] libSTARK. https://github.com/elibensasson/libstark, 2018. 6

[Lip16] Helger Lipmaa. Prover-efficient commit-and-prove zero-knowledge SNARKs. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 16*, volume 9646 of *LNCS*, pages 185–206. Springer, Cham, April 2016. 6

[MAGABMMT23] Héctor Masip-Ardevol, Marc Guzmán-Albiol, Jordi Baylina-Melé, and Jose Luis Muñoz-Tapia. eSTARK: Extending STARKs with arguments. Cryptology ePrint Archive, Report 2023/474, 2023. 6

[MGC+16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 112–127, 2016. 3

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society Press, May 2013. 7

[MRS17] Payman Mohassel, Mike Rosulek, and Alessandra Scafuro. Sublinear zero-knowledge arguments for RAM programs. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 501–531. Springer, Cham, April / May 2017. 7

[Nef01] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 116–125. ACM Press, November 2001. 8, 10, 14

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, Berlin, Heidelberg, August 1992. 4, 11

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 552–565. Springer, Berlin, Heidelberg, December 2001. 7

[se19] swisspost evoting. E-voting system 2019. https://gitlab.com/swisspost-evoting/e-voting-system-2019, 2019. 3

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Cham, August 2020. 3

[sie] Sieve intermediate representation. https://github.com/sieve-zk/ir. 29

[Wan] Xiao Wang. emp-tool. https://github.com/emp-toolkit/emp-tool. 3

[WSR+15] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*. The Internet Society, February 2015. 6

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*, pages 1074–1091. IEEE Computer Society Press, May 2021. 3, 4, 5, 31

[WYX⁺21] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 501–518. USENIX Association, August 2021. 3

[WYY⁺22] Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. AntMan: Interactive zero-knowledge proofs with sublinear communication. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2901–2914. ACM Press, November 2022. 3, 31

[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 299–328. Springer, Cham, August 2022. 32

[YH23] Yibin Yang and David Heath. Two shuffles make a ram: Improved constant overhead zero knowledge ram. Cryptology ePrint Archive, Paper 2023/1115, 2023. https://eprint.iacr.org/2023/1115. 5, 32, 42, 44

[YHH⁺23] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. Batchman and robin: Batched and non-batched branching for interactive zk. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1452–1466, New York, NY, USA, 2023. Association for Computing Machinery. 4, 5, 32, 42

[YHH⁺24] Yibin Yang, David Heath, Carmit Hazay, Vladimir Kolesnikov, and Muthuramakrishnan Venkitasubramaniam. Tight ZK CPU: Batched ZK branching with cost proportional to evaluated instruction. Cryptology ePrint Archive, Report 2024/456, 2024. 5, 42

[YHKD22] Yibin Yang, David Heath, Vladimir Kolesnikov, and David Devecsery. EZEE: Epoch parallel zero knowledge for ANSI C. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 109–123. IEEE, 2022. 3, 31

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, November 2021. 3, 4, 5, 22, 29, 31

[YWL⁺20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1607–1626. ACM Press, November 2020. 4

[Zav20] Greg Zaverucha. The picnic signature algorithm. Technical report, 2020. https://github.com/microsoft/Picnic/raw/master/spec/spec-v3.0.pdf. 3

[ZGK⁺18] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy*, pages 908–925. IEEE Computer Society Press, May 2018. 6

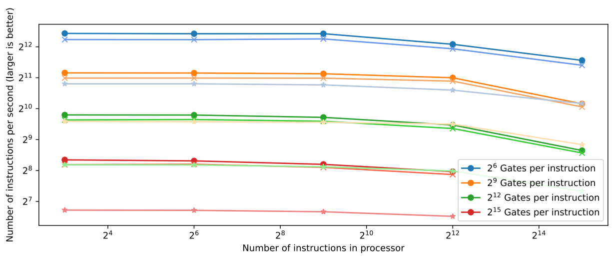
A Additional Benchmarks and Evaluation

In this section we provide additional benchmarks and evaluations that we did not include in the main body due to readability concerns. Namely, in Appendix A.1 we begin by presenting complete benchmarks for verifying correct executions of the processor under different bandwidth conditions and in Appendix A.2 we provide a best-effort comparison of Dora's performance with concurrent work.

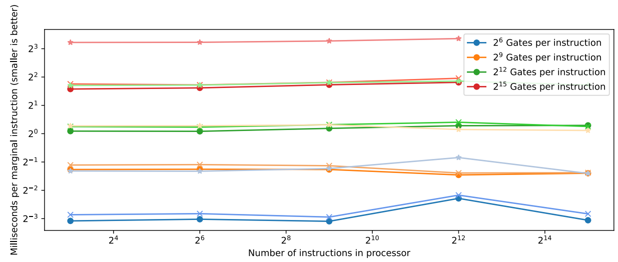
A.1 Additional Bandwidth Benchmarks

In Section 8, we presented an evaluation of Dora's performance under different network configurations. In order to preserve the readability of the main text, we chose to present only two suites of experiments: (1) in the first set of experiments, we kept bandwidth constant at 1Gbit and varied the size of the instructions in the processor in the set $\{2^6, 2^9, 2^{12}, 2^{15}\}$; (2) in the second set of experiments, we kept the size of the instructions in the processor constant at 2^{12}, but varied the bandwidth in the set {1Gbit, 100Mbit, 50Mbit}. For completeness, we include the full "cross product" of these sets in Figure 10 below. Note that the high bandwidth graphs are copied directly from Section 8 in order to make comparison easier for the reader.

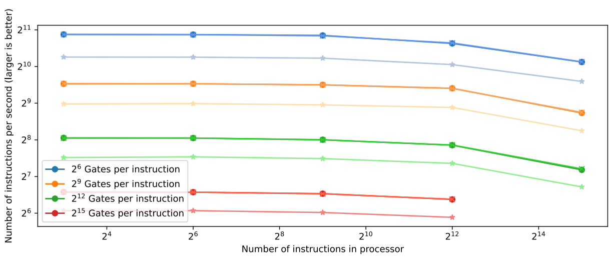
As discussed in the main body of the paper, we see Dora continues to operate efficiently even when bandwidth is constrained. Specifically, we see that reducing the available bandwidth from 1Gbit to 100Mbit only reduces the rate at which Dora can prove execution of a processor by a factor of 2. We believe this is largely because the most complex and time intensive parts of Dora are computation, rather than communication. We do, however, still see *some* impact when bandwidth is constrained. We believe this is because when messages are exchanged between the prover and the verifier (e.g., during OT correlation processing or the final batch proofs), the messages can be large. Thus, the total time it takes to transmit these messages increase when bandwidth reduces.



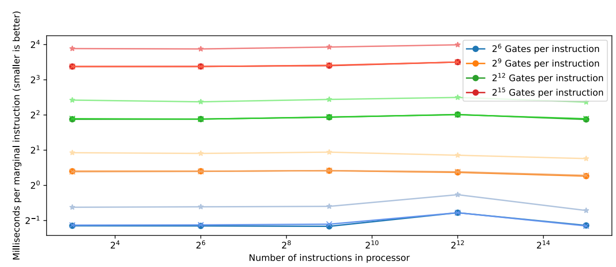
(a) Number of instructions per second Dora can execute in the High Bandwidth (1Gbit) setting. Larger is better.



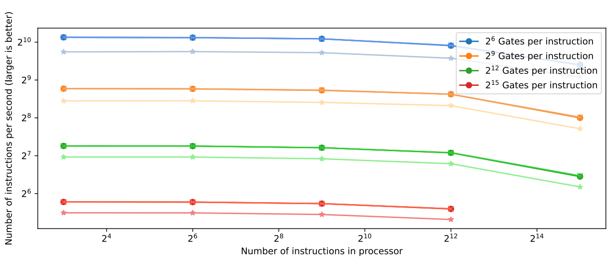
(b) Marginal milliseconds required to execute an additional step of the processor in the High Bandwidth (1Gbit) setting. Smaller is better.



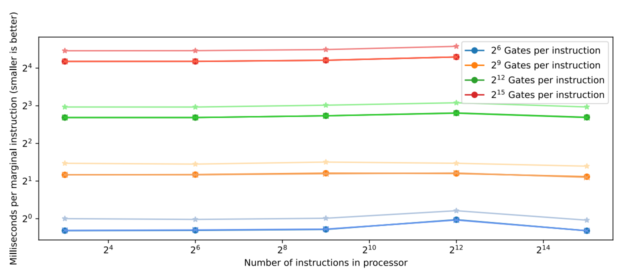
(c) Number of instructions per second Dora can execute in the Medium Bandwidth (100Mbit) setting. Larger is better.



(d) Marginal milliseconds required to execute an additional step of the processor in the Medium Bandwidth (100Mbit) setting. Smaller is better.



(e) Number of instructions per second Dora can execute in the Low Bandwidth (50Mbit) setting. Larger is better.



(f) Marginal milliseconds required to execute an additional step of the processor in the Low Bandwidth (50Mbit) setting. Smaller is better.

Figure 10: Evaluations of Dora's performance across different bandwidths. Graphs (a) and (b) are in the High Bandwidth setting, graphs (c) and (d) are in the Medium Bandwidth setting, and graphs (e) and (f) are in the Low Bandwidth setting. Hardware specifications and explanations for markers and hues can be found in Section 8.

A.2 Comparison with Concurrent Work

Two works [YHH+23, YH23], developed concurrently with our own, outline a similar approach to proving the correct execution of RAM programs. More specifically, Yang et al. [YHH+23] proposed Batchman and Robin, a pair of techniques that produce interactive zero-knowledge specially designed for proving a batch of disjunctions (eg. a set of processor circuits). Yang and Heath [YH23] proposed a new approach for creating zero-knowledge random access memory based on a pair of permutation proofs. It is straight forward to combine these two works to achieve a RAM zero-knowledge protocol with better/similar concrete performance as Dora; indeed, in follow up work completed subsequently to our own, some of these authors have done exactly this [YHH+24].

In this section we give an apples-to-apples comparison of Dora with [YHH+23] and [YH23]. At the highest level, this apple-to-apples comparison shows that Dora has a notable performance advantage over [YHH+23] when being used to prove the correct execution of the processor in each step, while the memory handling techniques proposed in [YH23] appear to outperform the techniques used in Dora by a factor of two.[16]

Batchman and Robin [YHH+23]. Yang et al. [YHH+23] begin by proposing Robin, a more communication efficient approach to disjunctive, VOLE-based zero-knowledge. Their key insight is that the prover and verifier, given a linearly homomorphic commitment to an extended witness, can compress that satisfiability check of each clause in the disjunction down to a constant size check (ie. if a committed value is 0). This protocol requires only a single random challenge from the verifier. They then propose Batchman, a way to *batch* many instances of these disjunctive statements together. They accomplish this by having the prover commit to the branch they want to satisfy in each statement in the batch, and then do a bespoke membership proof to show that the commitment contains a valid clause.

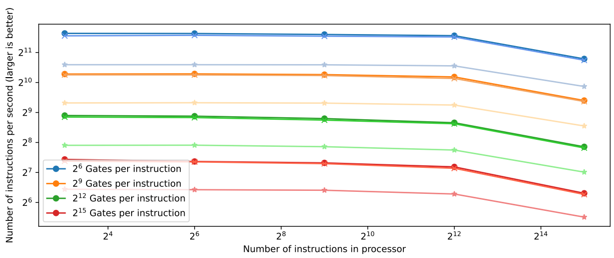
We note that Batchman does a small linear amount of work in the number of clauses in the disjunction, meaning Dora's asymptotic behavior is slightly better. However, we believe that using a ZKBag (or the read-only memory construction from [YH23]), the scheme can be improved to avoid this linear dependence on the number of clauses.

On the next page (Figure 12), we provide benchmarks for proving disjunctions, both absolute times and marginals, with Dora on equivalent hardware used to evaluate Batchman. In order to attempt to provide apples-to-apples comparisons of our evaluations, we contacted the authors of [YHH+23] to obtain results for a greater number of clauses on a machine similar to the server (`Intel Xeon Platinum 8259C`) we used for our benchmarks. We summerize these results, along with the appropriate direct comparisons from our Dora evaluation, in Figure 11. Although their setup is slightly different (e.g. consisting of two independent colocated machines), we observe that Dora has significantly better concrete performance, especially at the large processor size regime. Concretely, we see that for processor with 2^{15} instructions, Dora has between 1.5x-10x better performance than Batchman, depending on the exact network configurations and processor size.

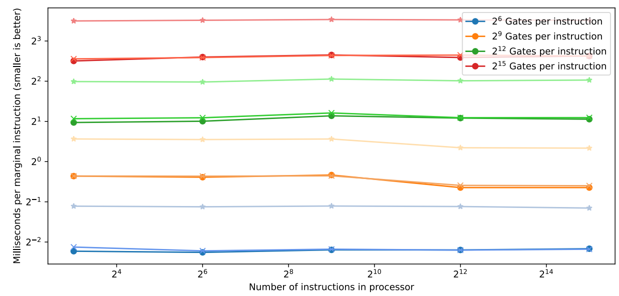
Network Bandwidth	Batchman			Dora			Performance Ratio		
	Size of Instructions			Size of Instructions			Size of Instructions		
	2^9	2^{12}	2^{15}	2^9	2^{12}	2^{15}	2^9	2^{12}	2^{15}
50 Mbps	19.42	11.84	2.89	193.61	66.52	23.57	9.97x	5.62x	8.16x
100 Mbps	30.09	21.77	5.16	269.52	93.17	32.91	8.96x	4.28x	6.38x
1 Gbps	148.62	82.88	17.88	375.98	129.10	45.64	2.53x	1.56x	2.43x

Figure 11: Performance of Batchman, as provided in private communication with the authors. The numbers shown are the number of steps applications per second Batchman performs when there are 2^{15} instructions, where the size of each instruction varies by column. (The conference version of this work had a typo in this table which was propagated throughout the rest of the text. This has been fixed in this version.)

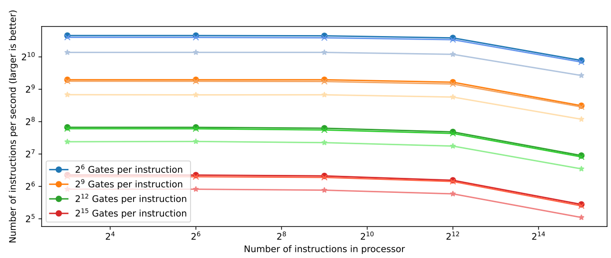
[16] As noted in the introduction, the initial version of this work misreported this comparison due to an error in our implementation. Specifically, we benchmarked Dora's performance on a 128-bit field, which concurrent work benchmarked performance on a 61-bit field. The benchmarks reported in this version rectify this mistake.



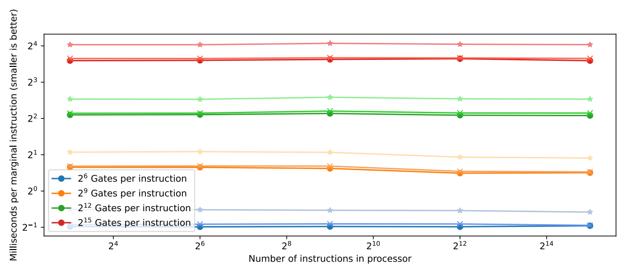
(a) Number of instructions per second Dora can execute in the High Bandwidth (1Gbit) setting using `Intel Xeon Platinum 8259C`. Larger is better.



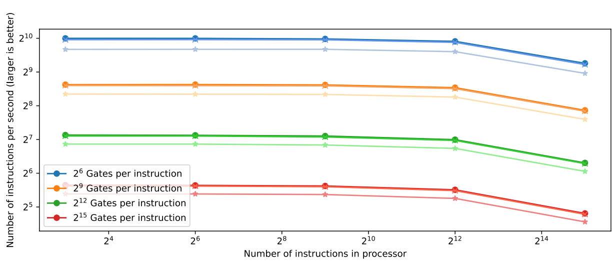
(b) Marginal milliseconds required to execute an additional step of the processor in the High Bandwidth (1Gbit) setting using `Intel Xeon Platinum 8259C`. Smaller is better.



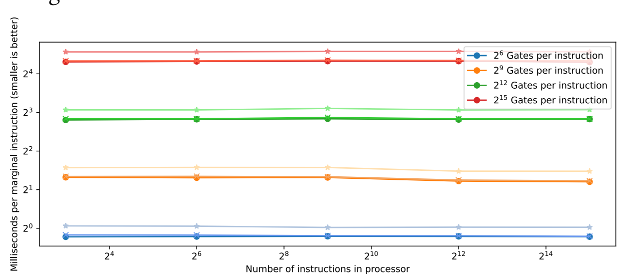
(c) Number of instructions per second Dora can execute in the Medium Bandwidth (100Mbit) setting using `Intel Xeon Platinum 8259C`. Larger is better.



(d) Marginal milliseconds required to execute an additional step of the processor in the Medium Bandwidth (100Mbit) setting using `Intel Xeon Platinum 8259C`. Smaller is better.



(e) Number of instructions per second Dora can execute in the Low Bandwidth (50Mbit) setting using `Intel Xeon Platinum 8259C`. Larger is better.



(f) Marginal milliseconds required to execute an additional step of the processor in the Low Bandwidth (50Mbit) setting using `Intel Xeon Platinum 8259C`. Smaller is better.

Figure 12: Evaluations of Dora's performance on `Intel Xeon Platinum 8259C`@ 2.50GHz with 128 GB of RAM. As with the equivalent graph in Section 8, the line's color represents the number of multiplication gates in each instruction used in the test (see legends). Graphs (a) and (b) are in the High Bandwidth setting, graphs (c) and (d) are in the Medium Bandwidth setting, and graphs (e) and (f) are in the Low Bandwidth setting.

Two Shuffles Make a RAM [YH23]. Yang and Heath also recently proposed a new approach for creating zero-knowledge random access memory. Their approach, which is very similar to ours, uses two permutation proofs to ensure that memory is treated consistently. While Dora uses time-stamping to ensure that a prover does not "read from the future," Yang and Heath use set membership proofs (which they implement using one of their permutation proofs). Their approach yields a *circuit* for random access memory, while ours results in a *protocol*. We provide benchmarks to compare concrete performance of our schemes, but note that the two share are conceptual core such that we would not anticipate performance to significantly diverge.

We provide equivalent benchmarks as in the main body of this work for Dora, but on equivalent hardware used in [YH23]. First, we perform 2^{23} memory operations when the memory space is set at some fixed size in $\{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$. The number of memory operations per second in reported in Figure 13 below. Note that performance slightly decreases, presumably because the absolute, single-threaded speed of the processor is lower—even if the processor as a whole is more powerful.

Next, in Figure 14 we report the *marginal*, per-access overhead of doing an additional memory operation. This is computed by observing the difference in total runtime between 2^{22} and 2^{23} memory operations. In Figure 10 of [YH23], the authors report $\sim 1.5\mu s$ per memory operation, while Dora's performance is $\sim 3\mu s$. We note that we see nothing inherent about the difference in performance between the two schemes and the gap in concrete performance may be an artifact of implementation details.

	Network Latency	Memory Space Size				
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
High Bandwidth	0 ms	262,382	257,367	257,469	253,916	245,863
	10 ms	252,054	250,766	246,231	242,445	236,432
	100 ms	149,046	149,200	147,945	145,968	141,358

	Network Latency	Memory Space Size				
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Medium Bandwidth	0 ms	151,198	151,444	151,222	148,350	141,807
	10 ms	146,531	147,076	145,729	143,788	138,027
	100 ms	111,210	109,863	110,439	109,396	104,383

	Network Latency	Memory Space Size				
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Low Bandwidth	0 ms	94,697	94,400	94,396	92,828	87,658
	10 ms	92,814	92,808	92,307	90,252	86,219
	100 ms	78,023	74,897	77,668	76,602	72,685

Figure 13: Number of memory operations (READ/WRITE) per second, averaged over 2^{23} operations when running on `Intel Xeon Platinum 8259C`. Larger is better. See Section 8 for details on the experimental setup.

High Bandwidth: Network Latency	Memory Space Size 2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
0 ms	$3.83\mu s$	$3.96\mu s$	$3.95\mu s$	$3.64\mu s$	$3.15\mu s$
10 ms	$3.97\mu s$	$3.96\mu s$	$4.05\mu s$	$4.13\mu s$	$3.25\mu s$
100 ms	$6.61\mu s$	$6.53\mu s$	$4.15\mu s$	$6.64\mu s$	$5.45\mu s$

Medium Bandwidth: Network Latency	Memory Space Size 2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
0 ms	$6.58\mu s$	$6.53\mu s$	$6.54\mu s$	$6.64\mu s$	$5.72\mu s$
10 ms	$6.83\mu s$	$6.77\mu s$	$6.82\mu s$	$6.83\mu s$	$5.49\mu s$
100 ms	$8.89\mu s$	$9.07\mu s$	$8.42\mu s$	$8.92\mu s$	$7.68\mu s$

Low Bandwidth: Network Latency	Memory Space Size 2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
0 ms	$10.54\mu s$	$10.57\mu s$	$10.52\mu s$	$10.54\mu s$	$9.67\mu s$
10 ms	$10.75\mu s$	$10.72\mu s$	$10.75\mu s$	$10.93\mu s$	$9.78\mu s$
100 ms	$12.72\mu s$	$13.74\mu s$	$12.77\mu s$	$12.75\mu s$	$11.53\mu s$

Figure 14: Marginal time for an *additional* memory operation, evaluated on `Intel Xeon Platinum 8259C`. Smaller is better. See Section 8 for details on the experimental setup.