VDORAM: Towards a Random Access Machine with Both Public Verifiability and Distributed Obliviousness

Huayi Qi
qi@huayi.email
School of Computer Science and Technology
Shandong University
Qingdao, Shandong, China

Minghui Xu
mhxu@sdu.edu.cn
School of Computer Science and Technology
Shandong University
Qingdao, Shandong, China

Xiaohua Jia
Department of Computer Science
City University of Hong Kong
Kowloon, Hong Kong SAR, China

Xiuzhen Cheng
School of Computer Science and Technology
Shandong University
Qingdao, Shandong, China

ABSTRACT

Verifiable random access machines (vRAMs) serve as a foundational model for expressing complex computations with provable security guarantees, serving applications in areas such as secure electronic voting, financial auditing, and privacy-preserving smart contracts. However, no existing vRAM provides distributed obliviousness, a critical need in scenarios where multiple provers seek to prevent disclosure against both other provers and the verifiers. Implementing a publicly verifiable distributed oblivious RAM (VDORAM) presents several challenges. Firstly, the development of VDORAM is hindered by the limited availability of sophisticated publicly verifiable multi-party computation (MPC) protocols. Secondly, the lack of readily available front-end implementations for multi-prover zero-knowledge proofs (ZKPs) poses a significant obstacle to developing practical applications. Finally, directly adapting existing RAM designs to the VDORAM paradigm may prove either impractical or inefficient due to the inherent complexities of reconciling oblivious computation with the generation of publicly verifiable proofs.

To address these challenges, we introduce CompatCircuit, the first multi-prover ZKP front-end implementation to our knowledge. CompatCircuit integrates collaborative zkSNARKs to implement publicly verifiable MPC protocols with rich functionalities beyond those of an arithmetic circuit, enabling the development of multi-prover ZKP applications. Building upon CompatCircuit, we present VDORAM, the first publicly verifiable distributed oblivious RAM. By combining distributed oblivious architectures with verifiable RAM, VDORAM achieves an efficient RAM design that balances communication overhead and proof generation time. We have implemented CompatCircuit and VDORAM in approximately 15,000 lines of code, demonstrating usability by providing a practical and efficient implementation. Our performance evaluation result reveals that the system still provides moderate performance with distributed obliviousness.

KEYWORDS

Verifiable RAM, Distributed oblivious RAM, Zero-knowledge virtual machine, Multi-party computation, Zero-knowledge proof

1 INTRODUCTION

Random access machines (RAMs) serve as a crucial model for expressing complex computational logic. A RAM execute a program on its processor (register-based [3, 11, 13, 31, 50, 56, 76] or stack-based [25, 57, 59]) with a random access memory[1]. Table 1 summarizes our classification of RAMs based on obliviousness and verifiability:

	Verifiable to participants or designated verifiers	Publicly verifiable
Non-oblivious	vRAM [45]	vRAMs [25, 57, 59] [21, 60, 76]
Oblivious with non-distributed secrets	vRAMs [26, 37–39, 74]	vRAMs [3, 11, 13, 75] [19, 29, 31]
Oblivious with distributed secrets	DORAMs [23, 47, 50]	This work: **VDORAM**

Table 1: Comparison of random access machine schemes.

Verifiable RAM (vRAM). A vRAM enables a prover to execute a RAM program on an input and generate a proof of correct execution. *Verifiability:* vRAMs relying on private-coin interactive zero-knowledge proofs (ZKPs) offer verifiability exclusively to designated verifiers [26, 37–39, 45, 74]. In contrast, vRAMs based on non-interactive zero-knowledge (NIZK) proofs achieve public verifiability, producing proofs that can be verified by any entity, either with a privacy consideration [3, 11, 13, 19, 29, 31, 75] or without it [21, 25, 57, 59, 60, 76]. Some publicly verifiable vRAM schemes with succinctness are also known as zero-knowledge virtual machines (zkVMs) [3, 60] or zero-knowledge Ethereum virtual machines (zkEVMs) [25, 57, 59]. Succinctness means the proof size and verification complexity sub-linear in the size of the statement, i.e.,

[1]In this manuscript, the abbreviation "RAM" consistently refers to random access machines. Because the major challenge in a RAM involves implementing and checking random access memory, a proportion of related literatures may refer to "RAM" as random access memory.

the verification time is much less than the proof generation time. Consequently, succinct vRAMs often act as layer-2 scaling solutions for blockchain smart contracts. *Obliviousness:* Non-oblivious vRAMs, although most of them are based on ZKPs, prioritize performance [21, 60, 76] or compatibility [25, 57, 59] by compromising on privacy. Conversely, other vRAMs inherently exhibit obliviousness due to the zero-knowledge property, either with public verifiability [3, 11, 13, 19, 29, 31, 75] or without it [26, 37–39, 74], ensuring that no information about the prover's secret inputs is leaked to verifiers beyond what can be inferred from the output. However, a significant constraint of oblivious vRAMs lies in the fact that the prover, being a single entity, is required to possess knowledge of all the secrets. This requirement restricts their use in scenarios where secrets are distributed among multiple provers.

Distributed oblivious RAM (DORAM): In DORAMs, a group of parties collaboratively execute a RAM program, each exclusively holding partial inputs, without relying on a central trusted entity knowing all secrets. Distributed obliviousness ensures that no information is leaked to any party during program execution, except for the revealed output. DORAMs are generally built upon multi-party computation (MPC) techniques. Semi-honest DORAMs [56, 61] lack verifiability, while maliciously secure DORAMs [23, 47, 50] inherently achieves verifiability within the parties, which is different from public verifiability.

To the best of our knowledge, no existing RAM schemes simultaneously offer both public verifiability and distributed obliviousness. Such a RAM would be valuable in scenarios where verifiers cannot trust any of the provers [2, 64], while each prover seeks to prevent the disclosure of their secrets against other parties or verifiers. Such scenarios exactly match the need for a multi-user privacy-preserving smart contract [51, 66] in blockchain, where miners do not entrust any of the parties. We therefore pose the following question:

Can we construct a publicly verifiable distributed oblivious RAM (VDORAM)?

Given the necessity of succinctness to minimize the precious verification time required by blockchain miners, we advocate for the development of VDORAM via a publicly verifiable MPC[2] through leveraging multi-prover ZKPs [64]. However, the implementation of a VDORAM presents significant challenges. Firstly, current publicly verifiable MPC protocols lack the necessary functionalities to support the construction of a RAM. Secondly, front-end implementations for multi-prover ZKPs remain absent. Thirdly, directly applying existing RAM designs within a framework combining oblivious computation and public verifiability may result in either inefficiency or inapplicability. We detail these challenges below.

First, there is limited availability of sophisticated publicly verifiable MPC protocols. We exclude non-succinct publicly verifiable MPCs [5, 7, 8, 67, 70] due to the substantial size of their proofs and the considerable time required for verification. For succinct publicly verifiable MPCs [46, 64], computations are expressed as arithmetic circuits, providing supports for addition (subtraction) and multiplication gates within finite fields. While it is theoretically possible to implement a VDORAM where each value is represented as $W = 32$ or 64 bits and all computations are executed through boolean operations, this approach introduces a noticeable overhead due to simulating bits within fields. Thus, we prioritize word-based RAMs, where each value corresponds to a single field element, as opposed to multiple bits. However, word-based RAMs necessitate the implementation of more sophisticated MPC protocols [17, 62, 68], such as equality checks, bit decomposition, and comparison, which, as of now, lack publicly verifiable counterparts. Thus, directly applying publicly verifiable MPC to implement a VDORAM is not applicable.

Second, the absence of front-end implementations for multi-prover ZKPs creates a substantial barrier to construct practical applications. ZKP front-ends are critical for any practical ZKP applications as they lay out the procedures to transform input data into intermediate and final values, subsequently mapping them to rank-1 constraint system (R1CS) witnesses utilized by the ZKP back-ends that generate the final proof. Existing multi-prover ZKP schemes [16, 64, 71] serve as back-ends, assuming that the R1CS witnesses are already in possession of provers, which is not applicable for some real-world applications, especially a VDORAM. Ozdemir et al. [64] assume there exists a compiler that can convert an existing MPC protocol into an R1CS format, without providing implementation details and practical tools. Furthermore, considering that ZKP front-ends execute supplementary computational steps to generate auxiliary inputs for the proof process, the direct adoption of MPC implementations like MPyC [69] and MP-SPDZ [48] for multi-prover ZKPs is not straightforward. Therefore, developing fully functional front-end designs needs to address significant complexity before constructing a VDORAM on multi-prover ZKPs.

Third, existing RAM designs are likely unsuitable or perform poorly when used for computations that require both distributed obliviousness and public verifiability. On the one hand, vRAMs introduce state-of-the-art consistency checking mechanisms such as memory coherence checks [13, 19, 26] and lookup arguments [3, 27] to minimize proof generation time after R1CS witnesses are available. However, vRAMs typically do not prioritize optimizing the execution time of the front-end computation, which generates witnesses through the simulation of RAM execution from inputs, as this time is usually negligible in plaintext. This assumption no longer holds true in distributed oblivious computation, and thus additional communication overhead occurs. On the other hand, DORAMs seek to minimize communication and computation overheads in oblivious operations, with a focus on optimizing oblivious data structures [49], such as oblivious arrays [47, 50] and oblivious caches [23] to facilitate efficient accesses across distributed secrets. DORAMs do not take publicly verifiable proof generation into consideration. In summary, a novel RAM design is needed that effectively balances the demands of both oblivious computation and publicly verifiable proof generation.

To summarize, our contributions are as follows:

(1) We introduce CompatCircuit framework, which, to the best of our knowledge, the first multi-prover ZKP front-end implementation. It seamlessly integrates with collaborative zkSNARKs [64] and a dishonest-majority MPC framework. This enables the development of multi-prover ZKP applications with an arbitrary number of provers holding exclusive

[2]Also known as a publicly auditable MPC.

inputs. CompatCircuit provides a familiar development experience for single-prover ZKP applications while abstracting away the complexities of underlying MPC.

(2) We present VDORAM, the first publicly verifiable distributed oblivious RAM to our best knowledge. By combining distributed oblivious architectures within verifiable RAM, VDORAM achieves an efficient RAM design, balancing communication overhead and proof generation time.

(3) We implement CompatCircuit and VDORAM in approximately 15,000 lines of code. The codebase is publicly available at https://github.com/CompatCircuit/vdoram-artifacts, allowing researchers to easily reproduce our results and explore potential applications. Our performance evaluations, considering various parameters, such as the number of parties, instruction types, and instruction cycles, demonstrate that the system still provides moderate performance with the integration of distributed oblivious computation.

Roadmaps. Section 2 introduces related works. Section 3 details the model and preliminaries. In Section 4, we present our VDORAM design, including the CompatCircuit framework, memory management, the full protocol, and analysis. The implementation and evaluation are presented in Section 5. Section 6 concludes our work and discusses possible future directions.

2 RELATED WORK

This section presents a review of related work, including both succinct and non-succinct verifiable RAMs. Furthermore, it encompasses distributed oblivious RAMs.

Succinct vRAMs. Ben-Sasson et al. proposed the first SNARK-based verifiable RAM, TinyRAM [11], in which the execution proof can be formulated as a zkSNARK arithmetic circuit. Subsequently, vnTinyRAM [13] was introduced, which provides universality, making it independent of the specific program being executed. These efforts enable subsequent research in the field.

Several vRAMs prioritize compatibility with the Ethereum Virtual Machine (EVM), aiming to support smart contract rollups for blockchain layer-2 solutions. These vRAMs include Polygon zkEVM [59], zkSync [57], and Scroll [25]. Since the EVM inherently lacks support for private variables, these projects generally favor succinctness over the zero-knowledge property. In contrast, Polygon Miden [58] strives to be an EVM-incompatible vRAM that caters to privacy concerns within the blockchain space, at the expense of EVM compatibility.

Other notable succinct vRAMs include Cairo [33], which presents a verifiable RAM rendered in zkSTARK, utilizing an algebraic intermediate representation (AIR) rather than an arithmetic circuit/R1CS, coupled with a write-once memory model. Risc0 [75] is another STARK-based vRAM supporting the RV32IM ISA. Additionally, there are vRAMs with WebAssembly (WASM) support [30]. Recently, Arun et al. designed a RISCV-based vRAM called Jolt [3], which increases efficiency by implementing lookup singularity: all circuits solely perform lookups into pre-computed tables.

Non-succinct vRAMs. Some vRAMs sacrifice succinctness to achieve significantly better proof generation times. In contrast to employing zk-SNARKs, these vRAMs construct their proofs utilizing non-succinct zero-knowledge proof techniques such as MPC-in-the-Head [41], ZK from vector oblivious linear evaluation (VOLE) [6, 20, 73], and ZK from garbled circuits [42]. In this context, proof schemes operating under the private coin setting are inherently interactive, necessitating verifier participation in the proof generation process. Conversely, schemes reliant on public coins can be made non-interactive by employing the Fiat-Shamir transformation [24], thereby preserving public verifiability.

Heath et al. developed a ZK oblivious RAM, termed BubbleRAM [37], where the zero-knowledge proof is constructed using garbled schemes [42]. Subsequently, BubbleCache [39] was introduced, which enhanced efficiency through the implementation of multi-level caching. Franzese et al. built a constant-overhead interactive verifiable RAM [26], improving memory checking efficiency using a polynomial equality check. Goel et al. [31] developed their implementation, named Dora, based on the proposed concept of ZKBag. By introducing disjunctive zero-knowledge[6, 32, 53–55], their approach further enhances efficiency by allowing introducing additional instructions to the processor circuit at no extra cost. Saint Guilhem et al. proposed the construction of verifiable RAM based on public-coin ZKPs [19].

Distributed Oblivious RAMs. Differing from vRAMs that focus on public verifiability, distributed oblivious RAMs (DORAMs) [72] represent a significant area in a multi-party setting. Here, each party holds a portion of the secret and they work together to execute the RAM without relying on a trusted entity to hold all the secrets. Marcel Keller introduced a practical implementation of an oblivious machine within the arithmetic black-box model [47], leveraging SPDZ [18], an MPC protocol that offers active security. Subsequently, Keller et al. devised a garbled-circuit-based RAM with active security [50], which minimizes the number of broadcast rounds between memory accesses. Ji et al. [43] concentrated on constructing a RAM capable of private function evaluation. Falk et al. have introduced a 3-party distributed ORAM framework [23] that boasts logarithmic overhead and is resilient to malicious adversaries. In addition, Hamlin et al. proposed their first construction of FHE-based ORAM [35, 36].

Numerous efforts have been directed toward implementing RAM with a focus on either public verifiability or distributed obliviousness. However, research remains sparse in scenarios where both properties are of significance. The existence of a VDORAM, verifiable distributed oblivious random access machine, is a question that has not been comprehensively addressed.

3 MODEL AND PRELIMINARIES

This section begins with an introduction to our model, including system roles, threat model, and the essential properties of our VDORAM. Subsequently, we will introduce several underlying building blocks, including collaborative zkSNARKs, multi-party computation, arithmetic circuits, and RAM runtimes.

3.1 System and Threat Model

Our VDORAM is designed to create a distributed oblivious vRAM that enables a group of provers to collaboratively execute a given program using their individual inputs. Upon completion, the provers collectively convince non-interactive verifiers, attesting to the integrity of the outputs.

The following are the system roles along with the associated threat model.

- *Provers.* Provers are responsible for operating the VDORAM runtime, supplying input data, and generating zero-knowledge proofs for verifiers. *Correctness:* The group of provers is considered malicious: it is assumed that not fewer than $t = 0$ provers will act honestly, i.e., all provers are eager to collude to compromise the correctness of results for their personal gains. *Privacy:* Each prover is considered as a polynomial time adversary, curious to obtain confidential information from other provers. Except for the collusion case, provers tend to protect their secrets from being exposed.
- *Non-interactive verifiers.* Analogous to participants in traditional NIZK contexts, verifiers in our system can be any entity that assesses the validity of the proofs provided by the provers. These verifiers are not entrusted with any confidential or sensitive data, except for that which can be inferred from the public outputs.
- *Trusted Initializer.* They generate the public parameters for collaborative zkSNARKs, as well as precomputed data such as beaver triples for MPC.
- *Program vendors.* They publicly provide programs to be executed within the VDORAM runtime. These programs are audited and are trusted not to embed additional information in their outputs.

Our VDORAM necessitates the following properties.

- *Completeness.* Honest provers can complete the computation process and generate a valid proof.
- *Knowledge soundness.* Even in the event of collusion among all provers to produce a false proof, the probability of successfully passing the verification process remains negligible.
- *Succinctness.* The verification time should be significantly less than the computation and proof generation time.
- *Zero-knowledge.* Verifiers should not acquire any confidential information from any prover, except for those derived from public output.
- *Distributed obliviousness.* Each prover should not acquire any confidential information from any other prover, except for those derived from public output and private output meant to be revealed among provers.

The provers firstly utilize an MPC protocol for computing all necessary intermediate values as secret shares. Subsequently, they apply collaborative zkSNARKs [64] to generate a publicly verifiable proof using these secret shares as witnesses. We adopt a dishonest-majority MPC protocol supporting mixed boolean logic and arithmetic computations over a finite field $\mathbb{F}_p$ and the ring $\mathbb{Z}_{2^n}$. Below explains the considerations:

- *Integrating zkSNARKs.* Our threat model assumes that the entire set of provers may be malicious. While a maliciously secure dishonest-majority MPC protocol guarantees the privacy of computations as long as at least t provers behave honestly, we additionally require public verifiability. This is achieved by enabling independent verification of the computations' correctness through collaborative zkSNARKs. This approach ensures that even if all provers collude, a public verifier can still verify the integrity of the computation.
- *Mixed computations.* Collaborative zkSNARKs typically operate on statements expressed as arithmetic operations over a finite field $\mathbb{F}_p$.However, unlike a standalone MPC application, when MPC is used to support zkSNARKs, it must compute auxiliary data in addition to the primary computational result. This necessitates supporting computations in both the boolean domain and the ring $\mathbb{Z}_{2^n}$. This mixed-domain computation is essential for generating the necessary inputs for the zkSNARK proof system.

3.2 Preliminaries

3.2.1 Collaborative zkSNARKs. A zero-knowledge Succinct Non-interactive ARgument of Knowledge (zkSNARK) is a cryptographic protocol in which a prover $\mathcal{P}$ convinces a verifier $\mathcal{V}$ that a pair $(x, w) \in R$ without revealing w. Here, x is referred to as the instance or public input, and w serves as the private input, also known as the witness. Initially, zkSNARKs [15, 28, 34, 63] were designed for a single prover $\mathcal{P}$. Then, Ozdemir et al. expanded this scheme to a multi-prover scenario, introducing the concept of collaborative zkSNARKs [64]. In a collaborative zkSNARK, m provers $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{m-1}$ each hold a witness w_i and they collectively aim to convince the verifier $\mathcal{V}$ that $(x, w_0, w_1, \ldots, w_{m-1}) \in R$. This protocol contains the following algorithms:

- $\text{Setup}(1^\lambda, R) \rightarrow \text{pp}$: Produces the public parameters pp.
- $\text{Prove}(\text{pp}, x, w_0, w_1, ..., w_{m-1}) \rightarrow \pi$: Generates a proof π if $(x, w_0, w_1, ..., w_{m-1}) \in R$ in MPC; aborts otherwise.
- $\text{Verify}(\text{pp}, x, \pi) \rightarrow \{0, 1\}$: Validates the proof π.

An (m, t) collaborative zkSNARK has properties as follows:

- *Completeness*: Honest provers generate a valid proof when they have valid witnesses such that $(x, w_0, w_1, ..., w_{m-1}) \in R$.
- *Knowledge soundness*: Provers without knowledge of valid witnesses cannot produce a valid proof.
- *t-zero-knowledge*: If less than t provers collude, provers and verifiers cannot gain any information about witnesses w_0, $w_1, ..., w_{m-1}$ (excluding w_i for prover $\mathcal{P}_i$).
- *Succinctness*: Both the proof size and the verification time are $o(|R|)$, where $|R|$ denotes the size of relation R.

3.2.2 Multi-Party Computation. A multi-party computation (MPC) protocol [10, 18, 48] involves m parties, each performing computations to determine the outcome $y \leftarrow f(x_0, x_1, \ldots, x_{m-1})$, where f is a function defined as $f : X^m \rightarrow Y$. In this protocol, each party i possesses an input $x_i \in X$. An MPC protocol is considered secure if it reveals no additional information beyond that which is implicit in the output y. As long as no more than t parties collude, the protocol maintains its security.

In particular, our research focuses exclusively on dishonest-majority MPC protocols. In *dishonest-majority* MPC protocols, where typically $t = m-1$, additive secret sharing is utilized. In this scheme, a secret value a is distributed among the parties as $a_0, a_1, \ldots, a_{m-1}$, each party holding a share such that the sum of all shares equals

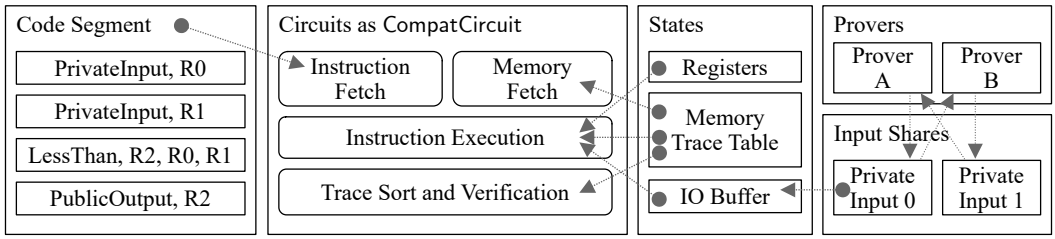

Figure 1: VDORAM architecture.

a. We use the notation $[a]$ to denote the secret shares of the value a held by each party. To compute addition $r \leftarrow a + b$, each party computes their share as $[r] \leftarrow [a] + [b]$ within ring or fields. To compute multiplication $r \leftarrow a \cdot b$, parties consume a pre-shared Beaver triple $([\alpha], [\beta], [\gamma])$ such that $\alpha \cdot \beta = \gamma$. They broadcast and reveal δ and ϵ where $[\delta] = [a] - [\alpha]$ and $[\epsilon] = [b] - [\beta]$. The final share is computed as $[r] \leftarrow [\gamma] + \delta \cdot [\beta] + \epsilon \cdot [\alpha] + \delta \cdot \epsilon$.

Publicly verifiable MPC, also known as publicly auditable MPC, is usually constructed by committing values to a public bulletin-board and constructing an NIZK proof checking the transcript [5, 7, 8, 46, 67]. Alternatively, Ozdemir et al. [64] proposed a novel publicly verifiable MPC construction from their proposed multi-prover ZKP, verifying statements in R1CS. We favor this latter approach due to its capability to enable the construction of efficient proofs focused on only checking the integrity of RAM states, as opposed to validating the entire sequence of computations leading to those states.

3.2.3 Arithmetic Circuit. An arithmetic circuit [9, 13, 14, 65] is a directed acyclic graph (DAG) consisting of gates and wires, commonly used in both MPC and zkSNARKs programming. Each wire carries a value within a finite field $\mathbb{F}_p$; additionally, they may also operate within a ring $\mathbb{Z}_{2^n}$ for MPC, which we will utilize later. Gates in the circuit take in wires as inputs and generate output wires based on the operation they perform – either addition or multiplication. Moreover, arithmetic circuits can be easily converted to rank-1 constraint system (R1CS), which is the most commonly used structure for representing zkSNARKs verification statements.

3.2.4 Verifiable Random Access Machine. We adopt the definition from [3] and [30]. A random access machine runtime is defined as a program receiving an input tuple $(\mathbf{I}, E, \mathsf{in})$, where $\mathbf{I}$ denotes a code image vector of instructions, E represents the entry point, and in denotes input data.

A register-based RAM maintains a state s, represented by the tuple $(\mathsf{pc}, \mathbf{R}, \mathcal{M}, \mathsf{halt})$, where pc denotes the program counter, referring the location of an instruction in the code image vector $\mathbf{I}$, $\mathbf{R}$ represents a set of registers that hold values, $\mathcal{M}$ denotes the memory state, and halt is a public bit indicating whether the machine has reached its termination. The RAM runtime simulates the semantics of each instruction and produces execution states. Each execution state s_i is generated from the previous state s_{i-1} by executing the related instruction. The execution of the RAM runtime is considered *valid* when each new state follows the expected result from the prior state, with all states except the last one remaining non-halted. A verifiable RAM consists of the following procedures:

- $\mathsf{Setup}(1^\lambda, \mathbf{I}, E) \rightarrow \mathsf{pp}$: Generates public parameters.
- $\mathsf{Compute}(\mathbf{I}, E, \mathsf{in}) \rightarrow \mathsf{out}, \mathbf{s}$: Computes output out with each machine state s_i by simulating the RAM runtime.
- $\mathsf{Prove}(\mathsf{pp}, \mathbf{I}, E, \mathsf{in}, \mathsf{out}, \mathbf{s}) \rightarrow \pi$: Generates an execution proof π; aborts if state transitions are invalid.
- $\mathsf{Verify}(\mathsf{pp}, \mathbf{I}, E, \mathsf{in}, \mathsf{out}, \pi) \rightarrow \{0, 1\}$: Verifies the proof π on whether state transitions are valid.

4 VDORAM: PUBLICLY VERIFIABLE DISTRIBUTED OBLIVIOUS RAM

We favor advocate for the development of VDORAM from a publicly verifiable MPC through leveraging multi-prover ZKPs, due to its capability to enable the construction of succinct proofs focused on only checking the integrity of RAM states, as opposed to validating the entire sequence of computations leading to those states. We present the overview of the VDORAM in Sec. 4.1, achieving publicly verifiable MPC with multi-prover ZKPs. Subsequently, in Sec. 4.2, we discuss our CompatCircuit framework. This framework provides an MPC-for-zkSNARK construction, facilitating the integration of commonly utilized MPC functionalities for a multi-prover vRAM. Following that, in Sec. 4.3, we explore a memory management scheme tailored for a multi-prover environment, making a trade-off between communication overhead in computation stage and the circuit size that determines the proof generation overhead. Finally, we outline our VDORAM protocol in Sec. 4.4 and provide analysis in Sec. 4.5.

4.1 Overview

The overall architecture of VDORAM is shown in Figure 1. A publicly provided program is represented as a code segment containing multiple instructions. Provers, who exclusively hold secret inputs, collaboratively execute the machine and produces a proof by executing instruction fetch, memory fetch, instruction execution, trace sort and verification circuits.

At a high level, we can implement VDORAM from adapting traditional single-prover vRAMs [13, 59], where only necessary verifications are involved to check the integrity of RAM states. This adaptation involves several modifications: migrating the plaintext front-end computation to an MPC-based oblivious computation, committing to hash digests of public values, and associating the results of oblivious computation with R1CS witnesses. The workflow is shown in Figure 2. During each iteration, the provers commence by covertly retrieving the instruction to be executed, based on a blind program counter. Concurrently, they discretely fetch the possible memory value that may be required during the instruction

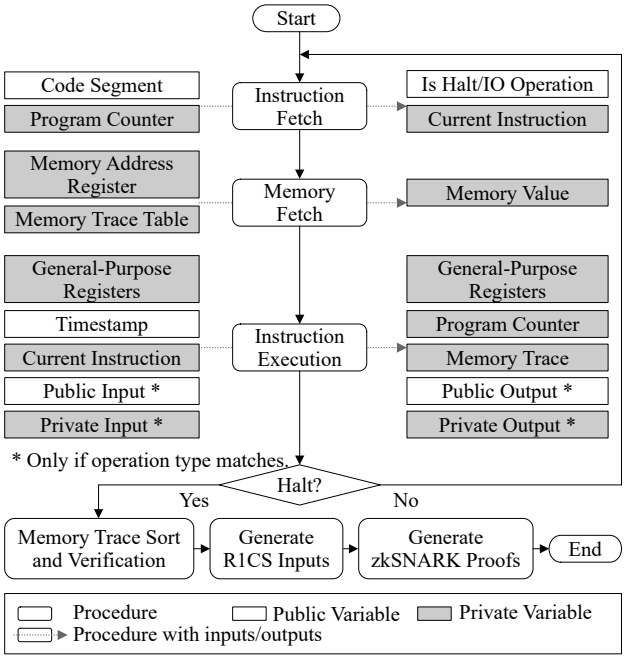

Figure 2: Workflow of VDORAM.

execution. If the instruction does not entail a memory operation, a dummy memory value is still fetched from the memory trace table. Following this, the provers execute the instruction by updating registers, interacting with I/O, and generating memory access operations – without knowing the specific nature of the instruction being executed. Even if there is no match, a dummy memory access is created and inserted into the memory trace table. These steps are repeated until the machine halts. Subsequently, the provers demonstrate the integrity of the memory accesses through a consistency check.

However, at a detailed level, we still face challenges, including the limited availability of sophisticated publicly verifiable MPC protocols, the lack of front-end implementations for multi-prover ZKPs, and the unsuitability or inefficiency of existing random access memory designs for computations requiring both distributed obliviousness and public verifiability. We will address these challenges in Sec. 4.2 and Sec. 4.3.

4.2 CompatCircuit: Developing Sophisticated Publicly Verifiable MPC Protocols with a Front-End for Multi-Prover ZKPs

We explore concrete methods for constructing MPC-based front-end implementations for multi-prover ZKPs. The key distinctions are summarized below.

Firstly, our implementation addresses all dependencies on plaintext access of values – a common assumption in single-prover ZKP front-ends – enabling computation functionalities to be obliviously carried out without information leakage during branching. For example, the implementation of an equality check in [19] necessitates the sole prover to conditionally assign varying values based on *knowing* the validity of the condition $a = b$. This requirement is substituted with a composition of oblivious subtraction and inversion-based zero test. Similarly, the inversion functionality also requires excluding branching, with both modifications to the MPC protocol and R1CS statements.

Secondly, we substitute MPC protocols that are unsuitable for generating R1CS witnesses with either a complete redesign or integration of other suitable protocols. For instance, an efficient MPC implementation of comparison [62] is not appropriate as a ZKP front-end because the simple output of a resultant bit indicating if one element is smaller than another is insufficient for constructing a valid statement that checks the correctness. This limitation is circumvented by a sequence of boolean operations that compare the decomposed bits of two elements, because the correctness of bit-decomposition can be expressed in R1CS statements. However, the popular bit-decomposition protocol implemented in MPyC [69] and MP-SPDZ [48] fails to meet ZKP requirements because it only extracts necessary bits and lacks a trivial extension to derive all bits from a field element. Consequently, we construct a fully bit-decomposition protocol improved from a prior MPC implementation [17], enhancing its efficiency.

Thirdly, we introduce a compatibility layer to unify separate codes for MPC and ZKP respectively. This layer provides a familiar development environment for single-prover ZKP applications, minimizing the complexities associated with underlying MPC. By unifying the implementations, the approach significantly lessens the likelihood of human errors, which are typically unavoidable when MPC and ZKP must be precisely aligned in extensive projects.

We propose our CompatCircuit as four primitives: field addition, multiplication, inversion-or-zero, and fully bit-decomposition. The protocols for inversion-or-zero, and fully bit-decomposition are shown in Figure 3. We then extend other commonly used functionalities provided in existing single-prover zkSNARK toolchains, particularly `circomlib` [40] and `jsnark_interface` [52], as the composition of the four primitives.

Inversion-or-zero: Inspired by [40], we introduce this primitive to facilitate an efficient construction of equality checks. In this protocol, each participant holds an arithmetic additive secret share $[a]$ of a field element $a \in \mathbb{F}_p$. The inversion-or-zero operation will produce $r \leftarrow 0$ if and only if $a = 0$. Otherwise, it outputs the inverse $r \leftarrow a^{-1}$, ensuring that $a \cdot a^{-1} = 1$. Importantly, information regarding whether $a = 0$ remains confidential. The typical efficient constant-round MPC construction, which depends on $a \neq 0$ and is facilitated by disclosing the product of a with a random multiplier, is unsuitable due to this confidentiality requirement. Instead, the MPC implementation harnesses Fermat's little theorem to compute the inverse, necessitating $\lceil \log_2 p \rceil$ multiplications. Additionally, adjustments are made to the R1CS statements. Ordinarily, one R1CS statement $a \cdot r = 1$ suffices, as it presupposes $a \neq 0$ in typical field inversions. However, to maintain correctness under our modified conditions, we employ four multiplications, ensuring the simultaneous validity of two statements: $a \cdot (a \cdot r - 1) = 0$ and $r \cdot (a \cdot r - 1) = 0$.

Fully bit-decomposition: We construct a fully bit-decomposition protocol that extracts all bits from a field element, building upon and enhancing the efficiency of a prior MPC implementation [17] since other constructions [48, 69] do not apply. Initially, each participant holds an arithmetic additive secret share $[a]$ of a field element

Protocol $\Pi_{\text{InvOrZero}}$

Input: A secret-shared field value $[a]$.

Protocol:

(1) Compute $r \leftarrow a^{p-2}$ with $\lceil \log_2 p \rceil$ multiplications.
(2) Compute witnesses for R1CS constraints $a \cdot (a \cdot r - 1) = 0$ and $r \cdot (a \cdot r - 1) = 0$.

Protocol Π_{FBits}

Input: A secret-shared field value $[a]$, l daBits, an edaBit $([b] \in \mathbb{F}_p, [b_0], [b_1], \ldots, [b_{l-1}] \in \{0,1\})$ providing secret shares of a random field value with its bits [22].

Protocol:

(1) Compute $[c] \leftarrow [a] - [b] \in \mathbb{F}_p$, and reveal $c \in \mathbb{F}_p$. Then, compute plaintext value $d \leftarrow c + 2^l - p \in \mathbb{Z}_{2^{l+1}}$, and perform plaintext bit-decomposition on d, getting bits $(d_0, d_1, \ldots, d_{l-1}, d_l)$.
(2) Use a bit adder with carry up to add bit sequence $([b_0], [b_1], \ldots, [b_{l-1}])$ with $(d_0, d_1, \ldots, d_{l-1}, d_l)$. Denote the result bit sequence as $([e_0], [e_1], \ldots, [e_l])$.
(3) Denote the most-significant bit $[e_l]$ as $[q] \in \{0,1\}$. Denote $(p_0, p_1, \ldots, p_{l-1})$ as the bit sequence of field prime $p \in \mathbb{Z}_{2^l}$. Multiply each public bit with $[1-q]$, getting $([f_0], [f_1], \ldots, [f_{l-1}])$.
(4) Use a bit adder with carry up to add bit sequence $([e_0], [e_1], \ldots, [e_{l-1}])$ with $([f_0], [f_1], \ldots, [f_{l-1}])$. Denote the results $([a_0, a_1, \ldots, a_{l-1}])$.
(5) For each boolean share $[a_i] \in \{0,1\} (i \in [0, l-1])$, convert it back to arithmetic share $[r_i] \in \mathbb{F}_p$ by consuming a daBit, using B2A_p protocol proposed in Prio+ [1].
(6) Compute witnesses for R1CS constraints $a = \sum_{i=0}^{l-1} r_i \cdot 2^i$ and $(r_0, r_1, \ldots, r_{l-1}) < (p_0, p_1, \ldots, p_{l-1})$.

Figure 3: Inverse-or-zero and fully bit-decomposition protocol in CompatCircuit.

$a \in \mathbb{F}_p$. This bit-decomposition operation results in $l = \lceil \log_2 p \rceil$ bits, denoted as r_i ($i \in [0, l-1]$), such that the equation $a = \sum_{i=0}^{l-1} r_i \cdot 2^i$ is satisfied. Subsequently, each participant receives an arithmetic additive secret share, denoted as $[r_i]$, for each bit b_i ($i \in [0, l-1]$). In the original implementation by [17], an additional bit $[q]$, indicating whether an overflow occurs, is computed from a separate comparison of two bit sequences, and p is subtracted only if $[q]$ is true. Our construction circumvents this comparison by consistently subtracting p during the first bit addition. Consequently, $[q]$ naturally emerges as the most significant bit of the result. The original conditional subtraction has been replaced by a conditional addition executed only when $[1-q]$ is true. This adjustment enhances the efficiency of the protocol.

We now construct other commonly used functionalities [40, 52] as the composition of the four primitives. Boolean logic and if-else selection can be directly represented using field addition and multiplication, zero test and equality check requires an inversion-or-zero operation, while less-than comparison require bit-decompositions.

- *Boolean logic.* Although operations should primarily handle field elements in $\mathbb{F}_p$ as inputs and outputs, it is also feasible to simulate boolean AND/OR/XOR/NOT operations with field additions and multiplications, assuming the input field element is in range $\{0,1\}$.
- *If-else selection.* This operation selects a value a if condition bit c is true, or b otherwise. This is usually implemented as $c \leftarrow a \cdot c + b \cdot (1-c)$.
- *Zero test.* This maps a field element to a boolean value indicative of whether it is zero. Given an element a, return $b = 1$ if $a = 0$, otherwise return 0. This can be implemented using the inversion-or-zero primitive: let inv represent the inversion-or-zero result of a, and then return $b \leftarrow 1 - a \cdot \text{inv}$.
- *Equality check.* To avoid conditionally branching introduced in [19], we migrate the implementation by [40], which begins by subtracting the two elements and subsequently zero-testing the difference.
- *Less-than comparison.* Given two elements a and b, return a bit c indicating whether $a < b$. It is constructed from initially performing fully bit-decomposition of the two elements and subsequently comparing them utilizing boolean operations. The bit sequences and intermediate results act as auxiliary witnesses to prove the correctness. Despite its higher complexity compared to a pure MPC implementation [62], the increased overhead is justified by the proof requirements.

With CompatCircuit, it becomes straightforward to construct the execution circuit for VDORAM. However, it remains to be determined whether the memory fetch, trace sort, and trace verification circuits can achieve efficient construction in obliviousness. This will be the focus of our investigation in the following subsection.

4.3 Memory Management: Balancing Oblivious Computation Overheads with Proof Generation Complexity

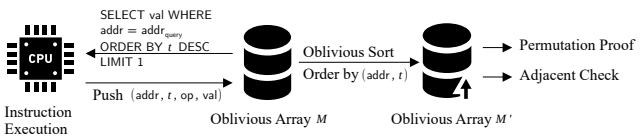

Figure 4: Memory management scheme of VDORAM.

As shown in Figure 4, we construct a multi-prover memory management scheme based on [26]. The main procedures are as below: (1) During each instruction execution, a memory access tuple $m = (\text{addr}, t, \text{op}, \text{val})$ is generated and saved to the oblivious array M, where t refers to a monotonically increasing timestamp and op indicates if the memory access is a load or a store instruction. Note that in our VDORAM, we also want to hide whether the instruction relates to memory access or not, and therefore, if not, we specify the memory address addr with the maximum value, $-1 \in \mathbb{F}_p$. (2) During each memory fetch, the stored memory value might be obliviously fetched with an address filter, and if there are multiple values, select the one with the largest t, which means the most recent value that corresponds to the address. (3) After the machine halts, the prover needs to prove the correctness of T memory accesses. First, sort the vector M' from M so that M' is ordered firstly by memory

address addr, then by time t. Then, a permutation check enforces that no values are altered during the sorting process. Following this, a memory consistency check scans the sorted vector, and for each pair of adjacent lines i and $i+1$, the following condition from [26] holds, ensuring that for each load operation, the value retrieved matches the most recently stored value at that address:

$$\begin{aligned}&((\mathsf{addr}_i < \mathsf{addr}_{i+1}) \vee ((\mathsf{addr}_i = \mathsf{addr}_{i+1}) \wedge (t_i < t_{i+1})))\\ \wedge&((\mathsf{addr}_i \neq \mathsf{addr}_{i+1}) \vee (\mathsf{val}_i = \mathsf{val}_{i+1}) \vee (\mathsf{op}_{i+1} = \mathtt{store}))\\ \wedge&((\mathsf{addr}_i = \mathsf{addr}_{i+1}) \vee (\mathsf{op}_{i+1} = \mathtt{store}))\end{aligned}$$

Now, we will discuss detailed construction in a multi-prover setting. By incorporating CompatCircuit, we can program this verification statement within CompatCircuit, facilitating the automatic construction of MPC protocol to compute and provide the necessary secret-shared auxiliary inputs for collaborative zkSNARKs. However, there are more considerations beyond a naive combination.

Oblivious array. Single-prover vRAMs employ a plaintext array to manage memory data. Transitioning to a multi-prover context, we can replace the plaintext array with an oblivious storage element similar to those utilized in MPC-based DORAM schemes. For instance, a 2-party Circuit ORAM [72] used by [47], or a 3-party random access memory [23].

In VDORAM, the oblivious storage is required to facilitate the confidential insertion or overwriting of new memory values at a specified address, retrieving the latest memory value corresponding to that address, and supporting memory consistency checks. We demonstrate the ideal functionality needed by our VDORAM in Figure 5, where the auxiliary data $\mathsf{aux} = (t, \mathsf{op})$. Note that, our VDORAM requires a unique $\mathcal{F}_{\text{HistoricalKV}}.\mathsf{Export}()$ functionality, which is not needed for a regular DORAM.

Functionality $\mathcal{F}_{\text{HistoricalKV}}$

Parameters. Memory size n that restricts the address $\mathsf{addr} \in [0, n-1]$.

Functionality. $\mathcal{F}_{\text{HistoricalKV}}.\mathsf{Add}(\mathsf{addr}, \mathsf{val}, \mathsf{aux})$: add or update memory value val associated with address addr.

$\mathcal{F}_{\text{HistoricalKV}}.\mathsf{Query}(\mathsf{addr})$: return the latest memory value val associated with address addr.

$\mathcal{F}_{\text{HistoricalKV}}.\mathsf{Export}()$: return an oblivious array containing all $(\mathsf{addr}, \mathsf{val}, \mathsf{aux})$ tuples, including those being overwritten.

Figure 5: Ideal functionality for the memory storage in VDORAM.

The implementation of $\mathcal{F}_{\text{HistoricalKV}}$ should maintain obliviousness, meaning provers must not acquire any supplementary information about any given addr. For instance, provers should even remain unaware of the number of times an *unknown* memory address has been accessed, as disclosing such information could compromise security. A conventional DORAM scheme commences by initializing the storage and then simulating memory accesses online, *overwriting* the storage if the address already contains a value. This approach of discarding outdated data is appropriate for traditional DORAM implementations, which do not require public verifiability since integrity can be evaluated based on the security guaranteed by maliciously secure MPC. To enable public verifiability, it is critical to retain all historical memory accesses from which the proof is derived.

One initial approach involves implementing $\mathcal{F}_{\text{HistoricalKV}}$ by adapting the oblivious storage employed in DORAM, along with an append-only oblivious array to store $(\mathsf{addr}, \mathsf{val}, \mathsf{aux})$ for each $\mathcal{F}_{\text{HistoricalKV}}.\mathsf{Add}()$ access. Nevertheless, despite the requirement for provers to allocate approximately double the space for two oblivious structures, existing efficient DORAM schemes support an insubstantial number of parties. This does not align with our VDORAM aims, which seek to accommodate an arbitrary number of provers beyond just two or three. Consequently, we have implemented this functionality in MPC using only one linearly-scanned oblivious array, with an additional requirement that the timestamp t contained in aux *must* exhibit a monotonically increasing pattern every time $\mathcal{F}_{\text{HistoricalKV}}.\mathsf{Add}()$ is invoked:

- Upon receiving $\mathcal{F}_{\text{HistoricalKV}}.\mathsf{Add}(\mathsf{addr}, \mathsf{val}, \mathsf{aux})$, insert the tuple to the end of array M.
- Upon receiving $\mathcal{F}_{\text{HistoricalKV}}.\mathsf{Query}(\mathsf{addr})$, for each memory tuple m_i in array M, compute $b_i \leftarrow \mathsf{EqualityCheck}(\mathsf{addr}, \mathsf{addr}_i) \in \{0, 1\}$. Then, set $b_i \leftarrow 0$ if there exists a larger $b_j = 1$ where $j > i$. Finally, return $\sum_{i=1}^{|M|} b_i \cdot \mathsf{val}_i$.
- Upon receiving $\mathcal{F}_{\text{HistoricalKV}}.\mathsf{Export}()$, return the array M.

Oblivious sort. Migrating vRAM into multi-prover setting, the vector M is no longer in plaintext. We must sort the vector in a manner that conceals its values, This is a problem well-studied in the literature [4] [44]. We opted for Bitonic mergesort; however, adapting this algorithm for VDORAM necessitates modifications. Initially, the scheme from [26] requires an ordered vector M' primarily by memory address addr, and secondly by time t. Typically, this is achieved by performing a stable sorting by time, followed by a second sorting by address. However, since Bitonic mergesort is inherently unstable, we introduce a complex comparer, slightly increasing overhead by adding an additional equality check: $m_1 < m_2 \iff (\mathsf{addr}_1 < \mathsf{addr}_2) \vee ((\mathsf{addr}_1 = \mathsf{addr}_2) \wedge (t_1 < t_2))$. Moreover, Bitonic mergesort requires the size of vector $|M|$ to be a power of 2. Hence, we must pad the vector before sorting and subsequently remove the padded elements post-sorting without plaintext access. We append the vector with padding item $m_{\mathsf{pad}} = (-1, -1, -1, -1)$, where $-1 \in \mathbb{F}_p$. These padding items, being greater than any real memory access tuples, allow provers to safely remove them from the end of the vector without exposing the actual contents.

Different consideration in memory check performance. In single-prover vRAM, memory check schemes such as in [19] are generally considered more efficient than [26], featuring reduced proof generation time. However, the efficiency and associated overhead of a scheme differ significantly within our VDORAM context as opposed to a single-prover vRAM. In a conventional single-prover scenario, the computation stage – where plaintext computations occur – is essentially negligible and considered almost *free* in comparison to the complexity of the circuit verifying the statements. In [26], the memory access vector M of size T is sorted, accompanied by a permutation proof and $T-1$ less-than checks on adjacent timestamp values. Conversely, [19] expands the vector to size $2T$, implementing a permutation proof and $T-1$ difference checks on

adjacent timestamp values. In the single-prover scenario, [19] offers enhanced efficiency over [26], as sorting the values in plaintext is inexpensive, irrespective of whether the vector size is T or $2T$. However, operations such as the heavyweight less-than or lightweight difference checks must be represented within an arithmetic circuit, and the size of this circuit constitutes the primary overhead in proof generation. In contrast, multi-prover scenarios replace all plaintext values with secret shares in MPC, making latency a crucial factor. Computation stage is no longer free in terms of cost: an MPC sorting network incurs substantial overhead due to the extensive communication required – we will further demonstrate in our evaluation, Sec. 5. Given these considerations, for our multi-prover implementation, we support the adoption of the memory checking protocol proposed in [26] in contrast to [19], due to its relative simplicity and reduced computational and communicational demands in multi-prover contexts.

4.4 Protocol Specification

Putting it all together, we construct our VDORAM, providing the full protocol in Figure 6. During each round, provers begin by retrieving the next instruction to be executed through an *instruction fetch* circuit. Except for the case of a halt or IO operation, provers get no extra information about which instruction will be executed next. Subsequently, regardless of whether the instruction involves a memory read/write operation, they fetch the memory value for a specified address from a blind memory trace table using a *memory fetch* circuit. Following this, the *instruction execution* circuit is utilized to interpret the execution by computing, updating registers, inserting a new row to the memory trace table, and interacting with public/private I/O if the instruction type matches. These three types of circuits are repeatedly executed upon fetching an instruction, continuing until the machine reaches a halt state. Upon completion, the *trace sort* circuit is run to blindly pad and sort the memory trace table. This is succeeded by the execution of a *trace verification* circuit, which checks the correctness of the entire memory management process.

Protocol $\Pi_{\mathbf{VDORAM}}$

Input: An instruction vector $\mathbf{I}$ with entrypoint E is publicly provided. m provers exclusively and privately holds N inputs in_i which corresponds to the i-th input the machine asks for.
Protocol:

(1) Trusted setup: the trusted initializer generates public parameters for collaborative zkSNARKs, as well as sufficient beaver triples, edaBit, and daBit for multi-party computation.
(2) Each prover construct and broadcast secret shares of his/her input $[\mathsf{in}_i]$ to one other provers.
(3) Set public timestamp $t \leftarrow 0$. Initialize all registers $[R_j] \leftarrow 0$. Set the program counter to the entrypoint $[\mathsf{pc}] \leftarrow E$. Initialize memory access vector $[\mathbf{M}] \leftarrow ()$.
(4) Provers run *instruction fetch* protocol. Given $\mathbf{I}$ and $[\mathsf{pc}]$ as input, provers fetches the next instruction $[I]$ as well as the instruction type indicator $\mathsf{op}_{\mathsf{masked}} \in \{\mathtt{input}, \mathtt{output}, \mathtt{secret}\}$. This protocol does not reveal information about an non-IO instruction.
(5) If $\mathsf{op}_{\mathsf{masked}} = \mathtt{input}$, provers assign the input value $[\mathsf{in}] \leftarrow [\mathsf{in}_i]$ where i denotes the count of previously provided inputs. Otherwise, provers publicly set it to 0.
(6) If $t \geq 1$, provers run *memory fetch* protocol. Given $[I]$, $[R_{\mathsf{addr}}]$, and $[\mathbf{M}]$, provers fetches the latest memory value $[\mathsf{val}]$. Private output $[\mathsf{val}]$ would be a secret-shared 0 if the instruction is not related with memory access or the address has never been stored with a value. In particular, if $t = 0$, provers publicly sets $[\mathsf{val}] \leftarrow 0$.
(7) Provers run *instruction execution* protocol. Given $[I]$, $[\mathsf{val}]$, t, $[\mathsf{pc}]$, $[\mathsf{in}]$, and all registers $[R_j]$, the protocol returns updated register values as $[\mathsf{pc}]'$ and $[R_j]'$. A memory access trace $[m] = ([\mathsf{addr}], t, [\mathsf{op}], [\mathsf{val}])$ is also produced. The protocol also returns the output value $[\mathsf{out}]$ if the operation type matches; otherwise, it's 0. Provers accordingly updates values and insert $[m]$ to vector $[\mathbf{M}]$.
(8) If $\mathsf{op}_{\mathsf{masked}} \neq \mathtt{halt}$, increase $t \leftarrow t + 1$ and go to the *instruction fetch* protocol.
(9) Provers run *trace sort* protocol to pad and sort the memory access vector $[M]$. Get sorted vector $[M]'$.
(10) Provers run *trace verification* protocol with input $[M]$ and $[M]'$. The verification result is publicly revealed.
(11) For all *instruction fetch*, *instruction execution*, and the final *trace verification* protocol, provers invoke collaborative zkSNARKs, providing all inputs, outputs, intermediate results, and auxiliary data with necessary hash digests, and getting an non-interactive succinct proof π.

Figure 6: VDORAM protocol.

4.5 Analysis

4.5.1 Communication overhead of CompatCircuit primitives. Let $l = \lceil \log_2 p \rceil$ denote the number of bits in the finite field $\mathbb{F}_p$. An addition operation does not require communication, while a multiplication operation requires broadcasting one revealed element, thus incurring 1 round of communication. The inversion-or-zero operation necessitates $l+4$ rounds of multiplications: l multiplications to secretly obtain the inverse, and an additional 4 multiplications for constructing the verification statements. A fully bit-decomposition operation incurs $19l + 2$ rounds of communications:

- 1 round for exposing c.
- $3l$ multiplications for an l-bit MPC adder with a plaintext operand, where each adder requires at most $3l$ multiplications.
- $6l$ multiplications: l for multiplying each bit p_i with $[1-q]$, $5l$ multiplications for an l-bit MPC adder with a secret operand.
- 1 round: exposing l boolean values for $\mathrm{B2A}_p$ protocol [1].
- $10l$ multiplications for computing R1CS witnesses: l for input range check, l for constructing the field element from input bits, $8l$ for ensuring the bits are smaller than the bit decomposition of field modulus p.

Definition 4.1. An (m, t) VDORAM with m provers $\mathbf{P} = \mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{m-1}$ holding public input $\mathsf{in}_{\mathsf{pub}}$ and secret shares of private input $\mathsf{in}_{\mathsf{priv}}$ is a RAM $(\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}})$ with the following procedures:

- $\mathsf{Setup}(1^\lambda, \mathbf{I}, E) \to \mathsf{pp}$: Generates zkSNARK parameters.
- $\mathsf{Compute}(\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}}) \to \mathsf{out}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s}$: Computes $\mathsf{out}_{\mathsf{pub}}$, $\mathsf{out}_{\mathsf{priv}}$ with each machine state s_i by simulating the RAM runtime.
- $\mathsf{Prove}(\mathsf{pp}, \mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s}) \to \pi$: Generates an execution proof π; aborts if state transitions are invalid.
- $\mathsf{Verify}(\mathsf{pp}, \mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{pub}}, \pi) \to \{0, 1\}$: Verifies the proof π on whether state transitions are valid.

and with the following properties:

- *Completeness:* For all $(\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}})$, the following statement holds:

$$\Pr\left[\mathsf{Verify}^H(\mathsf{pp}, \mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{pub}}, \pi) = 0 \;\middle|\; \begin{array}{l} H \leftarrow \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathbf{I}, E) \\ \mathsf{out}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s} \leftarrow \mathsf{Compute}^H(\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}}) \\ \pi \leftarrow \mathsf{Prove}^H(\mathsf{pp}, \mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

- *Knowledge soundness:* For all $(\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}})$ and for all sets of efficient algorithms $\mathbf{P} = \mathcal{P}_0^*, \mathcal{P}_1^*, \ldots, \mathcal{P}_{m-1}^*$, there exists an efficient extractor $\mathsf{Ext}^{H,\mathbf{P}^H}$ such that:

$$\Pr\left[\begin{array}{l}(\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \\ \mathsf{out}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}}, \\ \mathsf{out}_{\mathsf{priv}}, \mathbf{s}) \in R\end{array} \;\middle|\; \begin{array}{l} H \leftarrow \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathbf{I}, E) \\ \mathsf{in}_{\mathsf{priv}}, \mathsf{out}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s} \leftarrow \mathsf{Ext}^{H,\mathbf{P}^H}(\mathsf{pp}, \mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}) \end{array}\right] \geq \Pr\left[\begin{array}{l}\mathsf{Verify}^H(\mathsf{pp}, \mathbf{I}, E, \\ \mathsf{in}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{pub}}, \pi) = 1\end{array} \;\middle|\; \begin{array}{l} H \leftarrow \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathbf{I}, E) \\ \mathsf{out}_{\mathsf{pub}}, \pi \leftarrow \mathbf{P}^H(\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}) \end{array}\right] - \mathsf{negl}(\lambda)$$

  R denotes the collection of valid vRAM executions. $\mathsf{Ext}^{H,\mathbf{P}^H}$ denotes Ext has oracle access to H and may re-run the provers P for multiple times with H re-programmed.
- *Succinctness:* Proof size and verification time are $O(|\mathbf{s}|)$.
- *t-zero-knowledge:* For all efficient $\mathcal{A}$ controlling $k \leq t$ provers $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{k-1}$, there exists an efficient simulator Sim such that for all $(\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}})$ and for all efficient distinguishers D, $|D_0 - D_1| \leq \mathsf{negl}(\lambda)$ holds, where:

$$D_0 = \Pr\left[D^{H[\mu]}(\mathsf{tr}) = 1 \;\middle|\; \begin{array}{l} H \leftarrow \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathbf{I}, E) \\ b \in \{0,1\} = 1 \iff (\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s}) \in R \\ (\mathsf{tr}, \mu) \leftarrow \mathsf{Sim}^H(\mathsf{pp}, \mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{pub}}, (\mathsf{in}_{\mathsf{priv}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s})_{0,1,\ldots,k-1}, b) \end{array}\right]$$

$$D_1 = \Pr\left[D^H(\mathsf{tr}) = 1 \;\middle|\; \begin{array}{l} H \leftarrow \mathcal{U}(\lambda) \\ \mathsf{pp} \leftarrow \mathsf{Setup}^H(1^\lambda, \mathbf{I}, E) \\ \mathsf{tr} \leftarrow \mathsf{View}_{\mathcal{A}}^H(\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}}, \mathsf{out}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s}) \end{array}\right]$$

  tr is a transcript, $\mathsf{View}_{\mathcal{A}}^H$ denotes view of $\mathcal{A}$ when provers $\mathcal{P}$ interact with $\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}}, \mathsf{out}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s}$ in Compute and Prove procedures, μ is a partial function from H such that given an input x, function $H[\mu]$ equals $\mu(x)$ if x is in the domain of μ, otherwise equals $H(x)$.
- *t-Distributed-Obliviousness*: Conditions are same as *t-Zero-Knowledge*, except that $\mathsf{out}_{\mathsf{priv}}$ is visible among provers. So the definition is slightly different: update $\mathsf{out}_{\mathsf{pub}} \leftarrow \mathsf{out}_{\mathsf{pub}} \cup \mathsf{out}_{\mathsf{priv}}$ and $\mathsf{out}_{\mathsf{priv}} \leftarrow \emptyset$.

4.5.2 Communication overhead in the VDORAM protocol. In this analysis, we consider l, the number of bits in the finite field to be a small constant. We additionally denote T as the total number of iterations, N as the number of inputs, and I as the length of the instruction vector. It's worth noting that our VDORAM protocol employs a read-write memory model with a full address space, implying that the memory capacity is virtually unlimited (with a capacity of p). Therefore, the overhead is not affected by the size of the memory. The VDORAM protocol incurs the following communication overheads:

- $m \cdot N$ broadcasts are required for distributing the secret shares of the inputs.
- $O(T \cdot I)$ rounds of communication occur during T invocations of the instruction fetch protocol. During each fetch, a linear scan is conducted to determine the next instruction to be executed, taking $O(I)$ time.
- $O(T^2)$ rounds of communication are required for T instances of the memory fetch protocol. During each fetch, a linear scan is carried out to identify the most recent memory value to be read, taking $O(T)$ time.
- $O(T)$ rounds of communication take place for T instances of the instruction execution protocol. The overhead from the instruction execution circuit can be considered a constant

number of CompatCircuit primitives, as the number of registers and instruction types are small constants..
- $O(T \log^2 T)$ rounds are needed for the memory trace sorting protocol. The sorting algorithm requires $O(T \log^2 T)$ comparisons, each necessitating a small constant number of bit-decomposition primitives.
- $O(T)$ rounds are required for the memory trace verification protocol. This protocol checks each pair of adjacent lines, which also involves comparisons based on bit-decomposition.

4.5.3 Privacy Disclosure. Our protocol Π_{VDORAM} is designed to maintain the confidentiality of sensitive information against both provers and verifiers. This includes:

- Register values: The contents of the registers are kept private, including the program counter pc.
- Memory accesses: Information about memory value, address, and access type is concealed. The number of memory accesses equals to the total number of instruction cycles.
- Instructions: The specific instruction being executed at any given time remains hidden, with some I/O exceptions.

However, Π_{VDORAM} also requires some information to be publicly available. (1) I/O instruction type. Only `input`, `output`, and `halt` instructions are revealed to provers. Provers maintain the protocol, and thereby need to know when to provide input, when to expect output, and when the machine has completed its execution. *Mitigation:* For non-interactive programs, we can structure the program into three distinct phases: input, computation, and output. During the input and output phases, data are simply transferred to and from registers/memory without any actual processing. This minimizes the risk of sensitive information being leaked through the public disclosure of I/O-related instruction types. (2) Total number of iterations (T). Like most oblivious RAMs, the total number of instructions executed by the machine is made public. This is necessary to halt the machine and facilitate the proof generation process. *Mitigation:* Programmers should design their code to minimize the impact of revealing the total iteration count. If the iteration count is directly related to a confidential value, introduce a dummy loop with a randomly determined number of iterations. If certain computational steps are optional based on the value of a confidential variable, include dummy loops to ensure a consistent iteration count regardless of the condition.

Now, we formally define VDORAM as Definition 4.1 based on [64] and [12]. Then, we analyze security properties on our proposed VDORAM using techniques from [64].

Theorem 4.2. *If* (Setup, Prove, Verify) *is an* (m, t) *collaborative zkSNARK, and* $\mathsf{Compute}_{\mathsf{MPC}}$ *is an MPC protocol for* Compute *that is secure-with-abort against up to* t *corruptions, then a RAM* $(\mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{in}_{\mathsf{priv}})$ *with procedures* (Setup, $\mathsf{Compute}_{\mathsf{MPC}}$, Prove, Verify) *is an* (m, t) *VDORAM.*

Proof sketch. The completeness holds from the completeness of the underlying collaborative zkSNARK (Setup, Prove, Verify) and the correctness of MPC protocol $\mathsf{Compute}_{\mathsf{MPC}}$. The knowledge soundness holds from collaborative zkSNARK: even if more than t provers collude by producing wrong output from $\mathsf{Compute}_{\mathsf{MPC}}$, the knowledge soundness prevents a forged proof being verified. The succinctness also holds from collaborative zkSNARK.

t-zero-knowledge and t-distributed-obliviousness follow from t-zero-knowledge of collaborative zkSNARK and distributed obliviousness of MPC protocol: when $b = 1$, the t-zero-knowledge implies that π can be simulated from $\mathsf{pp}, \mathbf{I}, E, \mathsf{in}_{\mathsf{pub}}, \mathsf{out}_{\mathsf{pub}}, (\mathsf{in}_{\mathsf{priv}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s})_{0,1,\ldots,k-1}$. Since the security of secret sharing utlized in MPC protocol $\mathsf{Compute}_{\mathsf{MPC}}$ holds against up to t corruptions and $k \leq t$, adversary $\mathcal{A}$ get zero-knowledge from $(\mathsf{in}_{\mathsf{priv}}, \mathsf{out}_{\mathsf{priv}}, \mathbf{s})_{0,1,\ldots,k-1}$, therefore t-zero-knowledge holds in this case. Otherwise, $b = 0$, the security of MPC protocol $\mathsf{Compute}_{\mathsf{MPC}}$ implies that the view of adversary $\mathcal{A}$ can be directly simulated from the witnesses of corrupted provers. Thus, t-zero-knowledge and t-distributed-obliviousness properties hold. ■

5 IMPLEMENTATION AND EVALUATION

In this section, we present the implementation of our proposed VDORAM along with the underlying CompatCircuit. We also design experiments to evaluate the various factors that influence performance.

Implementation. Our implementation consists of approximately 15,000 lines of C# code and additional components. Specifically, it includes the CompatCircuit library for unifying multi-party computation and R1CS verification with an MPC implementation (≈ 8,500 lines), the VDORAM program (≈ 3,500 lines), unit tests (≈ 2,500 lines), evaluation programs and scripts (≈ 3,500 lines), and modifications to the Rust program `collaborative-zksnark` [64] (≈ 300 lines). Our source code is available at https://github.com/CompatCircuit/vdoram-artifacts.

Evaluation Setup. Our experiments were conducted on a server instance on ESXi 8.0 virtualization platform with various hardware allocation. Denote the prover count as m, the server instance simulating multiple parties was allocated with up to $4m$ vCPU cores from Intel Xeon 4214R @ 2.4 GHz, $16m$ GB of RAM, and $20m$ GB of disk space, running Debian Linux 12 as the operating system. Note that our implementation is *not* parallelized, but more than one CPU core is needed to handle the communication among other provers. The elliptic curve used in our experiment is BLS12-377, and the collaborative zkSNARKs variant employed is Plonk [28]. Each experiment is repeated 3 times.

The variables altered in our experiments included: the number of MPC parties (2, 4, 8, and optionally 16), RAM program instruction types (multiplication, comparison, hashing, memory store, and memory load), and the number of instruction cycles (4, 16, 20, 32, 50, and 64). We recorded time costs throughout the entire process, including the phases of MPC preprocessing, computation, ZKP setup, proof generation, and verification associated with executing VDORAM.

We first evaluated the performance overhead in CompatCircuit, as shown in Figure 7. When compared with the single-prover baseline, the computation time costs for a 2-prover configuration increased by up to 20 times. This rise can be attributed to the transition from single-party to multi-party computation, which brings additional complexities such as sharing revealing in Beaver multiplications and the implementation of the bit-decomposition protocol. The overhead is indispensable as parties in a multi-prover setting cannot compute in the same manner as a single-prover scenario. Subsequently, the increase in computation time becomes less steep

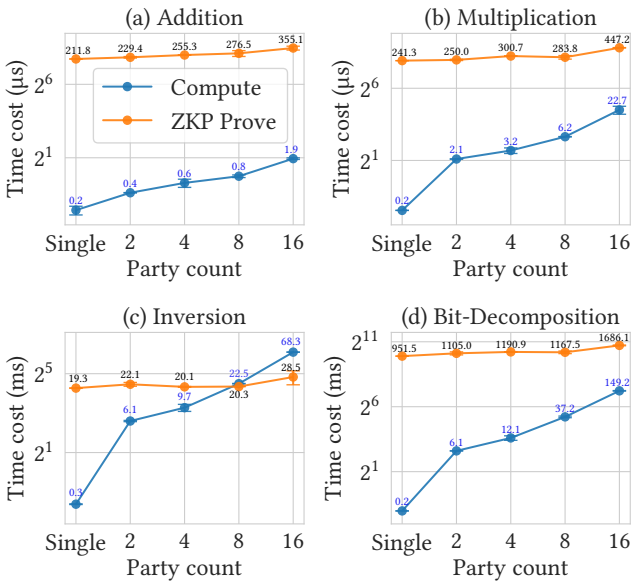

Figure 7: Performance overhead of CompatCircuit primitives.

as the number of parties grows. For instance, with an 8-prover setup, our system is capable of processing 1,250,000 additions, 150,000 multiplications, 45 inversions, or 25 bit-decompositions per second, which we consider to be a reasonable and acceptable performance outcome.

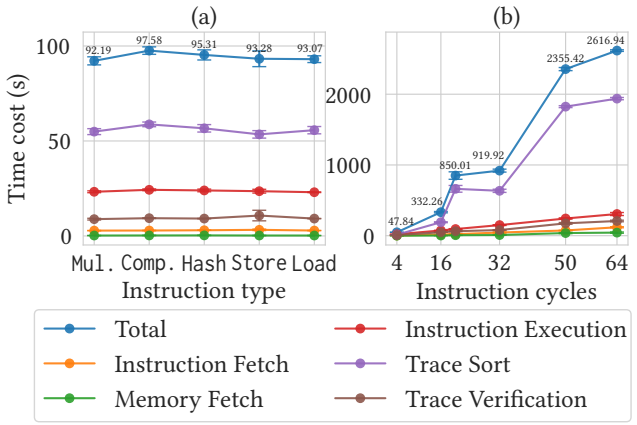

Figure 8: Computation time costs of VDORAM (8 parties) with (a) varying instruction types and (b) varying instruction cycles. Time costs remain constant with different types but increase with instruction cycles.

We subsequently assessed the performance of the VDORAM within an 8-prover configuration, as shown in Figure 8. We varied both the types and counts of instructions. In Figure 8(a), we set the instruction cycle count to 5, with varying instruction types. We observed that the execution time remains constant, regardless of the instruction type. This observation can be explained by the privacy demand, where everything *might* happen *must* happen, to prevent any chance of information leakage through the operation types used.

In Figure 8(b), the instruction cycle count varied from 4 to 64. We observed that the time costs associated with instruction fetch, memory fetch, instruction execution, and trace verification circuits are roughly constant per instruction (i.e., linear to the instruction cycle count). However, the time cost for the trace sort circuit increases significantly when the instruction cycle count reaches a power of two. This increase is expected because, in MPC, we used the Bionic mergesort algorithm which has an $O(T \log^2 T)$ time complexity in sequential computation and requires padding the items to a power of two. The trace sort circuit consumes a considerable amount of time in the computation stage. Fortunately, this circuit acts as an MPC-only CompatCircuit which does not require verification and can be further optimized by paralleling the computation.

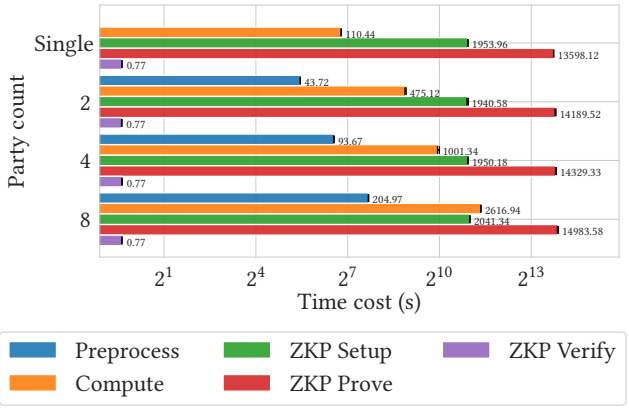

Figure 9: Time costs of overall procedures in VDORAM (instruction cycles: $T = 64$).

Lastly, we present the overall performance results across varying prover counts, using a single-prover setup as the baseline. The time costs are shown in Figure 9. We also estimated the bandwidth requirements during the computation and collaborative ZKP process: for communicating with each of the other $m - 1$ parties, the highest average bandwidth for an individual prover was 48.63 Mbps. Although the computation time increases more rapidly than the zero-knowledge proof generation time when the prover count increases – computation stage demands more frequent communication, our implementation still maintains a relatively reasonable overheads in total, compared with the single-prover baseline.

6 CONCLUSION AND FUTURE WORK

In this research, we have introduced CompatCircuit, a novel multi-prover ZKP front-end system. CompatCircuit combines collaborative zkSNARKs with a dishonest-majority MPC framework, facilitating the creation of multi-prover ZKP applications. Building upon CompatCircuit, we have presented VDORAM, the first publicly verifiable distributed oblivious RAM. By integrating distributed oblivious architectures with verifiable RAM, VDORAM achieves

an efficient RAM design that optimizes communication overhead and proof generation time. We have developed implementations of both CompatCircuit and VDORAM. Our performance evaluation result shows that the overall overhead remains relatively moderate with distributed obliviousness added. Future research directions include investigating the potential performance benefits of migrating lookup arguments [3, 27] to a multi-prover setting, optimizing the complexity of oblivious HistoricalKV functionality, and improving the memory management scheme to minimize the overhead in oblivious sort.

REFERENCES

[1] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. 2022. Prio+: Privacy preserving aggregate statistics via boolean shares. In *International Conference on Security and Cryptography for Networks*. Springer, 516–539. https://link.springer.com/chapter/10.1007/978-3-031-14791-3_23.

[2] Mohammed Alghazwi, Tariq Bontekoe, Leon Visscher, and Fatih Turkmen. 2024. Collaborative CP-NIZKs: Modular, Composable Proofs for Distributed Secrets. *arXiv preprint arXiv:2407.19212* (2024).

[3] Arasu Arun, Srinath Setty, and Justin Thaler. 2024. Jolt: Snarks for virtual machines via lookups. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 3–33.

[4] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 307–314.

[5] Carsten Baum, Ivan Damgård, and Claudio Orlandi. 2014. Publicly auditable secure multi-party computation. In *Security and Cryptography for Networks: 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings 9*. Springer, 175–196.

[6] Carsten Baum, Alex J Malozemoff, Marc B Rosen, and Peter Scholl. 2021. Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part IV 41*. Springer, 92–122.

[7] Carsten Baum, Emmanuela Orsini, and Peter Scholl. 2016. Efficient secure multiparty computation with identifiable abort. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part I 14*. Springer, 461–490.

[8] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. 2020. Efficient constant-round MPC with identifiable abort and public verifiability. In *Annual International Cryptology Conference*. Springer, 562–592.

[9] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. 2022. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing* 20, 6 (2022), 4733–4751. https://ieeexplore.ieee.org/abstract/document/10002421.

[10] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 2019. *Completeness theorems for non-cryptographic fault-tolerant distributed computation*. Association for Computing Machinery, New York, NY, USA, 351–371. https://doi.org/10.1145/3335741.3335756 https://dl.acm.org/doi/abs/10.1145/3335741.3335756.

[11] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference*. Springer, 90–108.

[12] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. 2016. Interactive oracle proofs. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part II 14*. Springer, 31–60. https://link.springer.com/chapter/10.1007/978-3-662-53644-5_2.

[13] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*. 781–796. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson.

[14] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. 2016. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 327–357. https://link.springer.com/chapter/10.1007/978-3-662-49896-5_12.

[15] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*. Springer, 738–768. https://link.springer.com/chapter/10.1007/978-3-030-45721-1_26.

[16] Hongrui Cui, Kaiyi Zhang, Yu Chen, Zhen Liu, and Yu Yu. 2021. MPC-in-Multi-Heads: A Multi-Prover Zero-Knowledge Proof System: (or: How to Jointly Prove Any NP Statements in ZK). In *European Symposium on Research in Computer Security*. Springer, 332–351.

[17] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. 2006. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*. Springer, 285–304.

[18] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*. Springer, 643–662. https://link.springer.com/chapter/10.1007/978-3-642-32009-5_38.

[19] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. 2022. Efficient proof of RAM programs from any public-coin zero-knowledge system. In *International Conference on Security and Cryptography for Networks*. Springer, 615–638.

[20] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. 2020. Line-point zero knowledge and its applications. *Cryptology ePrint Archive* (2020).

[21] Moumita Dutta, Chaya Ganesh, Sikhar Patranabis, Shubh Prakash, and Nitin Singh. 2024. Batching-efficient ram using updatable lookup arguments. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 4077–4091.

[22] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved primitives for MPC over mixed arithmetic-binary circuits. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II 40*. Springer, 823–852. https://link.springer.com/chapter/10.1007/978-3-030-56880-1_29.

[23] Brett Falk, Daniel Noble, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. 2023. DORAM revisited: maliciously secure RAM-MPC with logarithmic overhead. In *Theory of Cryptography Conference*. Springer, 441–470.

[24] Amos Fiat and Adi Shamir. 1986. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*. Springer, 186–194.

[25] Scroll Foundation. [n. d.]. Scroll – Native zkEVM Layer 2 for ethereum. https://scroll.io/ Accessed: Jul 15, 2024.

[26] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. 2021. Constant-overhead zero-knowledge for RAM programs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 178–191.

[27] Ariel Gabizon and Zachary J Williamson. 2020. plookup: A simplified polynomial protocol for lookup tables. *Cryptology ePrint Archive* (2020). https://eprint.iacr.org/2020/315.

[28] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. 2019. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive* (2019). https://eprint.iacr.org/2019/953.

[29] Joshua Gancher, Adam Groce, and Alex Ledger. 2017. Externally verifiable oblivious ram. *Proceedings on Privacy Enhancing Technologies* (2017).

[30] Sinka Gao, Guoqiang Li, and Hongfei Fu. 2024. ZKWASM: A ZKSNARK WASM Emulator. *IEEE Transactions on Services Computing* (2024). https://ieeexplore.ieee.org/abstract/document/10587123.

[31] Aarushi Goel, Mathias Hall-Andersen, and Gabriel Kaptchuk. 2023. Dora: A Simple Approach to Zero-Knowledge for RAM Programs. *Cryptology ePrint Archive* (2023).

[32] Aarushi Goel, Mathias Hall-Andersen, Gabriel Kaptchuk, and Nicholas Spooner. 2023. Speed-stacking: fast sublinear zero-knowledge proofs for disjunctions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 347–378.

[33] Lior Goldberg, Shahar Papini, and Michael Riabzev. 2021. Cairo–a Turing-complete STARK-friendly CPU architecture. *Cryptology ePrint Archive* (2021).

[34] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 305–326. https://link.springer.com/chapter/10.1007/978-3-662-49896-5_11.

[35] Ariel Hamlin, Justin Holmgren, Mor Weiss, and Daniel Wichs. 2019. Fully Homomorphic Encryption for RAMs. *Cryptology ePrint Archive* (2019).

[36] Ariel Hamlin, Justin Holmgren, Mor Weiss, and Daniel Wichs. 2019. On the plausibility of fully homomorphic encryption for RAMs. In *Annual International Cryptology Conference*. Springer, 589–619.

[37] David Heath and Vladimir Kolesnikov. 2020. A 2.1 KHz zero-knowledge processor with BubbleRAM. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2055–2074.

[38] David Heath and Vladimir Kolesnikov. 2021. PrORAM: Fast O (log n) Authenticated Shares ZK ORAM. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 495–525.

[39] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. 2021. Zero knowledge for everything and everyone: Fast ZK processor with cached ORAM for ANSI C programs. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1538–1556.

[40] iden3. [n. d.]. circomlib: Library of basic circuits for circom. https://github.com/iden3/circomlib Accessed: Aug 6, 2024.

[41] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2007. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 21–30.

[42] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 955–966.

[43] Keyu Ji, Bingsheng Zhang, Tianpei Lu, and Kui Ren. 2023. Multi-party private function evaluation for RAM. *IEEE Transactions on Information Forensics and Security* 18 (2023), 1252–1267.

[44] Kristján Valur Jönsson, Gunnar Kreitz, and Misbah Uddin. 2011. Secure multi-party sorting and applications. *Cryptology ePrint Archive* (2011).

[45] Yael Kalai and Omer Paneth. 2016. Delegating RAM computations. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part II 14*. Springer, 91–118.

[46] Sanket Kanjalkar, Ye Zhang, Shreyas Gandlur, and Andrew Miller. 2021. Publicly Auditable MPC-as-a-Service with succinct verification and universal setup. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 386–411.

[47] Marcel Keller. 2015. The oblivious machine-or: how to put the C into MPC. *Cryptology ePrint Archive* (2015).

[48] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590. https://dl.acm.org/doi/abs/10.1145/3372297.3417872.

[49] Marcel Keller and Peter Scholl. 2014. Efficient, oblivious data structures for MPC. In *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*. Springer, 506–525.

[50] Marcel Keller and Avishay Yanai. 2018. Efficient maliciously secure multiparty computation for RAM. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 91–124.

[51] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 839–858.

[52] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. 2018. xJsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 944–961. https://ieeexplore.ieee.org/abstract/document/8418647.

[53] Abhiram Kothapalli and Srinath Setty. 2022. SuperNova: Proving universal machine executions without universal circuits. *Cryptology ePrint Archive* (2022).

[54] Abhiram Kothapalli and Srinath Setty. 2024. HyperNova: Recursive arguments for customizable constraint systems. In *Annual International Cryptology Conference*. Springer, 345–379.

[55] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*. Springer, 359–388.

[56] Eyal Kushilevitz and Tamer Mour. 2019. Sub-logarithmic distributed oblivious RAM with small block size. In *IACR International Workshop on Public Key Cryptography*. Springer, 3–33.

[57] Matter Labs. 2020. ZKsync. https://zksync.io/ Accessed: Jul 15, 2024.

[58] Polygon Labs. n.d.. Polygon Miden | A rollup for high-throughput, private applications. https://polygon.technology/polygon-miden Accessed: Jul 15, 2024.

[59] Polygon Labs. n.d.. Polygon zkEVM | Scaling for the Ethereum Virtual Machine. https://polygon.technology/polygon-zkevm Accessed: Jul 15, 2024.

[60] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. 2024. Ceno: Non-uniform, Segment and Parallel Zero-knowledge Virtual Machine. *Cryptology ePrint Archive* (2024). https://eprint.iacr.org/2024/387.

[61] Steve Lu and Rafail Ostrovsky. 2013. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography Conference*. Springer, 377–396.

[62] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. 2021. Rabbit: Efficient comparison for secure multi-party computation. In *International Conference on Financial Cryptography and Data Security*. Springer, 249–270. https://link.springer.com/chapter/10.1007/978-3-662-64322-8_12.

[63] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 2019. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2111–2128. https://dl.acm.org/doi/abs/10.1145/3319535.3339817.

[64] Alex Ozdemir and Dan Boneh. 2022. Experimenting with collaborative zk-SNARKs: Zero-Knowledge proofs for distributed secrets. In *31st USENIX Security Symposium (USENIX Security 22)*. 4291–4308. https://www.usenix.org/conference/usenixsecurity22/presentation/ozdemir.

[65] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. 238–252.

[66] Huayi Qi, Minghui Xu, Dongxiao Yu, and Xiuzhen Cheng. 2024. SoK: Privacy-preserving smart contract. *High-Confidence Computing* 4, 1 (2024), 100183. https://www.sciencedirect.com/science/article/pii/S2667295223000818.

[67] Marc Rivinius, Pascal Reisert, Daniel Rausch, and Ralf Küsters. 2022. Publicly accountable robust multi-party computation. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2430–2449.

[68] Dragos Rotaru and Tim Wood. 2019. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *International Conference on Cryptology in India*. Springer, 227–249. https://link.springer.com/chapter/10.1007/978-3-030-35423-7_12.

[69] Berry Schoenmakers. 2018. MPyC—Python package for secure multiparty computation. In *Workshop on the Theory and Practice of MPC. https://github.com/lschoe/mpyc*.

[70] Berry Schoenmakers and Meilof Veeningen. 2015. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In *Applied Cryptography and Network Security: 13th International Conference, ACNS 2015, New York, NY, USA, June 2-5, 2015, Revised Selected Papers 13*. Springer, 3–22.

[71] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. 2016. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In *Applied Cryptography and Network Security: 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings 14*. Springer, 346–366.

[72] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 850–861.

[73] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2986–3001.

[74] Yibin Yang and David Heath. 2024. Two Shuffles Make a {RAM}: Improved Constant Overhead Zero Knowledge {RAM}. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1435–1452.

[75] RISC Zero. [n. d.]. RISC Zero | Universal Zero Knowledge. https://www.risczero.com/ Accessed: Aug 29, 2024.

[76] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. vRAM: Faster verifiable RAM with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 908–925.