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

Huayi Qi*†, Minghui Xu*✉, Xiaohua Jia‡, and Xiuzhen Cheng*
*School of Computer Science and Technology, Shandong University, Qingdao, Shandong, China
†Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China
‡Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong SAR, China
Email: qi@huayi.email, mhxu@sdu.edu.cn, csjia@cityu.edu.hk, xzcheng@sdu.edu.cn
✉ Corresponding author

***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, because existing solutions struggle with a paradigm mismatch between MPC and ZKP that limits the development of practical multi-prover ZKP front-ends. This gap arises because MPC protocols are optimized for minimal computation, whereas ZKPs require a complete trace for proving. Furthermore, adapting RAM designs is also challenging, as vRAMs are not built for the high costs of oblivious execution and existing DORAMs lack public verifiability.**

To address these challenges, we introduce CompatCircuit, the first multi-prover ZKP front-end implementation to our knowledge, designed to bridge this gap. CompatCircuit integrates collaborative zkSNARKs with novel MPC protocols, unifying computation and verification into a single compatible circuit paradigm. Building upon CompatCircuit, we present VDORAM, the first publicly verifiable distributed oblivious RAM. VDORAM reconciles the high communication latency of online MPC with the complexity of offline proof generation, resulting in a RAM design that balances these competing demands. We have implemented CompatCircuit and VDORAM in approximately 15,000 lines of code, demonstrating their practical feasibility through extensive experiments, including micro-benchmarks, comparative analysis, and program examples.

I. Introduction

Random access machines (RAMs) serve as a crucial model for expressing complex computational logic, especially in the field of secure computation. A RAM executes a program on its processor (register-based [12], [4], [78], [14], [52], [32], [57] or stack-based [59], [58], [26]) with a random access memory[1]. Table I summarizes our classification of RAMs based on obliviousness and verifiability:

Verifiable random access machine (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 [47], [27], [39], [38], [40], [76]. 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 [12], [14], [77], [4], [20], [32], [30] or without it [58], [59], [26], [78], [22], [62]. Some publicly verifiable vRAM schemes with succinctness are also known as zero-knowledge virtual machines (zkVMs) [4], [62] or zero-knowledge Ethereum virtual machines (zkEVMs) [59], [58], [26]. Succinctness means the proof size and verification complexity sub-linear in the size of the statement, i.e., 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 [22], [62], [78] or compatibility [58], [59], [26] by compromising on privacy. Conversely, other vRAMs inherently exhibit obliviousness due to the zero-knowledge property, either with public verifiability [12], [14], [77], [4], [20], [32], [30] or without it [27], [39], [38], [40], [76], ensuring that no information about the prover's secret inputs is leaked to verifiers beyond what can be inferred

TABLE I
Comparison of random access machine schemes.

	Verifiable to participants or designated verifiers	Publicly verifiable
Non-oblivious	vRAM [47]	vRAMs [58], [59], [26] [78], [22], [62]
Oblivious with non-distributed secrets	vRAMs [27], [39], [38], [40], [76]	vRAMs [12], [14], [77], [4] [20], [32], [30]
Oblivious with distributed secrets	DORAMs [49], [74], [52], [45], [24]	This work: **VDORAM**

[1]In this paper, 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.

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 random access machine (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 [64], [57] lack verifiability, while maliciously secure DORAMs [49], [52], [24] 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 [67], [3], while each prover seeks to prevent the disclosure of their secrets against other parties or verifiers. We therefore pose the following question:

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

At first glance, this appears to be an engineering challenge of integrating two known technologies, an MPC system and a multi-prover ZKP system like collaborative zkSNARKs [67]. However, a deeper investigation reveals a fundamental conflict between the design philosophies of MPC protocols and multi-prover ZKP. We detail these challenges along with our contributions in response.

There is a fundamental paradigm mismatch between MPC and ZKP, which has hindered the emergence of a practical multi-prover ZKP front-end. The goal of an MPC protocol is minimalist secure computation; its design is optimized to reduce communication and processing overhead by computing a final result from minimal information. In contrast, the goal of a ZKP system is verifiability. It demands a complete and static trace of the entire computation, where every single intermediate value must serve as a witness within a constraint system. While this integration is straightforward for basic arithmetic like addition and multiplication [48], the paradigm mismatch is particularly evident for more sophisticated functionalities such as comparisons, equality checks, and bit-wise operations. The MPC protocols for these tasks often rely on procedures that do not generate the complete set of auxiliary witnesses required to construct a valid and non-trivial ZKP constraint. This mismatch makes a simple integration of existing tools unworkable. For instance, a state-of-the-art MPC comparison protocol [65] determines a result by extracting only the most significant bit (MSB) of a difference, which is highly efficient for computation. However, this shortcut is incompatible with ZKP systems, as MSB-based approach does not hold on a finite field $\mathbb{F}_p$ where a ZKP system works on, and the single output bit is insufficient evidence to prove the comparison was performed correctly. Therefore, the mismatch has been a primary obstacle preventing the emergence of practical, programmable front-ends for multi-prover ZKPs. Consequently, prior works have had to either assume such a tool exists [67] or resort to proving randomly generated, non-programmable relationships [63], [41].

Contribution. Our work provides CompatCircuit, a multi-prover ZKP front-end that establishes a new MPC-for-ZKP paradigm. Instead of treating MPC and ZKP as incompatible systems, CompatCircuit unifies secure computation and proof generation into a single, compatible circuit paradigm. We identified core functionalities from single-prover front-ends [42], [53] and re-engineered them for the multi-prover context. This resulted in two new fundamental primitives beyond standard arithmetic: a modified inversion-or-zero protocol that safely handles zero inputs, and a full bit-decomposition protocol extracting *all* bits of a field element, fulfilling a strict ZKP requirement that existing MPC tools like MPyC and MP-SPDZ do not meet. Because the underlying primitives are designed to be both secure and provable, any gadget built from them automatically inherits these guarantees. Therefore we build a rich library of higher-level gadgets, such as comparisons, conditional selections, and equality checks, which are created through direct composition. By building on these primitives and gadgets, CompatCircuit allows developers to construct complex multi-prover ZKP applications with a familiar circuit-design experience, effectively brings an MPC-for-ZKP paradigm.

How to make the random access machine execution simultaneously oblivious and publicly verifiable. The entire state transition of the machine, including memory access, must be performed blindly among the provers, yet this blind execution must produce a trace that can be proven correct in a ZKP. Conventional vRAMs are unsuitable because their witness generation is a plaintext execution, which is fundamentally non-oblivious. They optimize under the assumption that this plaintext execution is computationally free. Conversely, existing DORAMs are designed for obliviousness but not for public verifiability. DORAMs seek to minimize communication and computation overheads in oblivious operations, with a focus on optimizing oblivious data structures [51], such as oblivious arrays [49], [52] and oblivious caches [24] to facilitate efficient access across distributed secrets, without considering the requirements of publicly verifiable proof generation.

Contribution. In response, we build VDORAM on top of our CompatCircuit system, the main architecture demonstrated in Figure 1. VDORAM is a register-based random access machine that allows an arbitrary number of provers to blindly execute a program on their distributed secret inputs. The machine state, including the program counter and register values, remains secret-shared throughout the execution. To prevent side-channel leakage, the execution time is constant regardless of the instruction type being processed, ensuring that the operational pattern reveals no information. In the end of the execution, a publicly verifiable proof can be generated to prove the validity of execution to external verifiers who does not

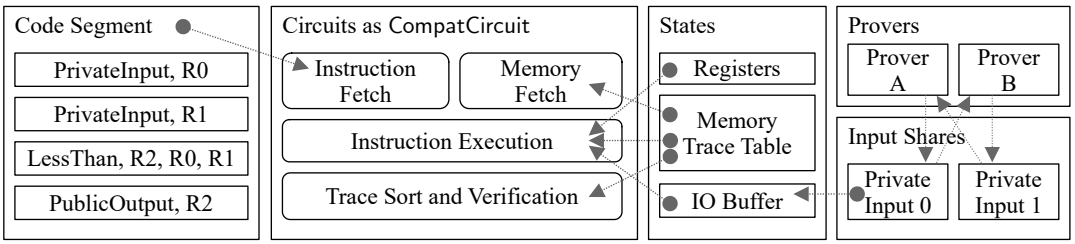


Fig. 1. VDORAM architecture.

trust any of the provers. A central component of VDORAM is its memory management scheme, which is architecturally designed to balance the inverted performance costs of this new paradigm. It reconciles the high communication latency of the online MPC computation with the complexity of offline proof generation. This leads to a design that strategically makes trade-off between these competing demands, resulting in a functional system despite the inherent overheads of its stronger security model. We also make VDORAM open-source at https://github.com/BDS-SDU/vdoram-artifacts/ and demonstrate the capabilities of VDORAM with 7 program examples, micro-benchmarks, and comparasion with prior works.

Our contributions are summarized as follows:

1) We introduce CompatCircuit, the first front-end framework, to our knowledge, for multi-prover ZKP applications that introduces the MPC-for-ZKP paradigm by unifying secure computation and proof generation into a single, compatible circuit.
2) We design and construct VDORAM, the first random access machine that achieves both public verifiability and distributed obliviousness.
3) We provide a full, open-source implementation of our CompatCircuit and VDORAM system and conduct a comprehensive evaluation, demonstrating the practical feasibility of the VDORAM model.

Roadmaps. Section II introduces related works. Section III details the model and preliminaries. In Section IV, we present our VDORAM design, including the CompatCircuit framework, memory management, the full protocol, and analysis. The implementation and evaluation are presented in Section V. Section VI concludes our work and discusses possible future directions.

II. 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 [12], in which the execution proof can be formulated as a zkSNARK arithmetic circuit. Subsequently, vnTinyRAM [14] 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 [58], and Scroll [26]. Since the EVM inherently lacks support for private variables, these projects generally favor succinctness over the zero-knowledge property. In contrast, Polygon Miden [60] 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 [34], 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 [77] is another STARK-based vRAM supporting the RV32IM ISA. Additionally, there are vRAMs with WebAssembly (WASM) support [31]. Recently, Arun et al. designed a RISCV-based vRAM called Jolt [4], 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 [43], ZK from vector oblivious linear evaluation (VOLE) [21], [7], [75], and ZK from garbled circuits [44]. 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 [25], thereby preserving public verifiability.

Heath et al. developed a ZK oblivious RAM, termed BubbleRAM [38], where the zero-knowledge proof is constructed using garbled schemes [44]. Subsequently, BubbleCache [40] was introduced, which enhanced efficiency through the implementation of multi-level caching. Franzese et al. built a constant-overhead interactive verifiable RAM [27], improving memory checking efficiency using a polynomial equality check. Goel et al. [32] developed their implementation, named Dora, based on the proposed concept of ZKBag. By introducing disjunctive zero-knowledge [7], [33], [54], [55], [56], 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 [20].

Distributed Oblivious RAMs. Differing from vRAMs that focus on public verifiability, distributed oblivious random access machines (DORAMs) [73] represent a significant area in a multi-party setting, which is also known as RAM-MPC. 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 [49], leveraging SPDZ [19], an MPC protocol that offers active security. Subsequently, Keller et al. devised a garbled-circuit-based RAM with active security [52], which minimizes the number of broadcast rounds between memory accesses. Ji et al. [45] concentrated on constructing a RAM capable of private function evaluation. Falk et al. have introduced a 3-party distributed RAM-MPC framework [24] that boasts logarithmic overhead and is resilient to malicious adversaries. In addition, Hamlin et al. proposed their first construction of FHE-based ORAM [37], [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.

III. 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.

A. 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 to verifiers: it is assumed that 0 provers will act honestly, i.e., all provers are eager to collude to compromise the correctness of results for their personal gains. *Privacy:* Provers do not seek privacy if they are able to compromise correctness. If it's nearly impossible to compromise correctness, the provers attempt to protect their secrets from being exposed: each prover is considered as a polynomial time adversary, curious to obtain confidential information from other provers, where at least t (e.g., $t = 1$) provers will act honestly to protect their secrets from being exposed.
- *Non-interactive verifiers.* Similar 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 [67] 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.

B. Preliminaries

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 [35], [66], [29], [16] 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 [67]. This concept were further improved in [63], [3], [41]. 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:

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

2) Multi-Party Computation: A multi-party computation (MPC) protocol [19], [11], [50] 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 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 $[r] \leftarrow [\gamma] + \delta \cdot [\beta] + \epsilon \cdot [\alpha] + \delta \cdot \epsilon$.

To facilitate complex operations beyond simple arithmetic, particularly those requiring bit-level manipulation within a prime field setting, we additionally utilize two techniques in mixed MPC: doubly-authenticated bit (daBit) [70] and extended doubly-authenticated bits (edaBit) [23]. A daBit consists of a random bit $r \in [0, 1]$ that is secret-shared among parties in both an arithmetic domain $[r]_p \in \mathbb{F}_p$ and a Boolean domain $[r]_2 \in [0, 1]$, useful to convert a Boolean secret share into an arithmetic share by utlizing $\mathsf{B2A}_p$ protocol [1]. Similarly, an edaBit provide shares of a random value $r \in \mathbb{F}_p$ in an arithmetic domain $[r]_p \in \mathbb{F}_p$, along with shares of its individual bits $[r]_i \in [0, 1]$ ($i \in [0, l)$, where l denotes the bit length of p) in a Boolean domain.

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 [6], [8], [9], [69], [48]. Alternatively, Ozdemir et al. [67] 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) Arithmetic Circuit: An arithmetic circuit [14], [68], [15], [10] 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).

4) Rank-1 Constraint System: An R1CS is a set of constraints, where *each* constraint is a triplet of vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ satisfying $(\mathbf{a} \cdot \mathbf{s}) \cdot (\mathbf{b} \cdot \mathbf{s}) - \mathbf{c} \cdot \mathbf{s} = 0$ in $\mathbb{F}_p$. Here, s is the solution vector, which contains instances (all constant or public inputs) and witnesses (all private inputs and all intermediate values). This structure forces any complex computation to be flattened into a series of additions and multiplications checks, which is typically done by a ZKP front-end.

5) Verifiable Random Access Machine: We adopt the definition from [4] and [31]. 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.

IV. 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 Section IV-A, achieving publicly verifiable MPC with multi-prover ZKPs. Subsequently, in Section IV-B, 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 Section IV-C, 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 Section IV-D and provide analysis in Section IV-E.

A. Overview

The overall workflow of VDORAM is shown in Figure 2. 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 [14], [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 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

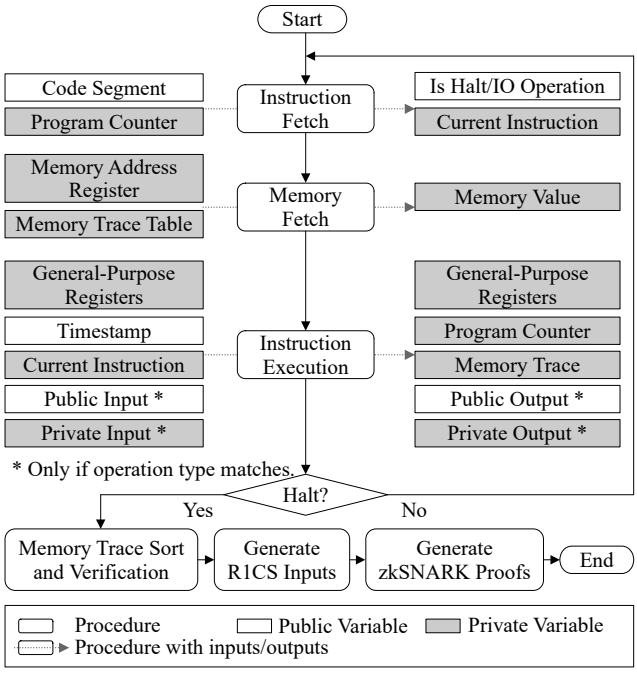


Fig. 2. Workflow of VDORAM.

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 Section IV-B and Section IV-C.

B. CompatCircuit: A Unified Front-End for Multi-Prover Computation and Multi-Prover Proofs

We introduce CompatCircuit, a front-end for multi-prover ZKPs that systematically adapts MPC protocols for generating R1CS witnesses. Our approach eliminates dependencies on plaintext values common in single-prover front-ends by replacing conditional branching with oblivious computations, such as using an inversion-based zero test for equality checks instead of relying on knowing the condition's validity [20]. We also redesign MPC protocols unsuitable for ZKPs; for instance, where an efficient MPC comparison [65] is insufficient for ZKP statement construction, we substitute it with a sequence of boolean operations on decomposed bits. To this end, we improved the bit-decomposition protocol from [18], that supports extract *all* bits from a finite field $\mathbb{F}_p$, overcoming the limitations of existing MPC libraries [71], [50] which do not extract all bits required for ZKP. Furthermore, CompatCircuit acts as a compatibility layer to unify the MPC and ZKP

codebases, providing a familiar development environment akin to single-prover ZKP tools and thereby minimizing implementation complexity and human error.

We propose our CompatCircuit as four primitives and gadgets. Primitives include 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` [42] and `jsnark_interface` [53], as the composition of the four primitives, namely gadgets.

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 [23].
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 $\mathsf{B2A}_p$ protocol proposed in Prio+ [1].
6) Compute witnesses for R1CS constraints $a = \sum_{i=0}^{l-1} r_i \cdot 2^i$ and $(r_0, r_1, \ldots, r_{l-1}) < (p_0, p_1, \ldots, p_{l-1})$.

Fig. 3. The protocols for inversion-or-zero and full bit-decomposition in CompatCircuit. The inversion-or-zero protocol generalizes the input domain to include zero while preserving privacy over whether the input is zero. The full bit-decomposition protocol ensures all bits of the input are extracted in their arithmetic shares, enabling the satisfaction of R1CS constraints.

Inversion-or-zero: Inspired by [42], 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 [18] since other constructions [50], [71] do not apply. Initially, each participant holds an arithmetic additive secret share $[a]$ of a field element $a \in \mathbb{F}_p$. 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 [18], 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 refer to Appendix A for the correctness.

We now construct other commonly used functionalities [42], [53] 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 $r \leftarrow a \cdot c + b \cdot (1-c)$.
- *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 $b \leftarrow 1 - a \cdot \mathsf{inv}$.

- *Equality check.* To avoid conditionally branching introduced in [20], we migrate the implementation by [42], 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 [65], the increased overhead is justified by the proof requirements.

With CompatCircuit, it becomes straightforward to construct the execution circuit for VDORAM. We provide protocols of all circuits in Appendix C. 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.

C. Memory Management: Balancing Oblivious Computation Overheads with Proof Generation Complexity

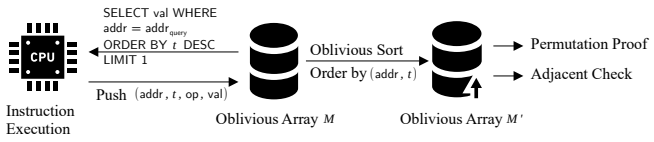


Fig. 4. Memory management scheme of VDORAM.

As shown in Figure 4, we construct a multi-prover memory management scheme based on [27]. The main procedures are as below: 1) During each instruction execution, a memory access tuple $m = (\mathsf{addr}, t, \mathsf{op}, \mathsf{val})$ 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, $p-1$ 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 [27] holds, ensuring that for each load operation, the value retrieved matches the most recently stored value at that address:

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

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 [73] used by [49], or a 3-party random access memory [24].

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.

Fig. 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. Protocol $\Pi_{\text{HistoricalKV}}$:

- Upon receiving $\Pi_{\text{HistoricalKV}}.\mathsf{Add}(\mathsf{addr}, \mathsf{val}, \mathsf{aux})$, insert the tuple to the end of array M.
- Upon receiving $\Pi_{\text{HistoricalKV}}.\mathsf{Query}(\mathsf{addr})$, 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 $\Pi_{\text{HistoricalKV}}.\mathsf{Export}()$, 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 [5] [46]. We opted for Bitonic mergesort; however, adapting this algorithm for VDORAM necessitates modifications. Initially, the scheme from [27] 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 $m_{\mathsf{pad}} = (p-1, p-1, p-1, p-1)$, where $p - 1 \in \mathbb{F}_p$. 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 [20] are generally considered more efficient than [27], 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 [27], 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, [20] expands the vector to size $2T$, implementing a permutation proof and $T-1$ difference checks on adjacent timestamp values. In the single-prover scenario, [20] offers enhanced efficiency over [27], 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 this overhead in our evaluation, Section V. Given these considerations, for our multi-prover implementation, we support the adoption of the memory checking protocol proposed in [27] in contrast to [20], due to its relative simplicity and reduced computational and communicational demands in multi-prover contexts.

D. Protocol Specification

Putting it all together, we construct our VDORAM, providing the full protocol below.

Protocol Π_{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. Provers generate 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 $\mathsf{op}_{\mathsf{masked}} \in \{\texttt{input}, \texttt{output}, \texttt{secret}\}$. This protocol does not reveal information about an non-IO instruction.
5) If $\mathsf{op}_{\mathsf{masked}} = \texttt{input}$, 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.

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.

E. Performance and Security Analysis

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 $\mathsf{B2A}_p$ 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.

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.

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.

The formal definition, our claim on the security properties, and security proofs of VDORAM will be shown in Appendix B.

V. 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` [67] (≈ 300 lines). Our source code is available at https://github.com/BDS-SDU/vdoram-artifacts. Figure 6 shows an example of designing a circuit in CompatCircuit. VDORAM details as well as the instruction set will be shown in Table IV, Appendix C.

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 provers 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 [29].

```
public class DemoCircuit : ICircuitBoardGenerator {
    public CircuitBoard GetCircuitBoard() {
        CircuitBoard cb = new();

        // Private inputs
        Wire a = Wire.NewPrivateInputWire("a");
        Wire b = Wire.NewPrivateInputWire("b");
        cb.AddWires(a, b);

        // Comparison gadget
        GadgetInstance g = new FieldLessThanGadget()
            .ApplyGadget([a, b], "a_lt_b");
        g.Save(cb);

        // Public output
        g.OutputWires[0].Name = "out_lt";
        g.OutputWires[0].IsPublicOutput = true;

        return cb;
    }
}
```

Fig. 6. An example of designing a circuit in CompatCircuit, our unified multi-prover ZKP front-end that unifies representation for secure computation and R1CS constraints.

The variables altered in our experiments included: the number of MPC provers (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.

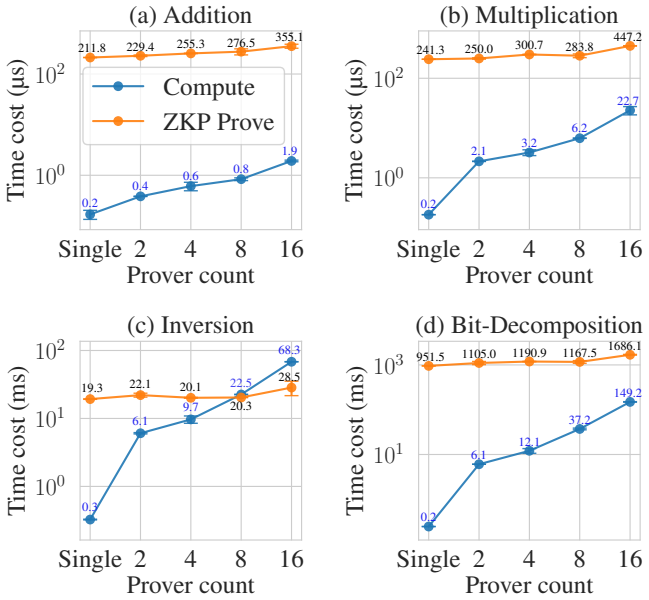


Fig. 7. Micro-benchmarks of CompatCircuit primitives.

Micro-benchmarks of CompatCircuit. 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 prover 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 as the number of provers 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.

Comparison of CompatCircuit. To contextualize CompatCircuit's capabilities, we compare it with prior multi-prover ZKP works, specifically Collaborative zkSNARKs [67], Liu et al. [63], and Hu et al. [41]. Table II summarizes the key differences among these systems.

TABLE II
COMPARISON OF COMPATCIRCUIT WITH PRIOR WORKS

Multi-Prover System	Primary Focus	Computation Support
Collaborative zkSNARKs [67]	ZKP back-end	Limited (hardcoded multiplications)
Liu et al. [63]	Proof delegation	No (proving random relationship)
Hu et al. [41]	Proof delegation	No (proving random relationship)
CompatCircuit (ours)	**ZKP front-end**	**Yes**

We experimentally compared CompatCircuit with Collaborative zkSNARK using a benchmark of n-th sequential squaring ($v \leftarrow v^2$), as this is the only hardcoded computation that Collaborative zkSNARK supports natively without a front-end. Figure 8 illustrates the time costs for different procedures with varying numbers of provers and input sizes. The results demonstrate that both systems exhibit comparable performance, indicating that CompatCircuit achieves its additional features without significant performance penalties.

Micro-benchmarks of VDORAM. We subsequently assessed the performance of the VDORAM within an 8-prover configuration, as shown in Figure 9. We varied both the types and counts of instructions. In Figure 9(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 9(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

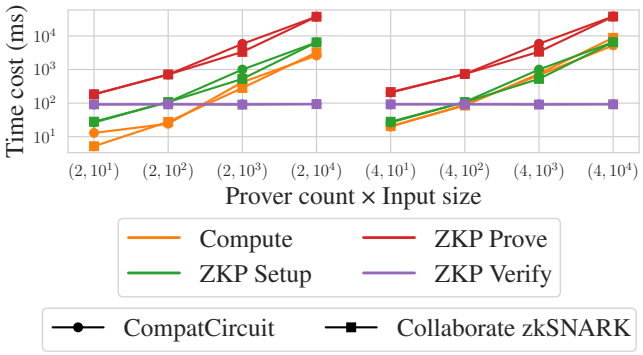


Fig. 8. Performance comparison between CompatCircuit and Collaborative zkSNARK. In this figure, the compute procedure also contains preprocessing time costs.

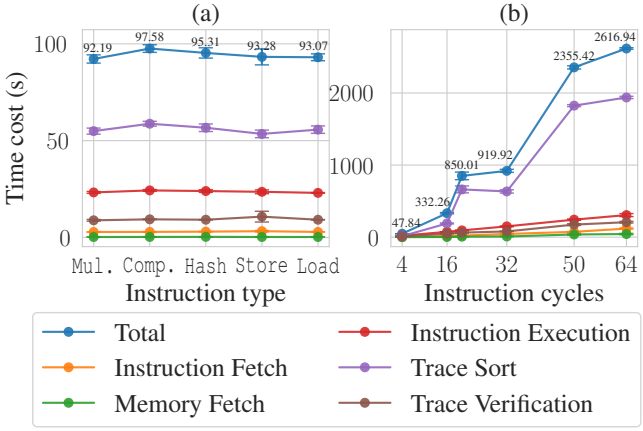


Fig. 9. Computation time costs of VDORAM (8 provers) with (a) varying instruction types and (b) varying instruction cycles. Time costs remain constant with different types but increase with instruction cycles.

(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.

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 10. We also estimated the bandwidth requirements during the computation and collaborative ZKP process: for communicating with each of the other $m - 1$ prover 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

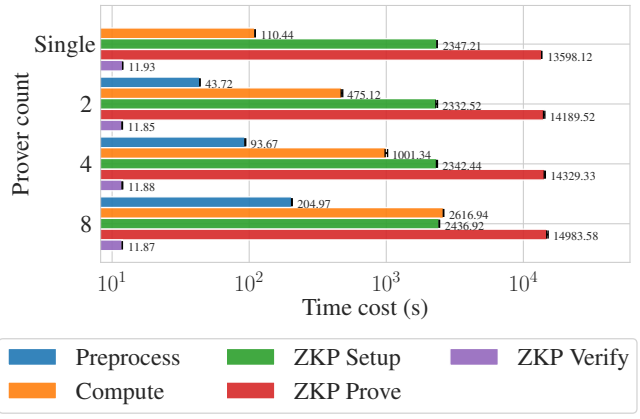


Fig. 10. Time costs of VDORAM procedures (instruction cycles: 64).

communication, our implementation still maintains a relatively reasonable overheads in total, compared with the single-prover baseline.

Program Examples of VDORAM. To evaluate VDORAM's applicability, we implemented seven example programs as shown in Figure 11: bubble sort, sliding window, set intersection, range query, longest continuous increasing subsequence (LCIS), Fibonacci, and binary search, highlighting VDORAM's versatility in handling various computational patterns.

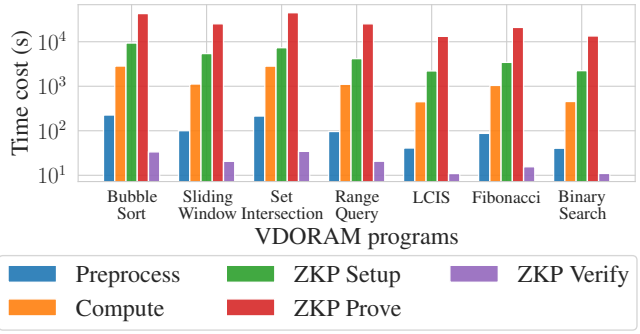


Fig. 11. Time costs of VDORAM procedures for programs (2 provers).

Comparison of VDORAM. We compared VDORAM with prior RAM systems: Jolt [4], SP1 [61], and Wang et al. [74]. Table III highlights the feature differences, emphasizing VDORAM's unique ability to support both public verifiability and distributed obliviousness for an arbitrary number of provers.

TABLE III
COMPARISON OF VDORAM WITH PRIOR WORKS

RAM System	Number of Parties	Public Verifiability	Distributed Obliviousness
Jolt [4]	1 prover	Yes	No
SP1 [61]	1 prover	Yes	No
Wang et al. [74]	2 parties	No	Yes
VDORAM (ours)	≥ 1 **provers**	**Yes**	**Yes**

We also conducted an experimental comparison on set intersection and binary search programs. Figure 12 shows the time costs for each system's main procedures. The results indicate that VDORAM's additional overhead is justified by its multi-prover and proof generation capabilities.

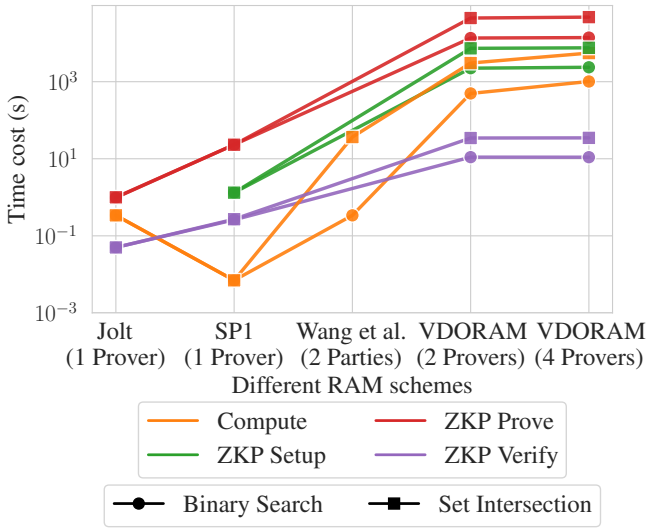


Fig. 12. Performance comparison of VDORAM with prior works on set intersection and binary search programs. In this figure, the compute procedure of Wang et al. and VDORAM also contains preprocessing time costs. Jolt does not require ZKP setup procedure.

VI. 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 a RAM design that optimizes communication overhead and proof generation time. We have developed implementations of both CompatCircuit and VDORAM. Our evaluation with micro-benchmarks, comparisons, and program examples validates the feasibility of our approach. Establishing this stronger security model also brings some limitations that highlight opportunities for further optimization, shown below.

- *Computation and Proof Overhead.* Our work establishes a functional memory model for arbitrary number of provers by deliberately trading higher computational communication overhead for a simpler memory consistency proof. Future work could advance this trade-off by developing more efficient n-party oblivious data structures to reduce communication costs. Concurrently, adapting modern verification techniques like lookup arguments [28], [4] to the multi-prover setting would further lower the proof generation overhead.

- *Potential Leakage Through I/O Instruction*. A minor information leakage channel exists as provers can observe the timing of public I/O and halt instructions, for which we have proposed mitigations in Section IV-E3. Future work could explore executing a VM with a blind I/O operation.
- *Trusted Setup Requirement*. The protocol relies on a trusted setup for the underlying collaborative zk-SNARKs, which is a common requirement for many zk-SNARK-based systems. Our implementation utilizes Plonk [29], requiring a universal trusted Setup. Future research could investigate transparent zk-SNARKs [72], [17] to eliminate the assumption of the trusted initializer.

ACKNOWLEDGMENT

This study was supported by the Key R&D Program of Shandong Province (No. 2025CXPT033), the National Natural Science Foundation of China (No. 62302266, 62232010, U23A20302), the Shandong Science Fund for Excellent Young Scholars (No. 2023HWYQ-008), and Shandong Provincial Natural Science Foundation (ZR2022ZD02).

REFERENCES

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

[2] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 191–219. Springer, 2016. https://link.springer.com/chapter/10.1007/978-3-662-53887-6_7.

[3] Mohammed Alghazwi, Tariq Bontekoe, Leon Visscher, and Fatih Turkmen. Collaborative cp-nizks: Modular, composable proofs for distributed secrets. *arXiv preprint arXiv:2407.19212*, 2024.

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

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

[6] Carsten Baum, Ivan Damgård, and Claudio Orlandi. 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*, pages 175–196. Springer, 2014.

[7] Carsten Baum, Alex J Malozemoff, Marc B Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part IV 41*, pages 92–122. Springer, 2021.

[8] Carsten Baum, Emmanuela Orsini, and Peter Scholl. 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*, pages 461–490. Springer, 2016.

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

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

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

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

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

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

[15] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. 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*, pages 327–357. Springer, 2016. https://link.springer.com/chapter/10.1007/978-3-662-49896-5_12.

[16] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 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*, pages 738–768. Springer, 2020. https://link.springer.com/chapter/10.1007/978-3-030-45721-1_26.

[17] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 769–793. Springer, 2020.

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

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

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

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

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

[23] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 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*, pages 823–852. Springer, 2020. https://link.springer.com/chapter/10.1007/978-3-030-56880-1_29.

[24] Brett Falk, Daniel Noble, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. Doram revisited: maliciously secure ram-mpc with logarithmic overhead. In *Theory of Cryptography Conference*, pages 441–470. Springer, 2023.

[25] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory*

and application of cryptographic techniques, pages 186–194. Springer, 1986.

[26] Scroll Foundation. Scroll – native zkevm layer 2 for ethereum. https://scroll.io/ Accessed: Jul 15, 2024.

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

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

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

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

[31] Sinka Gao, Guoqiang Li, and Hongfei Fu. Zkwasm: A zksnark wasm emulator. *IEEE Transactions on Services Computing*, 2024. https://ieeexplore.ieee.org/abstract/document/10587123.

[32] Aarushi Goel, Mathias Hall-Andersen, and Gabriel Kaptchuk. Dora: A simple approach to zero-knowledge for ram programs. *Cryptology ePrint Archive*, 2023.

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

[34] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo–a turing-complete stark-friendly cpu architecture. *Cryptology ePrint Archive*, 2021.

[35] Jens Groth. 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*, pages 305–326. Springer, 2016. https://link.springer.com/chapter/10.1007/978-3-662-49896-5_11.

[36] Ariel Hamlin, Justin Holmgren, Mor Weiss, and Daniel Wichs. Fully homomorphic encryption for rams. *Cryptology ePrint Archive*, 2019.

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

[38] David Heath and Vladimir Kolesnikov. A 2.1 khz zero-knowledge processor with bubbleram. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2055–2074, 2020.

[39] David Heath and Vladimir Kolesnikov. Proram: Fast o (log n) authenticated shares zk oram. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 495–525. Springer, 2021.

[40] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast zk processor with cached oram for ansi c programs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1538–1556. IEEE, 2021.

[41] Yuncong Hu, Pratyush Mishra, Xiao Wang, Jie Xie, Kang Yang, Yu Yu, and Yuwen Zhang. Dfs: Delegation-friendly zksnark and private delegation of provers. *Cryptology ePrint Archive*, 2025.

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

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

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

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

[46] Kristjän Valur Jönsson, Gunnar Kreitz, and Misbah Uddin. Secure multi-party sorting and applications. *Cryptology ePrint Archive*, 2011.

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

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

[49] Marcel Keller. The oblivious machine-or: how to put the c into mpc. *Cryptology ePrint Archive*, 2015.

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

[51] Marcel Keller and Peter Scholl. 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*, pages 506–525. Springer, 2014.

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

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

[54] Abhiram Kothapalli and Srinath Setty. Supernova: Proving universal machine executions without universal circuits. *Cryptology ePrint Archive*, 2022.

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

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

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

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

[59] Polygon Labs. Polygon zkevm — scaling for the ethereum virtual machine, 2022. https://polygon.technology/polygon-zkevm Accessed: Jul 15, 2024.

[60] Polygon Labs. Polygon miden — a rollup for high-throughput, private applications, 2024. https://polygon.technology/polygon-miden Accessed: Jul 15, 2024.

[61] Succinct Labs. Sp1. https://github.com/succinctlabs/sp1 Accessed: July 1, 2025.

[62] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. Ceno: Non-uniform, segment and parallel zero-knowledge virtual machine. *Cryptology ePrint Archive*, 2024. https://eprint.iacr.org/2024/387.

[63] Xuanming Liu, Zhelei Zhou, Yinghao Wang, Jinye He, Bingsheng Zhang, Xiaohu Yang, and Jiaheng Zhang. Scalable collaborative zk-snark and its application to efficient proof outsourcing. *Cryptology ePrint Archive*, 2024. https://eprint.iacr.org/2024/940.

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

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

[66] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 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*, pages 2111–2128, 2019. https://dl.acm.org/doi/abs/10.1145/3319535.3339817.

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

[68] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[69] Marc Rivinius, Pascal Reisert, Daniel Rausch, and Ralf Küsters. Publicly accountable robust multi-party computation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2430–2449. IEEE, 2022.
[70] Dragos Rotaru and Tim Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *International Conference on Cryptology in India*, pages 227–249. Springer, 2019. https://link.springer.com/chapter/10.1007/978-3-030-35423-7_12.
[71] Berry Schoenmakers. Mpyc—python package for secure multiparty computation. In *Workshop on the Theory and Practice of MPC. https://github. com/lschoe/mpyc*, 2018.
[72] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Annual International Cryptology Conference*, pages 704–737. Springer, 2020.
[73] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861, 2015.
[74] Xiao Wang, S Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of mips machine code. In *European Symposium on Research in Computer Security*, pages 99–117. Springer, 2016.
[75] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 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*, pages 2986–3001, 2021.
[76] Yibin Yang and David Heath. Two shuffles make a ram: Improved constant overhead zero knowledge ram. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1435–1452, 2024.
[77] RISC Zero. Risc zero — universal zero knowledge. https://www.risczero.com/ Accessed: Aug 29, 2024.
[78] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vram: Faster verifiable ram with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 908–925. IEEE, 2018.

APPENDIX

A. Security Analysis of CompatCircuit Primitives and Gadgets

Besides field addition and multiplication, we define the ideal functionalities of the other two CompatCircuit primitives here.

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

This functionality interacts with a set of prover servers $S_1, S_2, \ldots, S_m$ and an adversary $\mathcal{S}$. Let corr be the set of corrupted prover servers. Denote $l = \lceil \log_2 p \rceil$.

Upon receiving $(\text{InvOrZero}, [a])$ from all servers, do:

- Receive a set of shares $\{u_{i,j}\}_{i \in \text{corr}}$ from the adversary $\mathcal{S}$, where $u_{i,j}$ refers to the j-th share of multiplication result in computing a^{p-2} with fast exponentiation.
- Reconstruct a from $[a]$.
- Compute $r \leftarrow a^{p-2}$ with fast exponentiation.
- Sample random sharing $[r]$ of r, such that the shares of the corrupted servers are identical to those received from the adversary $\mathcal{S}$.
- Distribute $[r]$ to all servers.

Theorem A.1. *The protocol $\Pi_{\mathit{InvOrZero}}$ defined in Figure 3, Section IV-B securely realizes $\mathcal{F}_{\mathit{InvOrZero}}$ against a semi-honest adversary corrupting up to t servers.*

Proof. The correctness follows from the construction directly. The soundness follows from the R1CS constraints: if $a = 0$, any incorrect $r \neq 0$ violates the constraint $r \cdot (a \cdot r - 1) = 0$; otherwise, $a \neq 0$, any incorrect value $r \neq a^{p-2}$ violates either $a \cdot (a \cdot r - 1) = 0$ $(r = 0)$ or $r \cdot (a \cdot r - 1) = 0$ $(r \neq 0)$. The zero-knowledge property is ensured by the underlying collaborative-zkSNARKs protocol [67] since a, r, and intermediate values $a \cdot r$, $a \cdot r - 1$ are included as witnesses of collaborative-zkSNARKs, revealing no information beyond what the instances exposed to the verifier by definition. The t-distributed-obliviousness is satisfied because all involved values – input $[a]$, output $[r]$, and intermediate values in computing $[r] \leftarrow [a]^{p-2}$ are secret shares. Constant p is a public parameter. No public values are revealed in the protocol. Besides communications involved by field multiplications, the protocol only perform local computations. Therefore, corrupted servers cannot learn any information about the input. The t-distributed-obliviousness is thereby guaranteed. □

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

This functionality interacts with a set of prover servers $S_1, S_2, \ldots, S_m$ and an adversary $\mathcal{S}$. Let corr be the set of corrupted prover servers. Denote $l = \lceil \log_2 p \rceil$.

Upon receiving $(\text{FBits}, [a])$ from all servers, do:

- Receive shares $\{u_{i,j}\}_{i \in \text{corr}}$ from adversary $\mathcal{S}$, where $u_{i,j}$ refers to the j-th share in bit adders with carry up and $\mathsf{B2A}_p$ protocol.
- Reconstruct a from $[a]$.
- Compute the plaintext bit-decomposition on a, getting bits $(r_0, r_1, \ldots, r_{l-1})$.
- Sample random sharing $[r_0, r_1, \ldots, r_{l-1}]$ of $(r_0, r_1, \ldots, r_{l-1})$, such that the shares of the corrupted servers are identical to those received from the adversary $\mathcal{S}$.
- Distribute $[r_0, r_1, \ldots, r_{l-1}]$ to all servers.

Theorem A.2. *The protocol Π_{FBits} defined in Figure 3, Section IV-B securely realizes $\mathcal{F}_{\mathit{FBits}}$ against a semi-honest adversary corrupting up to t servers.*

Proof. The correctness of Π_{FBits} can be analyzed using similar techniques in [18]: let $a, b, c, \ldots$ denote the integer values corresponding to the secret shares $[a], [b], [c], \ldots$ manipulated throughout the protocol. The reveal value $c = a - b \in \mathbb{F}_p$ implies $c = a - b + q' \cdot p$ $(q' \in \{0, 1\})$ over integers. The next computation $d = c + 2^l - p \in \mathbb{Z}_{2^{l+1}}$ does not involve overflow or underflow, yields the integer $d = a - b + 2^l + (q' - 1)p$. The subsequent addition $e = b + d$ does not involve overflow or underflow neither, yielding $e = b + d = a + 2^l + (q' - 1)p \in \mathbb{Z}_{2^{l+1}}$, i.e., $l + 1$ bits. We denote the most significant bit e_l as q. It is important to note that e can also be casted into an l-bit element $\mathbb{Z}_{2^l}$, where the bit q representing whether an overflow happens in $\mathbb{Z}_{2^l}$, i.e., $q = 1 \iff e \geq 2^l$. Next, we prove $q = q'$: if $q' = 1$, $e = a + 2^l \geq 2^l$; otherwise, $q' = 0$, $e = a + 2^l - p = 2^l + (a - p) < 2^l$ (since $a < p$). Now we have $q = q'$, the computation $f = (1 - q)p = (1 - q')p$ and final addition $a' = e + f = a + 2^l + (q' - 1)p + (1 - q')p = a + 2^l$ yield the correct l-bit representation of a. In the end, the boolean shares of bits representing a' can be correctly converted into

arithmetic shares of bits using $\mathsf{B2A}_p$ protocol [1]. We have therefore proven the correctness of protocol Π_{FBits}.

The soundness holds from R1CS constraints. There are two cases for corrupted r_i ($i \in [0, l)$). The first case is $\sum_{i=0}^{l-1} r_i \cdot 2^i$ results in an overflow in $\mathbb{Z}_p$, which causes $\sum_{i=0}^{l-1} r_i \cdot 2^i = a + p = a \in \mathbb{F}_p$, passing the first constraint. This violates the second constraint $(r_0, r_1, \ldots, r_{l-1}) < (p_0, p_1, \ldots, p_{l-1})$ which requires $\sum_{i=0}^{l-1} r_i \cdot 2^i < p$ over integers. For the other case, any corrupted bits r_i different than the correct ones result in a $\sum_{i=0}^{l-1} r_i \cdot 2^i \in [0, l) \neq a \in \mathbb{F}_p$, violating the first constrant. Thus, the soundness holds.

The zero-knowledge also follows from the underlying collaborative-zkSNARKs protocol [67] since p, l, p_i ($i \in [0, l)$) are instances derived from the public parameter p while a, r_i ($i \in [0, l)$), and all intermediate values are included as witnesses, revealing nothing beyond what instances expose to the verifier by definition.

For t-distributed-obliviousness, we first prove all public and secret share values revealing no information: values p, l, p_i ($i \in [0, l)$) are derived from the public parameter p. Value $c \in \mathbb{F}_p$ is exposed but leaks no information about the input, since c is computed from $a - b$ where b is a uniformly random value in $\mathbb{F}_p$, resulting in a uniformly random value c. Values d and d_i ($i \in [0, l]$) are directly computed from c and public values l and p, revealing no information. e_i ($i \in [0, l]$) are secret shares computed from shares of random bits b_i ($i \in [0, l)$) and public value d_i ($i \in [0, l]$). q is denoted from e_l. f_i ($i \in [0, l)$) are secret shares computed from secret shares $[1 - q]$ and public value p_i ($i \in [0, l)$). a'_i ($i \in [0, l)$) are secret shares computed from secret shares e_i ($i \in [0, l)$) and f_i ($i \in [0, l)$), revealing no information about the input. The final step also reveals no information, ensured by the $\mathsf{B2A}_p$ protocol [1]. Therefore, all public values and secret share values revealing no information. Besides communications involved by field multiplications, boolean operations, and the $\mathsf{B2A}_p$ protocol [1], the protocol Π_{FBits} only perform local computations. Corrupted servers cannot learn any information about the input. The t-distributed-obliviousness is thereby guaranteed. □

With CompatCircuit primitives defined, all other complex operations known as gadgets are directly composed from the primitives, without needing to separate the secure computation phase and R1CS constraints for zero-knowledge proofs. These protocols mainly include MiMC hash Π_{MiMCHash}, zero testing Π_{ZeroTest}, equality check $\Pi_{\text{EqualityCheck}}$, and less-than comparison Π_{LessThan}. We omit the construction here and refer readers in our source code for more details. Figure 13 below demonstrates the dependencies of these gadgets.

Theorem A.3. *The gadget protocols listed in Figure 13 securely realize the corresponding ideal functionalities against a semi-honest adversary corrupting up to t servers.*

Proof. The correctness of these gadgets followed from the construction directly. The soundness and zero-knowledge follow from the underlying CompatCircuit primitives since the gadgets are directly composed by a sequence of CompatCircuit primitives. Besides directly involing the four CompatCircuit primitives, these protocols only perform local computations and involve only secret shares, and therefore corrupted servers cannot learn any information about the input. The t-distributed-obliviousness of these gadget protocols is thereby guaranteed. □

B. Formal Definition and Security Claim of VDORAM

Now, we formally define VDORAM as Definition A.1 based on [67] and [13].

Theorem A.4. *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.*

Theorem A.4 shows our claim on the security properties of VDORAM. Based on security analysis on underlying CompatCircuit primitives and gadgets as well as VDORAM circuits, we can prove the security properties of VDORAM. We will provide the proof in Appendix C.

C. Protocols and Security Analysis of VDORAM Circuits

VDORAM consists of five circuits: instruction fetch $\Pi_{\text{InstructionFetch}}$, memory fetch $\Pi_{\text{MemoryFetch}}$ (without ZKP), instruction execution $\Pi_{\text{InstructionExecution}}$, trace sort $\Pi_{\text{TraceSort}}$ (without ZKP), and trace verification $\Pi_{\text{TraceVerification}}$. By incorporating with CompatCircuit, these circuits simultaneously demonstrate both computational steps for secure computation and the constraints for zero-knowledge proof generation. Protocols of these circuits are shown below.

1) Instruction Fetch: Provers who execute the VM need to blindly fetch the next instruction to be executed. Given an n-size code image vector $\mathbf{I}$ as a public input and the program counter pc as a private input, provers fetches the next instruction I as a private output, by selecting the pc-th item from $\mathbf{I}$. As is typical in traditional zkSNARKs frameworks, accessing a private index involves iterating over each element I_i in the vector $\mathbf{I}$ and setting the selection bit s_i to 1 if and only if the item index i is equal to the private index pc. The desired instruction $I = I_{\mathsf{pc}}$ is then determined by computing the sum of all products $s_i \cdot I_i$.

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

This functionality interacts with a set of prover servers $S_1, S_2, \ldots, S_m$ and an adversary $\mathcal{S}$. Let corr be the set of corrupted prover servers.

Upon receiving $(\text{InstructionFetch}, [\mathbf{I}], [\mathsf{pc}])$ from all servers, do:

- Receive shares $\{u_{i,j}\}_{i \in \mathsf{corr}}$ from adversary $\mathcal{S}$, where $u_{i,j}$ refers to the j-th share in field multiplication and $\Pi_{\text{EqualityCheck}}$ protocol.
- Reconstruct $\mathbf{I}$ from $[\mathbf{I}]$ and pc from $[\mathsf{pc}]$.
- Let I be the pc-th item in $\mathbf{I}$.

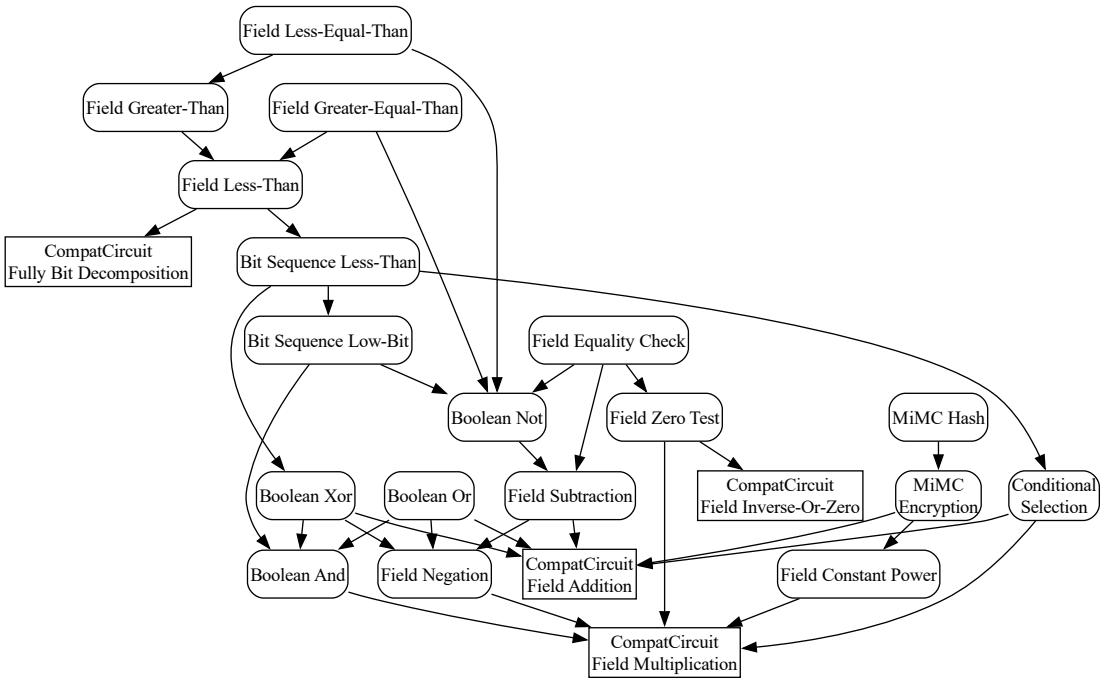


Fig. 13. CompatCircuit gadgets. With CompatCircuit primitives defined, these gadgets can be directly composed from the primitives, without needing to separate the secure computation phase and R1CS constraints for zero-knowledge proofs.

- Denote the type of I as op. If op is in $\{\texttt{input}, \texttt{output}, \texttt{secret}, \texttt{halt}\}$, set $\mathbf{op_{masked}} \leftarrow$ op; otherwise, set $\mathbf{op_{masked}} \leftarrow 0$.
- Sample random sharing $[I]$ of I, such that the shares of the corrupted servers are identical to those received from the adversary $\mathcal{S}$.
- Distribute $[I]$ to all servers. Publicly output $\mathbf{op_{masked}}$.

Protocol $\Pi_{\text{InstructionFetch}}$

This protocol interacts with a set of prover servers $S_1, S_2, \ldots, S_m$.

Upon receiving $(\text{InstructionFetch}, [\mathbf{I}], [\mathsf{pc}])$ from all servers, do:

- For each index $i \in [1, |[\mathbf{I}]|]$, compute $[s_i] \leftarrow \Pi_{\text{EqualityCheck}}([i], [\mathsf{pc}])$.
- Compute and output $[I] \leftarrow \sum_{i=1}^{n} [s_i] \cdot [I_i]$.
- Denote the type of I as op. If op is in $\{\texttt{input}, \texttt{output}, \texttt{secret}, \texttt{halt}\}$, set $\mathbf{op_{masked}} \leftarrow$ op; otherwise, set $\mathbf{op_{masked}} \leftarrow 0$. Publicly output $\mathbf{op_{masked}}$.

Theorem A.5. *The protocol $\Pi_{\text{InstructionFetch}}$ securely realizes $\mathcal{F}_{\text{InstructionFetch}}$ against a semi-honest adversary corrupting up to t servers.*

Proof. The correctness follows from the construction directly. The soundness follows from the underlying CompatCircuit primitives and gadgets. The zero-knowledge holds since beyond the public output $\mathbf{op_{masked}}$ acting as an instance, other involved values are included as witnesses, ensuring zero-knowledge property provided by CompatCircuit primitives and gadgets. The t-distributed-obliviousness holds since (a) besides the public output $\mathsf{op}_{\mathbf{masked}}$, other values are secret shares, and (b) besides communications involved by field multiplications and the $\Pi_{\text{EqualityCheck}}$ protocol as a CompatCircuit gadget, the protocol $\Pi_{\text{InstructionFetch}}$ only perform local computations. Therefore, corrupted servers cannot learn any information about the input. The t-distributed-obliviousness is thereby guaranteed. □

2) Memory Fetch: Since all memory operations are confidential, it requires a mechanism for provers to blindly retrieve the correct memory value for a specified address. Given an n-size memory trace table $\mathbf{m}$ and a memory address addr as private inputs, a memory fetch circuit fetches the latest memory value memval as a private output, from the row whose memory address matches addr. If multiple rows meet this condition, the row with the highest global step counter should be chosen. Given that rows are ordered sequentially according to their global step counter, the implementation is similar with instruction fetch circuit. Specifically, it sets the selection bit s_i to 1 if and only if the memory address of current row addr_i equals addr. However, since it is possible for multiple rows to have $s_i = 1$, provers must preserve only one true bit without explicit knowledge of each bit's state. To do so, provers perform a LowBit computation, which modifies the vector $\mathbf{s} = (s_{n-1}, s_{n-2}, \ldots, s_0)$ by keeping only the first true bit in this vector while setting all remaining bits to 0. Following this procedure, provers can confidentially compile the private output as $\mathsf{memval} \leftarrow \sum_{i=0}^{n-1} s_i \cdot \mathsf{memval}_i$. Furthermore, to minimize overhead, this circuit is employed

Definition A.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:*

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

  R denotes the collection of valid RAM 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 $O(|\mathbf{s}|)$.*
- *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* $\mathsf{Compute}$ *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 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* $\mathsf{out}_{\mathsf{priv}} \leftarrow \varnothing$*.*

exclusively in MPC and not for zero-knowledge proofs. The correctness of this step is verified within the trace verification circuit.

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

This functionality interacts with a set of prover servers $S_1, S_2, \ldots, S_m$ and an adversary $\mathcal{S}$. Let corr be the set of corrupted prover servers.

Upon receiving $(\text{MemoryFetch}, [\mathbf{m}], [\mathsf{addr}])$ from all servers, do:

- Receive shares $\{u_{i,j}\}_{i \in \mathsf{corr}}$ from adversary $\mathcal{S}$, where $u_{i,j}$ refers to the j-th share in field multiplication, Π_{LowBit} protocol, and $\Pi_{\text{EqualityCheck}}$ protocol.
- Reconstruct $\mathbf{m}$ from $[\mathbf{m}]$ and addr from $[\mathsf{addr}]$.

- Let m be the row in $\mathbf{m}$ whose memory address matches addr. If there are multiple rows that match, choose the one with the highest global step counter.
- Sample random sharing $[m]$ of m, such that the shares of the corrupted servers are identical to those received from the adversary $\mathcal{S}$.
- Distribute $[m]$ to all servers.

Protocol $\Pi_{\text{MemoryFetch}}$

This protocol interacts with a set of prover servers $S_1, S_2, \ldots, S_m$.

Upon receiving $(\text{MemoryFetch}, [\mathbf{m}], [\mathsf{addr}])$ from all servers, do:

- For each row $i \in [1, |[\mathbf{m}]|]$, compute $[s_i] \leftarrow \Pi_{\text{EqualityCheck}}([\mathsf{addr}], [\mathsf{addr}_i])$.
- Denote $\mathbf{s} = (s_{n-1}, s_{n-2}, \ldots, s_0)$.
- Perform a LowBit computation Π_{LowBit} on $[\mathbf{s}]$ to obtain $[\mathbf{s}']$.
- Extract $s_{n-1}, s_{n-2}, \ldots, s_0$ from $[\mathbf{s}']$.
- Compute and output $[m] \leftarrow \sum_{i=1}^{n} [s'_i] \cdot [m_i]$.

Theorem A.6. *The protocol $\Pi_{\text{MemoryFetch}}$ securely realizes $\mathcal{F}_{\text{MemoryFetch}}$ against a semi-honest adversary corrupting up to t servers.*

Proof. The correctness follows from the construction directly. This protocol does not directly involve collaborative zkSNARKs – the zero-knowledge property directly holds since it's not included in the proof while the soundness will be ensured in the trace verification protocol. The t-distributed-obliviousness holds since besides (a) all values are secret shares, and (b) communications involved by field multiplications, the Π_{LowBit} protocol and the $\Pi_{\text{EqualityCheck}}$ protocol as CompatCircuit gadgets, the protocol $\Pi_{\text{MemoryFetch}}$ only perform local computations. Therefore, corrupted servers cannot learn any information about the input. The t-distributed-obliviousness is thereby guaranteed. □

3) Instruction Execution: Given global step counter t as a public input, public/private input values $\mathsf{in}_{\mathsf{pub}}$ and $\mathsf{in}_{\mathsf{priv}}$, along with the following private inputs: an instruction I, general-purpose registers $\mathbf{R}$, the program counter register pc, a fetched memory value memval, the instruction execution circuit generates outputs that include: updated register values $\mathbf{R}'$ and pc', a new memory trace row to be inserted into the memory trace table, and public/private output values $\mathsf{out}_{\mathsf{pub}}$ and $\mathsf{out}_{\mathsf{priv}}$. IO values are set to 0 if the instruction type mismatch.

This circuit processes several types of instructions:

- Register operations, which involve moving, swapping, or shifting general-purpose registers.
- Conditional jump. Program counter pc typically increases itself by one. In conditional jump instruction, if the general-purpose register specified in the instruction holds a zero value, the program counter is set to a different address.
- Computations that involve one or two general-purpose registers as inputs and update the result in another specified register. These computations include arithmetic operations, comparisons, equality checks, boolean operations, and MiMC hash [2].

Implementing these instruction types in CompatCircuit requires skills similar to those used in typical zkSNARKs development, including private index accessing, conditional variable assigning, as well as other CompatCircuit gadgets. Specific implementation details are omitted here for brevity. We list the instruction set in Table IV.

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

This functionality interacts with a set of prover servers $S_1, S_2, \ldots, S_m$ and an adversary $\mathcal{S}$. Let corr be the set of corrupted prover servers.

Upon receiving $(\text{InstructionExecution}, [\mathsf{in}_{\mathsf{pub}}], [\mathsf{in}_{\mathsf{priv}}], [\mathsf{pc}], [\mathsf{memval}], [\mathsf{R}], [\mathsf{I}])$ from all servers, do:

- Receive shares $\{u_{i,j}\}_{i \in \mathsf{corr}}$ from adversary $\mathcal{S}$, where $u_{i,j}$ refers to the j-th share in CompatCircuit primitives and gadgets.
- Reconstruct $\mathsf{in}_{\mathsf{pub}}$, $\mathsf{in}_{\mathsf{priv}}$, pc, memval, $\mathbf{R}$, and I from $[\mathsf{in}_{\mathsf{pub}}]$, $[\mathsf{in}_{\mathsf{priv}}]$, $[\mathsf{pc}]$, $[\mathsf{memval}]$, $[\mathbf{R}]$, and $[\mathsf{I}]$, respectively.
- Let I be the instruction type specified in I. Simulate the semantic of I and create the outputs $\mathbf{R}'$, pc', memval', $\mathsf{out}_{\mathsf{pub}}$, and $\mathsf{out}_{\mathsf{priv}}$.
- Sample random sharing $[\mathbf{R}']$, $[\mathsf{pc}']$, $[\mathsf{memval}']$, $[\mathsf{out}_{\mathsf{pub}}]$, and $[\mathsf{out}_{\mathsf{priv}}]$, such that the shares of the corrupted servers are identical to those received from the adversary $\mathcal{S}$.
- Distribute $[\mathbf{R}']$, $[\mathsf{pc}']$, $[\mathsf{memval}']$, $[\mathsf{out}_{\mathsf{pub}}]$, and $[\mathsf{out}_{\mathsf{priv}}]$ to all servers.

Protocol $\Pi_{\text{InstructionExecution}}$

This protocol interacts with a set of prover servers $S_1, S_2, \ldots, S_m$.

Upon receiving $(\text{InstructionExecution}, [\mathsf{in}_{\mathsf{pub}}], [\mathsf{in}_{\mathsf{priv}}], [\mathsf{pc}], [\mathsf{memval}], [\mathbf{R}], [\mathsf{I}])$ from all servers, do:

- Enumerate all possibilities of different types of $[I]$. For each possibility, simulate the semantic of the instruction type and create the outputs $\mathbf{R}'$, pc', memval', $\mathsf{out}_{\mathsf{pub}}$, and $\mathsf{out}_{\mathsf{priv}}$.
- Compose each final output value with exclusive possible output values with a secret bit indicating whether the instruction type matches.

Theorem A.7. *The protocol $\Pi_{\text{InstructionExecution}}$ securely realizes $\mathcal{F}_{\text{InstructionExecution}}$ against a semi-honest adversary corrupting up to t servers.*

Proof. The correctness follows from the construction directly. The soundness holds from the underlying CompatCircuit primitives and gadgets. The zero-knowledge holds since beyond the instances $\mathsf{in}_{\mathsf{pub}}$ and $\mathsf{out}_{\mathsf{pub}}$, all other values act as witnesses, ensuring zero-knowledge property provided

TABLE IV
VDORAM INSTRUCTION SET

Instruction Type	Type Visibility	Operands	Semantics
`Halt`	Public	-	Terminate program execution
`PublicInput`	Public	#0	Register #0 $\leftarrow$ External public input
`PrivateInput`	Public	#0	Register #0 $\leftarrow$ External private input
`PublicOutput`	Public	#0	External public output $\leftarrow$ Register #0
`Shift`	Private	-	Cyclic right shift all registers
`Swap`	Private	#0 #1	Swap values of Register #0 and Register #1
`Move`	Private	#0 #1	Register #0 $\leftarrow$ Register #1
`Set`	Private	#0 #1	Register #0 $\leftarrow$ Immediate value #1
`Noop`	Private	-	No operation instruction
`JumpIfZero`	Private	#0 #1	Conditional jump. When Register #0 $=0$, set program counter to immediate value #1
`Store`	Private	#0	Memory write. Write value of Register #0 to memory address indicated by address register
`Load`	Private	#0	Memory read. Read from memory address indicated by address register, store value to Register #0
`Hash`	Private	#0 #1	MiMC hash computation. Register #0 $\leftarrow \mathcal{H}$(Register #1)
`Add`	Private	#0 #1 #2	Field addition. Register #0 $\leftarrow$ Register #1 + Register #2
`Sub`	Private	#0 #1 #2	Field subtraction. Register #0 $\leftarrow$ Register #1 $-$ Register #2
`Mul`	Private	#0 #1 #2	Field multiplication. Register #0 $\leftarrow$ Register #1 $\times$ Register #2
`Inv`	Private	#0 #1	Field inversion. Register #0 $\leftarrow$ Register #1^{-1}
`Norm`	Private	#0 #1	Field zero check. When Register #1 $=0$, Register #0 $\leftarrow 0$, otherwise Register #0 $\leftarrow 1$
`And`	Private	#0 #1 #2	Logical AND. Register #0 $\leftarrow$ Register #1 $\wedge$ Register #2
`Or`	Private	#0 #1 #2	Logical OR. Register #0 $\leftarrow$ Register #1 $\vee$ Register #2
`Xor`	Private	#0 #1 #2	Logical XOR. Register #0 $\leftarrow$ Register #1 $\oplus$ Register #2
`Not`	Private	#0 #1	Logical NOT. Register #0 $\leftarrow \neg$Register #1
`LessThan`	Private	#0 #1 #2	Field less-than operation. When Register #0 $<$ Register #1, Register #0 $\leftarrow 1$, otherwise Register #0 $\leftarrow 0$

by CompatCircuit primitives and gadgets. The t-distributed-obliviousness holds since (a) besides I/O values $\mathsf{in}_{\mathsf{pub}}$, $\mathsf{in}_{\mathsf{priv}}$, $\mathsf{out}_{\mathsf{pub}}$, $\mathsf{out}_{\mathsf{priv}}$ ($\mathsf{in}_{\mathsf{priv}}$ and $\mathsf{out}_{\mathsf{priv}}$ are revealed among provers while kept secretly as witnesses in the proof), all other values are secret shares, and (b) besides communications involved by CompatCircuit primitives and gadgets, the $\Pi_{\text{InstructionExecution}}$ protocol only perform local computations. Therefore, corrupted servers cannot learn any information about the input. The t-distributed-obliviousness is thereby guaranteed. □

4) Trace Sort: A trace sort circuit accepts a memory trace table $\mathbf{m}$ as a private input and outputs the table in the required order as a private output. Similar to the memory fetch circuit, this circuit is also designed exclusively for MPC purposes.

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

This functionality interacts with a set of prover servers $S_1, S_2, \ldots, S_m$ and an adversary $\mathcal{S}$. Let corr be the set of corrupted prover servers.

Upon receiving $(\text{TraceSort}, [\mathbf{m}])$ from all servers, do:

- Receive shares $\{u_{i,j}\}_{i\in\mathsf{corr}}$ from adversary $\mathcal{S}$, where $u_{i,j}$ refers to the j-th share in field multiplications, Π_{LessThan} protocol, and $\Pi_{\text{EqualityCheck}}$ protocol.
- Reconstruct $\mathbf{m}$ from $[\mathbf{m}]$.
- Sort $\mathbf{m}$ firstly by memory address and then by global step counter, denoted as $\mathbf{m}'$.
- Sample random sharing $[\mathbf{m}']$, such that the shares of the corrupted servers are identical to those received from the adversary $\mathcal{S}$.
- Distribute $[\mathbf{m}']$ to all servers.

Protocol $\Pi_{\text{TraceSort}}$

This protocol interacts with a set of prover servers $S_1, S_2, \ldots, S_m$.

Upon receiving $(\text{TraceSort}, [\mathbf{m}])$ from all servers, do:

- Pad the table $[\mathbf{m}]$ so its number of items is a power of two, with all values be the maximum $p-1 \in \mathbb{F}_p$. Denote the padded table as $\mathbf{m}_{\mathsf{pad}}$, and the number of padding rows as n_{pad}.
- Sort the padded table $\mathbf{m}_{\mathsf{pad}}$ with Bionic sort, using a custom comparer $m_1 < m_2 \iff (\mathsf{addr}_1 < \mathsf{addr}_2) \vee ((\mathsf{addr}_1 = \mathsf{addr}_2) \wedge (t_1 < t_2))$, denoted as $\mathbf{m}_{\mathsf{sorted}}$.
- Remove the last n_{pad} rows from $\mathbf{m}_{\mathsf{sorted}}$, denoted as $\mathbf{m}'_{\mathsf{sorted}}$.
- Output $\mathbf{m}'_{\mathsf{sorted}}$.

Theorem A.8. *The protocol $\Pi_{\text{TraceSort}}$ securely realizes $\mathcal{F}_{\text{TraceSort}}$ against a semi-honest adversary corrupting up to t servers.*

Proof. The correctness follows from the construction of Bitonic sorting and $p-1$ is the maximum element in $\mathbb{F}_p$. This

protocol does not directly involve collaborative zkSNARKs – the zero-knowledge property directly holds since it's not included in the proof while the soundness will be ensured in the trace verification protocol. The t-distributed-obliviousness holds since (a) all values are secret shares, and (b) besides communications involved by field multiplications and the Bitonic sort implemented with the Π_{LessThan} protocol and $\Pi_{\text{EqualityCheck}}$ protocol as CompatCircuit gadgets, the protocol $\Pi_{\text{TraceSort}}$ only perform local computations. Therefore, corrupted servers cannot learn any information about the input. The t-distributed-obliviousness is thereby guaranteed. □

5) Trace Verification: A trace verification circuit takes a memory trace table $\mathbf{m}$ as a private input and returns a bit b, indicating whether the specified restrictions are met. The checking rules are already introduced in Equation 1, Sec. IV-C.

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

This functionality interacts with a set of prover servers $S_1, S_2, \ldots, S_m$ and an adversary $\mathcal{S}$. Let corr be the set of corrupted prover servers.

Upon receiving $(\text{TraceVerification}, [\mathbf{m}])$ from all servers, do:

- Receive shares $\{u_{i,j}\}_{i \in \mathsf{corr}}$ from adversary $\mathcal{S}$, where $u_{i,j}$ refers to the j-th share in field multiplications, Π_{LessThan} protocol, and $\Pi_{\text{EqualityCheck}}$ protocol.
- Reconstruct $\mathbf{m}$ from $[\mathbf{m}]$.
- For each row $i \in [1, |\mathbf{m}| - 1]$, compute the boolean result b_i of Equation 1.
- Compute the final result $r \leftarrow b_0 \wedge b_1 \wedge \cdots \wedge b_{|\mathbf{m}|-1}$.
- Sample random sharing $[r]$, such that the shares of the corrupted servers are identical to those received from the adversary $\mathcal{S}$.
- Distribute $[r]$ to all servers.

Protocol $\Pi_{\text{TraceVerification}}$

This protocol interacts with a set of prover servers $S_1, S_2, \ldots, S_m$.

Upon receiving $(\text{TraceVerification}, [\mathbf{m}])$ from all servers, do:

- For each row $i \in [1, |\mathbf{m}| - 1]$, compute the boolean result $[b_i]$ of Equation 1 using field multiplications, Π_{LessThan} protocol, and $\Pi_{\text{EqualityCheck}}$ protocol.
- Compute the final result $[r] \leftarrow [b_0] \wedge [b_1] \wedge \cdots \wedge [b_{|\mathbf{m}|-1}]$.
- Output r.

Theorem A.9. *The protocol $\Pi_{\text{TraceVerification}}$ securely realizes $\mathcal{F}_{\text{TraceVerification}}$ against a semi-honest adversary corrupting up to t servers.*

Proof. The correctness follows from the construction directly. For the soundness, $(\mathsf{addr}_i < \mathsf{addr}_{i+1}) \vee ((\mathsf{addr}_i = \mathsf{addr}_{i+1}) \wedge (t_i < t_{i+1}))$ ensures the soundness of protocol $\Pi_{\text{TraceSort}}$, where an unsorted trace would violates the constraint. The rest two constraints ensures the soundness of memory operation in protocol $\Pi_{\text{InstructionExecution}}$: (a) $(\mathsf{addr}_i \neq \mathsf{addr}_{i+1}) \vee (\mathsf{val}_i = \mathsf{val}_{i+1}) \vee (\mathsf{op}_{i+1} = \texttt{store})$ ensures the constrant that unless the memory address is increased or a store instruction is involved, the memory value must not be changed, and (b) $(\mathsf{addr}_i = \mathsf{addr}_{i+1}) \vee (\mathsf{op}_{i+1} = \texttt{store})$ ensures if a new memory address is used, the first instruction must be store. Therefore, the soundness holds from the constraints above and the soundness property provided by CompatCircuit primitives and gadgets. The zero-knowledge holds because besides the instance r, the involved values are witnesses, ensuring zero-knowledge property provided by CompatCircuit primitives and gadgets. The t-distributed-obliviousness holds since (a) besides the public output r, other values are secret shares, and (b) besides communications involved by field multiplications and the Bitonic sort implemented with the Π_{LessThan} protocol and $\Pi_{\text{EqualityCheck}}$ protocol as CompatCircuit gadgets, the protocol $\Pi_{\text{TraceVerification}}$ only perform local computations. Therefore, corrupted servers cannot learn any information about the input. The t-distributed-obliviousness is thereby guaranteed. □

Finally, after performing security analysis on the dependencies of VDORAM protocols, we can prove the security of the VDORAM protocol Π_{VDORAM}.

Proof of Theorem A.4. The VDORAM protocol Π_{VDORAM} is constructed as a composition of the protocols $\Pi_{\text{MemoryFetch}}$, $\Pi_{\text{InstructionExecution}}$, $\Pi_{\text{TraceSort}}$, and $\Pi_{\text{TraceVerification}}$, as well as basic CompatCircuit primitives and gadgets. The completeness, soundness, zero-knowledge properties follow from these protocols. Besides calling these protocols, servers only do local computations. Therefore, the VDORAM protocol is secure against a semi-honest adversary corrupting up to t servers, demonstrating the t-distributed-obliviousness property. □

D. Artifact Appendix

The artifact provides the source code for VDORAM, the first publicly verifiable distributed oblivious RAM, and its underlying framework, the first multi-prover ZKP frontend CompatCircuit designed for Collaborative zkSNARKs. The implementation includes the C# code for the multi-prover ZKP front-end and the VDORAM runtime, along with a modified Rust-based collaborative zk-SNARK back-end. The artifact contains all necessary codes and scripts to reproduce the performance evaluation presented in the paper.

1) Description & Requirements: This section lists the information necessary to recreate the experimental setup used for the artifact evaluation.

How to access. The artifact is available in a public Git repository at https://github.com/BDS-SDU/vdoram-artifacts. It is also available on Zenodo: https://doi.org/10.5281/zenodo.15855167.

Hardware dependencies. The experiments can be run on a single machine that simulates all parties. The recommended hardware specifications are:

- CPU: 8 or more cores
- Memory: 16 GB or more
- Disk: 40 GB of free space

Software dependencies. The artifact can be tested on Ubuntu Server 24.04 LTS. The following software is required:

- .NET SDK 8.0 (e.g., 8.0.117)
- Rust nightly toolchain (e.g., 1.90.0-nightly)
- Standard build tools: `build-essential`, `git`, `curl`, `openssh-client`, `dos2unix`

All dependencies can be installed using the system's package manager (`apt`) and `rustup`, as detailed in the `README.md`.

2) Artifact Installation & Configuration: The installation process involves setting up the required software dependencies, cloning the repository, and compiling the projects.

1) Install the software dependencies listed above using `apt` and `curl` as detailed in the `README.md` file.
2) Clone the artifact repository from the provided URL.
3) Compile the C# projects by running `dotnet restore` and `dotnet build -c Release` in the `CompatCircuit` directory.
4) Compile the Rust-based ZKP client by running `cargo build --release --bin client` in the `collaborative-zksnark-mod/mpc-snarks` directory.

Detailed commands are provided in the `README.md` file.

3) Experiment Workflow: The experimental workflow is managed by a series of shell scripts. The high-level process is as follows:

1) **Configuration:** Select an experiment configuration (e.g., number of parties) by editing the `config.sh` file.
2) **Distribution:** Run scripts to build the executables and distribute them to the designated locations for the experiment. For the single-machine evaluation, this is all handled locally.
3) **Execution:** Run the main experiment script. This will execute the three main stages of the evaluation: the preprocessing stage, the computation stage and the proof generation & verification stage.
4) **Monitoring & Collection:** Monitor the experiment's progress by tailing the log files. Once complete, collect the resulting logs and R1CS files for analysis.

4) Major Claims: Our paper makes the following major claims, which can be validated by the provided artifact.

- (C1): **CompatCircuit Performance.** Our CompatCircuit framework is a front-end for multi-prover ZKPs. Its core primitives exhibit reasonable computation and proof generation overheads that scale predictably with the number of parties. This is proven by experiment (E1), with results corresponding to Figure 7 in the paper.
- (C2): **VDORAM Performance.** Our VDORAM implementation's performance is consistent across different instruction types (demonstrating obliviousness), and its cost scales linearly with the number of instruction cycles. This is proven by experiment (E1), with results corresponding to Figure 8 in the paper.
- (C3): **VDORAM Scalability.** The overall overhead of VDORAM's distributed computation and proof generation remains moderate when scaling the number of parties, as compared to a single-prover baseline. This is proven by experiment (E1), with results corresponding to Figure 9 in the paper.

5) Evaluation: This section describes the single, comprehensive experiment (E1) to run, which validates all major claims.

Experiment (E1) [End-to-End Performance Evaluation] [2 human-hours + 20 compute-hours]: This experiment validates claims C1, C2, and C3 by running the full evaluation pipeline. It measures the performance of CompatCircuit primitives and the complete VDORAM system for different party counts and workloads.

[How to] The experiment is divided into three main stages: the preprocessing stage, the computation stage and the proof generation & verification stage. The following steps should be repeated for each party count configuration.

[Preparation] Follow the installation and configuration steps in the `README.md` to prepare the environment. This includes installing dependencies, compiling all projects, and ensuring the machine can SSH to itself for the scripts to work correctly.

[Execution]

1) **Preprocessing Stage:** Run the preprocessing script (`run-preprocess.sh`) to measure the time cost of MPC preprocessing.
2) **Computation Stage:** For a given party count N, copy `config-e1-nN.sh` to `config.sh`. Run the scripts to build, distribute, and prepare the experiment files as described in the `README.md`. Execute the computation benchmarks using `7a-exp-123-single.sh` (for $N = 1$) or `7b-exp-123-mpc-thread.sh` (for $N > 1$). After each run, use the `download-remote-log.sh` and `download-remote-r1cs.sh` scripts to collect the results. Remember to clear the generated logs and R1CS files using the provided `clear-remote-*.sh` scripts before switching to a different party count configuration.
3) **Proof Generation & Verification Stage:** After the computation stage for a given party count N is complete, navigate to the `prove-scripts` directory. Set the `PARTY_COUNT` in `config.sh` to N. Run `prove-r1cs-single.sh` (for $N = 1$) or `prove-r1cs-multiparty.sh` (for $N > 1$). This stage will use the R1CS files generated in the previous step to measure ZKP setup, proving, and verification times.

[Results] The timing information in the log files contains the raw data for the figures in the paper.

- The logs from the **Preprocessing Stage:** provide the data for the Preprocess portion of Figure 9.
- The logs from the **Computation Stage** provide the data for the Compute portion of Figures 7 and 9, and all of Figure 8.
- The logs from the **Proof Generation & Verification Stage** provide the data for the ZKP Prove portion of

Figure 7 and ZKP Setup/Prove/Verify portion of Figure 9.

By plotting the collected data, one can verify that the performance trends match those presented in the paper, thereby validating claims C1, C2, and C3.

6) Notes: A few final notes are included to assist with the evaluation process.

- Check the Artifact Evaluation Step-By-Step Guide (`CompatCircuit/Experiments/README.md`) for all detailed instructions.
- In the source code, VDORAM has a code name `CollaborativeZkVm`. The `CompatCircuit` folder contains source codes for both CompatCircuit and VDORAM.
- To accelerate the artifact evaluation process, we have trimmed down the experiment scale. The approach to perform a complete evaluation is also mentioned in the `README.md` file above.