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

Aarushi Goel
aarushi@purdue.edu
Purdue University
West Lafayette, USA

Mathias Hall-Andersen
mathias@hall-andersen.dk
Aarhus University/Galois/zkSecurity
Aarhus/Portland/San Francisco
Denmark/USA/USA

Gabriel Kaptchuk
kaptchuk@umd.edu
University of Maryland
College Park, USA

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.

CCS Concepts

• **Security and privacy → Privacy-preserving protocols**.

Keywords

Zero-Knowledge Proofs, RAM Programs, Linearly Homomorphic Commitments

ACM Reference Format:
Aarushi Goel, Mathias Hall-Andersen, and Gabriel Kaptchuk. 2024. Dora: A Simple Approach to Zero-Knowledge for RAM Programs. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690213

1 Introduction

Zero-knowledge proofs and arguments[1] [45, 46] 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 [7, 8, 13, 18, 21, 36, 39, 48, 50, 52, 56, 57, 71, 75–77, 79, 84]. As a result, zero-knowledge proofs are now being integrated as a key component of deployed systems [10, 72, 85].

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 [47, 52–54] or proving that software is legally compliant [5, 16].

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* [68, 74]: 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 [52]. 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 [12, 14, 15, 31, 35, 51, 53, 54] 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.

[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.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690213

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. [12, 15] introduced a processor called TinyRAM with only 29 instructions for this purpose, and recent works have created other processors with even fewer instructions [35, 51, 54]. 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, that it is a *disjunction* of the supported instructions. A sequence of recent works on disjunctive zero-knowledge [8, 41, 43, 52, 61–63] 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 [8, 31, 33, 75, 79, 82] 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 [41, 43, 52] and incrementally verifiable computation [61, 63]. 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 [6, 8, 20, 75, 79, 80] can be used, whereas Pedersen commitments [70] 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 [34].[5]
- *Concretely Efficient:* Dora is concretely efficient. We implement Dora and integrate it into the `swanky` [37] 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 (less powerful than a typical laptop), can prove correct execution of a program at >1000 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.

Concurrent Work. Two recent works [81, 82], developed concurrently with our own, focus on designing more efficient zero-knowledge random access memory [81] and for proving statements with processor-like structures [82]. Although not done, it is straightforward to combine these two works to achieve a RAM zero-knowledge protocol.[6]

[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 9.
[6] In a follow-up work [83], the authors of [81, 82] explore this approach.

A crucial difference between these works and Dora, lies in the relative simplicity of our approach. For instance, in [82], Heath et al. demonstrate how to adapt specific techniques from VOLE-based zero-knowledge proof systems [33, 79] 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. In the full version of our paper [42], we include a concrete, best-effort comparison to these concurrently developed works. We find that Dora offers faster proving times (1.5x-11x) for processor execution but is slower ($\approx 2x$) at updating memory.

1.2 Related Work

Zero-knowledge for RAM programs emerged as a problem of interest following the work of Ben-Sasson et al. [11, 12, 14, 15], 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 below). Recent works have improved performance, including the work of Heath et al. [51, 53, 54], Franzese et al. [35] and Delpech de Saint Guilhem et al. [31]. 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, eg. [25, 75, 79]. 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 [30, 47, 52].

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 [2, 29, 38] and a significant number of recent work [3, 8, 27, 41, 43, 49, 52, 59]. 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, eg. [3, 8, 41, 52], 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 [63] and SuperNova [61], which are a part of an emerging literature on concretely efficient IVC [17, 19, 22, 23]. In Nova [63] , 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. Kothapalli and Setty then proposed SuperNova [61], an extension of Nova that supports *non-uniform* IVC for "free," and discuss how to apply their techniques to verifying processor 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 [62], 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 [9, 26, 32, 40, 44, 55, 60, 64, 66, 73, 86] that focus on building concretely efficient zk-SNARKs (zero-knowledge succinct non-interactive arguments of knowledge), where the prover cost grows only with the size of the program execution. For instance, Buffet [73], vRAM [86], Mirage [60], MUX-Marlin [32] and Sublonk [26] 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 [9, 40, 44, 55, 64, 66] 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 [24, 28, 65] 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 Background: Template for RAM ZK

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 [11, 12, 15, 47, 51, 54]. 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* $\mathsf{C}_{\mathrm{proc}}$, and (2) a *memory checker circuit* $\mathsf{C}_{\mathrm{mem}}$. $\mathsf{C}_{\mathrm{proc}}$ 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.[7] The memory checker circuit $\mathsf{C}_{\mathsf{mem}}$ enforces that memory is treated consistently—that is, when a value is read from a particular memory address, $\mathsf{C}_{\mathsf{mem}}$ 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 $\mathsf{C}_{\mathsf{proc}}$ and $\mathsf{C}_{\mathsf{mem}}$.

Current Approaches to $\mathsf{C}_{\mathsf{proc}}$. Prior work has emphasized the need for a *small* $\mathsf{C}_{\mathsf{proc}}$, at the expense of expressiveness. For example, Ben-Sasson et al. [12] describe a minimal $\mathsf{C}_{\mathsf{proc}}$ called TinyRAM, which contains 27 instruction that can be represented in $\leq$ 972 gates.[8] This is because the final circuit contains t copies of $\mathsf{C}_{\mathsf{proc}}$, 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 $\mathsf{C}_{\mathsf{mem}}$. 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 (ADDRESS, VALUE) within an ORAM (eg. [67]), which is either maintained by the verifier (if the proof will be executed interactively) or represented in a non-black box manner within $\mathsf{C}_{\mathsf{mem}}$. 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 $\mathsf{C}_{\mathsf{mem}}$ 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 [31, 35, 47].

[7] 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.

[8] 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.

3 Our Approach

We now give an overview of the key techniques that we use to construct Dora. Given the simplicity of our approach, we emphasize the overview in this section is sufficient to understand our entire construction. However, for the sake of completeness, we also include a detailed description of Dora in the subsequent sections and a formal description in the Appendix. This formal description is deferred to the Appendix because formalism demands it is notationally complex, despite its conceptual simplicity.

3.1 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.

Constructing a ZKBag. It is clear to see that ZKBag is closely reminiscent of many existing cryptographic primitives, including *set membership proofs*. The main challenge is in ensuring that the communication and computation complexity of each insertion and retrieval operation is *constant* and independent of the total number of times these operations are called. This is important, because these interfaces will be called many (i.e., $O(t)$) times within Dora.

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 [69] (which we defer to the full version of our paper [42]).

3.2 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 $\mathsf{C}_{\mathsf{proc}}$ and $\mathsf{C}_{\mathsf{mem}}$.

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*[9]—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 $|I|$.

Handling Memory in Dora using ZKBag. Handling memory accesses with ZKBag is straightforward, as ZKBag's properties are virtually identical to those required for ensuring memory consistency. The prover and the verifier 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[10]:

(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.[11] 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. A baseline, intuitive approach would be to simply use ZKBag as a constant overhead *set membership proof*; at each step, the prover would do a (read-only) read of a *circuit* describing one of the instructions, and then prove that the committed circuit is satisfied. While intuitive, this would require either non-blackbox use of the commitment scheme or use of a special zero-knowledge proof system that supports committed constraints, either of which is sub-optimal.

Rather than store instructions in a ZKBag, we build on an approach from prior works on IVCs [61, 63] 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

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

[10] 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.

[11] 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.

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 [63]. 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 $\overrightarrow{x}$ is satisfied if there exists a witness $\overrightarrow{w}$ such that $\mathbf{A} \cdot \overrightarrow{z} \circ \mathbf{B} \cdot \overrightarrow{z} = \mathbf{C} \cdot \overrightarrow{z}$, where $\overrightarrow{z} = \overrightarrow{w} \| \overrightarrow{x}$. A relaxed R1CS relation injects two additional *error* parameters, $u \in \mathbb{F}$ and $\overrightarrow{e} \in \mathbb{F}^m$, and is satisfied if there exists a $\overrightarrow{z} = \overrightarrow{w} \| \overrightarrow{x} \| u$ such that $(\mathbf{A} \cdot \overrightarrow{z}) \circ (\mathbf{B} \cdot \overrightarrow{z}) = u \cdot (\mathbf{C} \cdot \overrightarrow{z}) + \overrightarrow{e}$. The power of relaxed R1CS is that it permits *folding*: given a fixed relation $\mathbf{A}, \mathbf{B}, \mathbf{C}$, and two instances $(\overrightarrow{x}_1, u_1, \overrightarrow{e}_1)$ and $(\overrightarrow{x}_2, u_2, \overrightarrow{e}_2)$, it is possible to combine the two into a new instance $(\overrightarrow{x}, u, \overrightarrow{e})$ for the same relation $\mathbf{A}, \mathbf{B}, \mathbf{C}$. Importantly, a prover can only satisfy the new instance $(\overrightarrow{x}, u, \overrightarrow{e})$ if they had valid witnesses $\overrightarrow{w}_1, \overrightarrow{w}_2$ to the initial instances (except with negligible probability). We defer the details of this folding procedure to Section 4.2.

Dora leverages this technique as follows: the prover and verifier initialize a ZKBag and (publicly) insert a relaxed R1CS instance (as defined by $\overrightarrow{e}$ and $\overrightarrow{z}$) 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 9 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.

4 Preliminaries

Due to space constraints, we defer certain preliminaries – including the definition and construction of a commit-and-prove zero-knowledge (ZK) proof, as well as the construction of Neff-style [69] multi-set equality proofs – to the full version of our paper [42].

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.

4.1 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 [8]) and *non-interactive* instantiations (eg. Pedersen [70]). Due to space constraints, we defer full version of these definitions to [42] and only include an informal description here.

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 $\mathsf{val}' \stackrel{?}{=} \mathsf{val}$.
- $((\mathsf{com}, \mathsf{op}), (\mathsf{com})) \leftarrow \pi^{\mathsf{LCom}}_{\mathsf{Comb}}$: 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.

Short-Hand Notation. For simplicity, we use the notation $[\![v]\!]$ denotes a commitment to some value $\overrightarrow{v}$. We often abuse notation and use $[\![\overrightarrow{x}]\!]$ to denote a linearly homomorphic commitment to a vector of elements in $\overrightarrow{x} \in \mathbb{F}^*$. 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.

4.2 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 [63]:

Definition 1 (Relaxed R1CS, [63]). *A relaxed R1CS (Rank-1 Constraint System) [63] is defined by three matrixes* $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}^{m \times m}$. *A witness* w *satisfies an instance* $(\overrightarrow{\mathsf{e}}, \overrightarrow{\mathsf{x}}, \mathsf{u})$ *iff. the "extended witness"* $\overrightarrow{\mathsf{z}} = \overrightarrow{\mathsf{w}} \| \overrightarrow{\mathsf{x}} \| \mathsf{u} \in \mathbb{F}^m$ *satisfies:* $(\mathbf{A} \cdot \overrightarrow{\mathsf{z}}) \circ (\mathbf{B} \cdot \overrightarrow{\mathsf{z}}) = \mathsf{u} \cdot (\mathbf{C} \cdot \overrightarrow{\mathsf{z}}) + \overrightarrow{\mathsf{e}}$. *For ease of notation, refer to Relaxed R1CS instances by their extended witness* $\overrightarrow{\mathsf{z}}$ *and error term* $\overrightarrow{\mathsf{e}}$*, which in turn defines* $\overrightarrow{\mathsf{w}}, \overrightarrow{\mathsf{x}}$, *and* u.

One valuable feature of Relaxed R1CS instances, as noted by [63], is that they can be "folded." That is, given two Relaxed R1CS instances $(\overrightarrow{\mathsf{z}_1}, \overrightarrow{\mathsf{e}_1})$ and $(\overrightarrow{\mathsf{z}_2}, \overrightarrow{\mathsf{e}_2})$ and a randomly sampled $r \in \mathbb{F}$, we can define a new instance $(\overrightarrow{\mathsf{z}}, \overrightarrow{\mathsf{e}})$ as:

$$\overrightarrow{\mathsf{e}} = \overrightarrow{\mathsf{e}_1} + r \cdot \overrightarrow{T} + r^2 \cdot \overrightarrow{\mathsf{e}_2}, \quad \mathsf{u} = \mathsf{u}_1 + r \cdot \mathsf{u}_2, \quad \overrightarrow{\mathsf{z}} = \overrightarrow{\mathsf{z}_1} + r \cdot \overrightarrow{\mathsf{z}_2}, \text{ where}$$

$$\overrightarrow{T} = \mathbf{A} \cdot \overrightarrow{\mathsf{z}_1} \circ \mathbf{B} \cdot \overrightarrow{\mathsf{z}_2} + \mathbf{A} \cdot \overrightarrow{\mathsf{z}_2} \circ \mathbf{B} \cdot \overrightarrow{\mathsf{z}_1} - \mathsf{u}_1 \cdot \mathbf{C} \cdot \overrightarrow{\mathsf{z}_2} - \mathsf{u}_2 \cdot \mathbf{C} \cdot \overrightarrow{\mathsf{z}_1}$$

Importantly, this folding process is *sound*, in that if either $(\overrightarrow{\mathsf{z}_1}, \overrightarrow{\mathsf{e}_1})$ or $(\overrightarrow{\mathsf{z}_2}, \overrightarrow{\mathsf{e}_2})$ are not satisfied, then $(\overrightarrow{\mathsf{z}}, \overrightarrow{\mathsf{e}})$ is also unsatisfied with high probability (over the choice of r). An additional fact about the folding scheme above (not directly used in Nova [63]) 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 1 where $\overrightarrow{\mathsf{e}} = \overrightarrow{0} \in \mathbb{F}^m$ and $\mathsf{u} = 1$. Throughout the section, to simplify notation, we will refer to relaxed R1CS instances by their error term $\overrightarrow{\mathsf{e}} \in \mathbb{F}^m$ and extended witness $\overrightarrow{\mathsf{z}} \in \mathbb{F}^m$; which define $\overrightarrow{\mathsf{w}}, \overrightarrow{\mathsf{x}}, \mathsf{u}$.

5 Zero-Knowledge Bag

As discussed in Section 3.1, 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.

5.1 Defining ZKBag

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. We omit formally writing out the inputs to each protocol due to space constraints, but they are included in the formal descriptions in the full version of our paper [42]:

- $((\mathsf{pp}, \mathsf{skey}), (\mathsf{pp}, \mathsf{rkey})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Setup}}$: Setup 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.
- $((\mathsf{bag}, \mathsf{state}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Init}}$: 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.
- $((\mathsf{bag}', \mathsf{state}'), (\mathsf{bag}')) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Insert}}$: 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.
- $((\mathsf{bag}', \mathsf{state}'), (\mathsf{bag}')) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Remove}}$: 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.
- $((b), (b)) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{VerEmpty}}$: 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*. Due to space constraints, we are unable to include formal definitions of these properties. For *correctness*, we require that 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. We formalize *knowledge soundness* using an extractor and *zero-knowledge* with a simulator in standard ways.

5.2 Realizing a ZKBag Protocol

Due to space constraints, we defer a formal description of our ZKBag protocol to the full-version of our paper [42]. At a high level, the protocol proceeds as follows[12]:

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

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

Initialize: $((\mathsf{bag}, \mathsf{state}), (\mathsf{bag})) \leftarrow \pi^{\mathsf{ZKBag}}_{\mathsf{Init}}$

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

[12]This notationally simple description of the zkbag protocol is sufficient to understand our construction.

bag $\mathcal{R}$, and (3) some private state $\mathbb{B}$ for the sender that will hold plaintext information about the committed values. $(\mathcal{I}, \mathcal{R})$ correspond to bag and $\mathbb{B}$ forms the private state of the sender.

Insert: $((\text{bag}', \text{state}'), (\text{bag}')) \leftarrow \pi^{\text{ZKBag}}_{\text{Insert}}$

In order to insert a (committed) item $[\![\overrightarrow{v}]\!]$ into the bag, the receiver samples a random tag $\leftarrow\!\!\$\, \mathbb{F}$. Both parties invoke $\pi^{\text{LCom}}_{\text{Commit}}$ on shared randomness to obtain $[\![\text{tag}]\!]$ and add $([\![\text{tag}]\!], [\![\overrightarrow{v}]\!])$ to the set of "input elements" $\mathcal{I}$. Additionally, the sender records the tag and values by adding $(\text{tag}, \overrightarrow{v})$ to $\mathbb{B}$.

Remove: $((\text{bag}', \text{state}'), (\text{bag}')) \leftarrow \pi^{\text{ZKBag}}_{\text{Remove}}$

To remove an element $\overrightarrow{v}$ from bag, the sender retrieves the corresponding tag using $\mathbb{B}$, creates fresh commitments to tag and $\overrightarrow{v}$. Both parties then add these fresh commitments $([\![\text{tag}]\!], [\![\overrightarrow{v}]\!])$ to the set of "removed elements" $\mathcal{R}$.

Final Verification: $((b), (b)) \leftarrow \pi^{\text{ZKBag}}_{\text{VerEmpty}}$

To ensure that the bag is empty, the parties simply check (set) equality of the inserted $\mathcal{I}$ and removed $\mathcal{R}$ elements using the $\pi^{\text{ZKMultiSet}} := (\pi^{\text{ZKMultiSet}}_{\text{Setup}}, \pi^{\text{ZKMultiSet}}_{\text{Prove}}, \pi^{\text{ZKMultiSet}}_{\text{Verify}})$ protocol.

We defer the proof of the following to the full version [42]:

Theorem 5.1. *Assuming that π^{LCom} in a secure linearly homomorphic commitment scheme (see Section 4.1), and $\pi^{\text{ZKMultiSet}}$ is a commit-and-prove style multi-set equality proof system,then π^{ZKBag}, shown in Section 5.2, is a LinCom-Based Zero-Knowledge Bag, as defined in Section 5.1.*

6 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 $(\text{addr}, \text{val})$, 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 $(\text{addr}, \text{val})$ 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.

6.1 Defining Memory Consistency

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: Init, Read, Update, Verify. As with ZKBag in the previous section, we do not write out the formal inputs to each protocol for ease of readability, but they can be seen in the full version of our paper [42]:

- $((\text{state}_P), (\text{state}_V)) \leftarrow \pi^{\text{Memory}}_{\text{Init}}$: 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.
- $((\text{state}_P), (\text{state}_V)) \leftarrow \pi^{\text{Memory}}_{\text{Update}}$: The prover and verifier take a commitment $([\![\text{addr}]\!], [\![\text{val}]\!])$ along with a commitment to the new value $[\![\text{val}']\!]$. Additionally, the prover takes in the actual value and opening to the commitments. The result is an updated state for each party.
- $((\text{state}_P), (\text{state}_V)) \leftarrow \pi^{\text{Memory}}_{\text{Read}}$: The prover and verifier take a commitment $([\![\text{addr}]\!], [\![\text{val}]\!])$—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.
- $((b), (b)) \leftarrow \pi^{\text{Memory}}_{\text{Verify}}$: 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).

6.2 A Protocol for Verifying Memory Consistency

We provide a formal description of our protocol π^{Memory} in the full-version of our paper [42]. In brief, this protocol is:

Initialize: $((\text{state}_P), (\text{state}_V)) \leftarrow \pi^{\text{Memory}}_{\text{Init}}$

During initialize, the parties invoke $\pi^{\text{ZKBag}}_{\text{Setup}}$ and $\pi^{\text{ZKBag}}_{\text{Init}}$ to initialize and setup a ZKBag. They use multiple invocations of $\pi^{\text{ZKBag}}_{\text{Insert}}$ to insert the initial contents of memory into the ZKBag.

Update: $((\text{state}_P), (\text{state}_V)) \leftarrow \pi^{\text{Memory}}_{\text{Update}}$

To update the value stored at some memory address addr, the parties first invoke $\pi^{\text{ZKBag}}_{\text{Remove}}$ to remove the old *address-value* tuple $([\![\text{addr}]\!], [\![\text{val}]\!])$ from ZKBag and then invoke $\pi^{\text{ZKBag}}_{\text{Insert}}$ to insert the new tuple $([\![\text{addr}]\!], [\![\text{val}']\!])$ into ZKBag. Importantly, the same commitment $[\![\text{addr}]\!]$ is used in both protocol invocations.

Read: $((\text{state}_P), (\text{state}_V)) \leftarrow \pi^{\text{Memory}}_{\text{Read}}$

To read the value stored at some memory address addr, the parties first invoke $\pi^{\text{ZKBag}}_{\text{Remove}}$ to remove the address-value tuple $([\![\text{addr}]\!], [\![\text{val}]\!])$ from ZKBag and then invoke $\pi^{\text{ZKBag}}_{\text{Insert}}$ to re-insert this tuple into ZKBag. The same commitments $[\![\text{addr}]\!]$ and $[\![\text{val}]\!]$ are used in both protocol invocations.

Verify: $((b), (b)) \leftarrow \pi^{\text{Memory}}_{\text{Verify}}$

At the end, to check if memory was consistently handeled throughout the protocol execution, the parties use multiple invocations of $\pi^{\text{ZKBag}}_{\text{Remove}}$ to remove all the remaining contents of ZKBag and then finally call $\pi^{\text{ZKBag}}_{\text{VerEmpty}}$.

The proof of the protocol follows trivially from ZKBag.

7 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 5) 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 6) to obtain a zero-knowledge proof system for RAM programs that enforces (1), (2), and (3).

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 full-version [42] for details) 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 4.1), 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 4.2, 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 $\mathcal{R}^{\mathsf{ZKDisj}}$. We design a commit-and-prove zero-knowledge proof system for $\mathcal{R}^{\mathsf{ZKDisj}}$ using a ZKBag protocol π^{ZKBag} (see Section 5) and the folding scheme for relaxed R1CS from [63]. Our protocol 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. 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 $\mathsf{inst}^{\mathsf{th}}_j$ instruction using (see Section 4.2) 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 $\mathsf{inst}^{\mathsf{th}}_j$ 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.

We include a formal description of this protocol and a proof of the following theorem in the full-version of our paper [42].

Theorem 7.1. *Assuming that π^{LCom} in a secure linearly homomorphic commitment scheme (see Section 4.1), and π^{ZKBag} is a zero-knowledge bag (see Section 5), then π^{ZKDisj} is a LinCom-based commit-and-prove zero-knowledge for $\mathcal{R}^{\mathsf{ZKDisj}}$.*

8 Dora: Zero-Knowledge for RAM Programs

We construct Dora using our protocols from Section 6 and Section 7. 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 for $\mathcal{R}^{\mathsf{zkRAM}}$, defined as follows:

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 $\llbracket\overrightarrow{\mathsf{st}_j}\rrbracket$, $\llbracket\mathsf{inst}_j\rrbracket$, $\llbracket\overrightarrow{\mathsf{ReadVal}_j}\rrbracket$, $\llbracket\overrightarrow{\mathsf{OldWriteVal}_j}\rrbracket$, where $\llbracket\overrightarrow{\mathsf{st}_j}\rrbracket$ is a concatenation of commitments to the program counter $\llbracket\overrightarrow{\mathsf{pc}_j}\rrbracket$ and values stored in the registers including (but not limited to) $\llbracket\overrightarrow{\mathsf{ReadAddr}_j}\rrbracket$, $\llbracket\overrightarrow{\mathsf{WriteAddr}_j}\rrbracket$, $\llbracket\overrightarrow{\mathsf{WriteVal}_j}\rrbracket$, 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 7 and π^{Memory} be the protocol from Section 6. 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 protocol into initialization and execution phases:

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 $[\![\overrightarrow{\mathsf{st}_0}]\!]$ and $[\![\overrightarrow{M_0}]\!]$.
- Invoke $\pi^{\mathsf{Memory}}_{\mathsf{Init}}$ on $[\![\overrightarrow{M_0}]\!]$ 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 $[\![\overrightarrow{\mathsf{st}_j}]\!]$, $[\![\mathsf{inst}_j]\!]$, $[\![\overrightarrow{\mathsf{ReadVal}_j}]\!]$, $[\![\overrightarrow{\mathsf{OldWriteVal}_j}]\!]$, where $[\![\overrightarrow{\mathsf{st}_j}]\!]$ includes commitments to the program counter $[\![\overrightarrow{\mathsf{pc}_j}]\!]$ and register values including $[\![\overrightarrow{\mathsf{ReadAddr}_j}]\!]$, $[\![\overrightarrow{\mathsf{WriteAddr}_j}]\!]$, $[\![\overrightarrow{\mathsf{WriteVal}_j}]\!]$. use $\overrightarrow{\mathsf{ReadVal}_j}$, $\overrightarrow{\mathsf{st}_{j-1}}$, and $\overrightarrow{\mathsf{st}_j}$ to compute a commitment to the extended witness $[\![\overrightarrow{\mathsf{z}_{\mathsf{inst}_j}}]\!]$ 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 $[\![\overrightarrow{\mathsf{pc}_{j-1}}]\!]$ and to read $[\![\overrightarrow{\mathsf{ReadVal}_j}]\!]$ from address $[\![\overrightarrow{\mathsf{ReadAddr}_j}]\!]$.
- Invoke $\pi^{\mathsf{Memory}}_{\mathsf{Update}}$ to replace $[\![\overrightarrow{\mathsf{OldWriteVal}_j}]\!]$ with $[\![\overrightarrow{\mathsf{WriteVal}_j}]\!]$ at the location $[\![\overrightarrow{\mathsf{WriteAddr}_j}]\!]$.
- Finally, run the j^{th} step in the execution phase in $\pi^{\mathsf{ZKDisj}}_{\mathsf{Prove}}$ using $[\![\overrightarrow{\mathsf{z}_{\mathsf{inst}_j}}]\!]$ 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.

Due to space constraints, we defer a formal proof of the following Theorem to the full-version of paper [42]:

Theorem 8.1. *Assuming that π^{LCom} in a secure linearly homomorphic commitment scheme (see Section 4.1), π^{Memory} is a protocol for checking memory consistency (see Section 6) and π^{ZKDisj} be a commit-and-prove zero-knowledge for $\mathcal{R}^{\mathsf{ZKDisj}}$ as defined in Section 7. 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 for $\mathcal{R}^{\mathsf{zkRAM}}$ with statistical soundness error bounded by $\frac{2m+9t+3\ell}{|\mathbb{F}|}$.*

9 Implementation and Evaluation

We implement Dora and provide thorough micro-benchmarks for the sub-protocols described in Section 7 and Section 6.

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., [4]). 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.

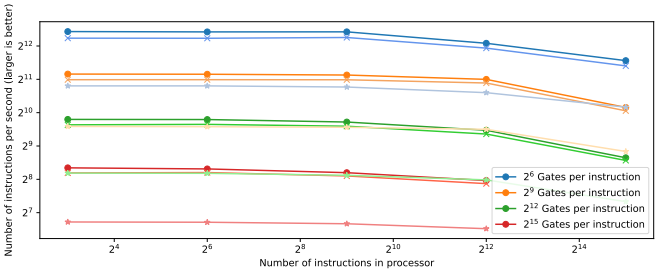
We note that the extractor of the underlying commitment scheme may, itself, rely on rewinding. As a result rewinding may be used during the full Dora extractor—which might seem to raise the specter of exponential security loss again. However, observe that we can extract each committed value immediately 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 rand is the output of the random oracle evaluated on c, then $f(\mathsf{rand}) \neq 0$ 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.

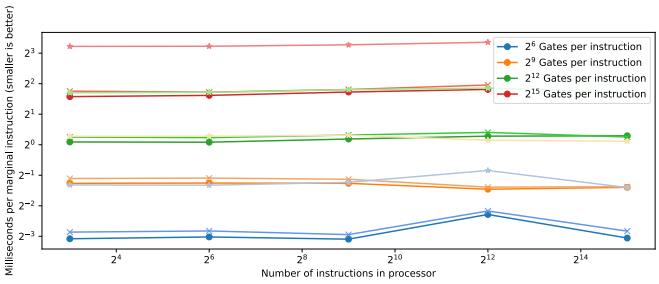
Concretely, the statistical soundness error of our largest tested parameters is $\approx 2^{-40}$. To see this, recall that Dora's soundness error is bounded by $\frac{2m+9t+3\ell}{|\mathbb{F}|}$ (Theorem 8.1) and we set $m = 2^{20}$, $t = 50,000 \approx 2^{16}$ and $\ell = 2^{15}$ for our largest tests. 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` [37] framework, a suite of secure computation and zero-knowledge tools.[13] Our code is intentionally designed to be interoperable with the emerging SIEVE intermediary representation (IR) [1] 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 [79]). `swanky` generates the prerequisite VOLE correlations using KOS OT-extension protocol [58]. 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

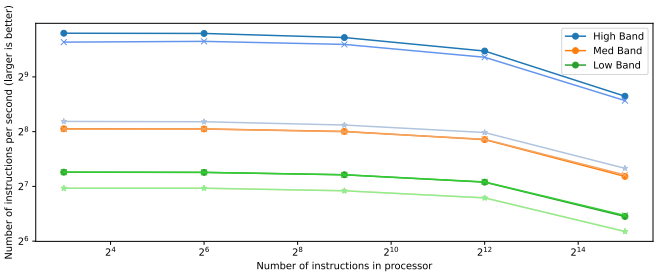
[13] Code available at https://github.com/rot256/research-dora/



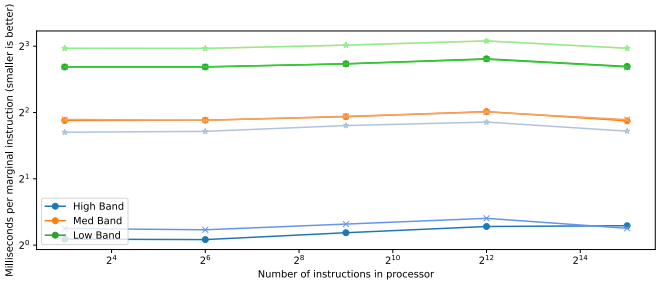
(a) Instructions/sec with Dora (over 50,000 steps). Larger is better.



(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 1: 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 •, is 0ms latency, (2) the middle hue, marked with ×, is 10ms latency, and (3) the lightest hue, marked with ⋆, 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).

(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)`.

9.1 Verifying Processor Execution

We begin by benchmarking our disjunctive zero-knowledge protocol that ensures each application of the processor circuit is done correctly (Section 7). We realize this protocol as a custom plugin for the SIEVE IR [1] 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 1:

(1) In Figure 1a, 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 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 1b, 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.

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 2: 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 3: Marginal communication for verifying an *additional* processor step (in KB). Calculated by interpolating between 25,000 and 50,000 steps.

(3) In Figure 1c and Figure 1d we replicate the above experiments while varying the bandwidth in the connection between the prover and the verifier (full benchmarks are provided in the full-version of our paper [42]). 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 2 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 3. 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 3 is largely constant, with jitter due to OT batching.

9.2 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 1 as a good approximation of the overall performance of Dora.

Benchmarks. We present our memory benchmarks in two tables:

(1) In Figure 4 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 5 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 4, 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 5 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 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.

9.3 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., [7, 8, 31, 75, 77, 79], etc. The downside of this approach is that the proof must be over *entire* processor (as was done in [47, 51, 54, 84]). As the processor's branching factor increases, Dora 's performance improves relative to this approach.

	Network Latency	Memory Space Size				
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
High Bandwidth	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	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	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 4: Number of memory ops/sec averaged over 2^{23} ops when running on `AMD Milan EPYC 7003`. Larger is better.

	Network Latency	Memory Space Size				
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
High Bandwidth	0 ms	3.09μs	2.95μs	3.05μs	3.07μs	2.61μs
	10 ms	3.16μs	3.06μs	3.29μs	3.28μs	2.73μs
	100 ms	5.68μs	5.63μs	5.68μs	5.80μs	4.84μs
Medium Bandwidth	0 ms	5.70μs	5.68μs	5.72μs	5.71μs	5.08μs
	10 ms	5.87μs	5.93μs	5.98μs	5.97μs	5.32μs
	100 ms	8.12μs	8.03μs	8.14μs	8.07μs	7.16μs
Low Bandwidth	0 ms	9.56μs	9.41μs	9.42μs	9.38μs	8.88μs
	10 ms	9.65μs	9.76μs	9.96μs	9.90μs	9.22μs
	100 ms	11.93μs	11.78μs	11.93μs	11.81μs	11.00μs
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 5: Marginal time for an *additional* memory operation, evaluated on `AMD Milan EPYC 7003`. Smaller is better. Notice that marginal cost is independent of memory size.

We estimate that Dora becomes more efficient than a naïve proof strategy (e.g., a commit-and-prove protocol using QuickSilver [79]) 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 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 [12]).

A naïve approach would represent the processor as a large circuit and prove it with a SNARK like Orion [78], 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 $\approx$ 280 times faster.

Disjunction-optimized SNARKs like MuxProofs [32] and Sublonk [26] 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 $\approx$ 1.25 seconds per step, while Dora takes under 10ms, making it $\approx$ 125 times faster. The authors of Subplonk [26, Figure 3] claim that it takes $\approx$ 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 $\approx$ 85 times faster.

Disjunction-Optimized Linear-sized Proofs. As discussed in Section 1.1, two works [81, 82], developed concurrently with our own, also focus on designing zero-knowledge for proving correct execution of RAM programs. In the full-version of our paper [42],we give a best-effort, apples-to-apples comparison between our approaches. We find that our memory approach is slightly slower (≈2x) than [81] while our processor approach is faster (1.5x-11x) than [82]. 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. 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

[1] Sieve intermediate representation. https://github.com/sieve-zk/ir.
[2] 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.
[3] 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.
[4] 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.
[5] 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.
[6] 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.
[7] 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.
[8] 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.
[9] 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.
[10] 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.
[11] 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.
[12] 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.
[13] 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.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019.
[20] 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.
[21] 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.
[22] 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.
[23] 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.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth: {Zero-Knowledge} proofs of real world vulnerabilities, 2023.
[31] 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.
[32] 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.
[33] 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.
[34] 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.
[35] 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.
[36] 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.
[37] Galois, Inc. swanky: A suite of rust libraries for secure computation. https://github.com/GaloisInc/swanky, 2019.
[38] 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.
[39] 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.
[40] genSTARK. https://github.com/guildofweavers/genstark, 2020.
[41] 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.

[42] Aarushi Goel, Mathias Hall-Andersen, and Gabriel Kaptchuk. Dora: Processor expressiveness is (nearly) free in zero-knowledge for RAM programs. Cryptology ePrint Archive, Report 2023/1749, 2023.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

[63] 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.

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

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

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

[73] 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.

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

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

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

[86] 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.