

A Low-Communication Garbled RAM from Homomorphic Secret Sharing

Chase Fickes
University of Virginia
muj2bn@virginia.edu

Jinye He
University of Virginia
qfn5bh@virginia.edu

Wei-Kai Lin
University of Virginia
wklin@virginia.edu

Abstract

Garbled circuits are fundamental in modern cryptography and secure two- or multi-party computation. For real-world programs that are naturally expressed in the Random-Access Machine (RAM) model, garbled RAM is the RAM counterpart of garbled circuits: they avoid the cost of compiling the entire program into a circuit, with communication complexity serving as the primary efficiency metric. We study the setting in which the RAM program is public, while the input data and memory contents remain secret except for the prescribed output.

We construct a garbled RAM for programs running in time T over memory consisting of N words of W bits each, the scheme achieves

$$O(T \cdot (W + \lambda) \cdot \log N) + \text{poly}(\lambda)$$

bits in communication, where λ is the security parameter. For sufficiently large word size $W \geq \lambda$ and running time $T \geq \text{poly}(\lambda)$, the communication becomes $O(TW \log N)$, asymptotically matching the bandwidth of an optimal oblivious RAM. Since any secure garbled RAM must hide memory accesses as in oblivious RAMs, our construction can be viewed as compiling an oblivious RAM into a non-interactive analogue, without incurring additional asymptotic communication cost.

Technically, our garbled RAM builds on the recent succinct garbled circuits of Ishai, Li, and Lin (Crypto'25) and Li, Lin, and Lu (Eurocrypt'26), which are in turn based on homomorphic secret sharing. Accordingly, we inherit their circular-power variants of the Decisional Diffie-Hellman or Ring Learning-With-Errors assumptions. We also use techniques from the recent work on garbled arithmetic RAM, called Zebra, by Gu, Ghoshal, and Shi (Eurocrypt'26). Compared to Zebra, which achieves the same $O(TW \log N)$ communication for *arithmetic* RAM programs that only perform arithmetic over large and bounded integers, our work focuses on the standard *boolean* RAM model. Compared to the best garbled RAM in the random oracle model, due to Liu, Liu, Luo, and Heath (ACM CCS'26), we remove a multiplicative factor λ in communication.

Keywords: Succinct garbling, Random-access machines, Ring learning-with-errors.

Contents

1	Introduction	1
1.1	Our Results	2
1.2	Related Works	4
2	Overview of Previous Techniques	5
2.1	Garbled RAM from ORAM	5
2.2	Dynamic Circuits with Route	7
2.3	Succinct Garbled Circuits and HGate	8
3	Overview of Our Garbled RAM	9
3.1	Concepts for Garbled Route from Homomorphic Secret Sharing	10
3.2	Procedure of Garbled Route	13
3.3	Garbled RAMs from Garbled Route	15
4	Preliminary	17
4.1	Definition of Garbling	17
4.2	Hardness Assumptions	18
4.3	Homomorphic Tools	20
4.4	Homomorphic Gate	21
4.5	Dynamic Circuit	24
5	Constructing Garbled Dynamic Circuits	24
5.1	Routing Gate	24
5.2	Garbling Dynamic Circuits	29
5.3	Instantiations Under Paillier Groups	30
5.4	Instantiations Under Prime-Order Groups	35
5.5	Instantiations Under Lattices	37
6	Garbling Bucket using Route	39
6.1	Background: Perfect ORAM	40
6.2	Garbling a Perfect ORAM	41
7	Garbled RAM from Route and Bucket	43
7.1	Simulate RAM Programs using Dynamic Circuits	43
7.2	Cost of Garbled RAMs	50
8	Optimizations to Fixed Factor Hashing	52
8.1	Hashing under DDH	53
8.2	Hashing under Lattices	55
A	Appendix	66
A.1	Perfect ORAM Algorithms	66
B	Disclosure of AI Usage	68

1 Introduction

Garbled circuits have been a central and extensively studied cryptographic primitive since the seminal work of Yao [Yao86, BMR90, NPS99, KS08, BHR12, ZRE15, RR21]. A garbling scheme involves a garbler, Alice, and an evaluator, Bob. Alice samples a garbling key sk , and when given a circuit C and an input inp , using sk , Alice garbles C into a garbled circuit $\tilde{C}$ and then encodes inp into garbled input $\tilde{\text{inp}}$. Given $\tilde{C}$ and $\tilde{\text{inp}}$, Bob evaluates the computation and recovers $C(\text{inp})$. Informally, security requires that Bob learns $C(\text{inp})$ and nothing else about inp . We require that $\tilde{C}$ is independent of inp and that $\tilde{\text{inp}}$ is independent of C to be non-trivial.

The garbled Random-Access Machine (RAM) was introduced to improve efficiency for computations that can be represented efficiently as RAM programs [LO13, GHL⁺14, GLOS15, GLO15, LO17]. Garbled RAMs are functionally similar to garbled circuits, but garbled RAM considers computation represented by a RAM-model program P , where Alice garbles P into a garbled program $\tilde{P}$, and Bob evaluates $\tilde{P}$ and $\tilde{\text{inp}}$ to obtain the result $P(\text{inp})$. The security requirement is the same as garbled circuits: Bob learns only $P(\text{inp})$ but no more information about inp . Throughout this paper, we suppose that the program P (and analogously the circuit C) is public and known to Bob; in general, if P consists of Alice’s secrets, Alice can instead put her secret in the input inp and garble another public program P' that reads the secret from inp .

Communication efficiency: succinctness of garbled RAMs. The main efficiency metric for garbling schemes, for both RAMs and circuits, is communication, often dominated by the lengths $|\tilde{P}|$ or $|\tilde{C}|$. Following Yao’s scheme [Yao86], many garbled circuits achieve $O(|C| \cdot \lambda)$ -bit communication [KS08, BHR12, ZRE15, RR21], where $|C|$ is the number of gates in the circuit, and λ is the security parameter. As mentioned, garbled RAM aims to achieve better communication than garbled circuits. Consider a RAM program P that, for a given input size, runs in time T and uses a memory of N words, where each word is W bits. A naive baseline to garble P is to construct and then garble an equivalent circuit of size $O(TNW)$, which takes communication $O(TNW\lambda)$. Garbled RAM requires non-trivial communication that is *sublinear* in N [LO13].

The literature builds two kinds of garbled RAMs (GRAMs), under various cryptographic assumptions. The first kind, called *succinct* GRAMs [CH16, CCC⁺16, ACC⁺16, CCHR16, AL18], requires communication to be $O(|P| \cdot \text{poly}(\lambda, \log(TNW)))$, i.e., linear in the description length of P . However, known succinct GRAMs rely on very strong assumptions, such as indistinguishability obfuscation (iO) [BGI⁺01, JLS21, JLS22, RVV24], and the running time of both the garbler and evaluator is practically prohibitive. The second kind of GRAMs requires no succinctness [LO13, GHL⁺14, GLOS15, GLO15, LO17, HKO22, PLS23, HKO23, YPHK23, GTRS25, GGS26]. They are based on weaker assumptions, e.g., one-way functions, and achieve practicality and take more communication $O(TW \cdot \text{poly}(\lambda, \log N))$. In this line, the GRAM of Liu, Liu, Luo, and Heath [LLH26] achieves $O(T(W + \log^2 N) \log N \cdot \lambda) \cdot \omega(1)$ communication in the random oracle model, or alternatively from circular-correlation robust hash [KS08, CKKZ12, BHKR13].

A natural question is to minimize the communication of GRAMs under weak assumptions. Recall the recent advances on garbled circuits, which achieve 1-bit communication per boolean gate without the multiplicative factor λ and are thus called *succinct* garbled circuits [MORS24, LWYY25b, ILL25b, LWYY25a, MORS25, ILL25a]. An ideal GRAM would be TW -bit communication for any program that runs in time T and on W -bit words. As in succinct garbled circuits, we focus on

assumptions that are unknown to imply iO.

Can we construct garbled RAMs that achieve $O(TW)$ -bit communication, and ideally $TW + \text{poly}(\lambda)$ bits, under assumptions weaker than indistinguishable obfuscation?

To answer the question, Gu, Ghoshal, and Shi recently proposed a GRAM, called Zebra, and achieved $O(TW(\lambda + \log N))$ for word size $W \geq \lambda \cdot \log^\epsilon N$ for any constant ϵ [GGS26].¹ The work is based on the Decisional Composite Residuosity (DCR) assumption [Pai99, DJN10] and achieves the lowest known communication without iO. Still, $O(TW \cdot \lambda)$ is far from TW because the DCR security parameter λ is typically more than a thousand. We would also want smaller word sizes, ideally $W = \Omega(\log N)$ as in the standard RAM model. Moreover, given that DCR is a group-based assumption that is broken by quantum attacks such as Shor’s Algorithm [Sho94], another question is how to get low-communication GRAM with plausible post-quantum security. Thus, in addition to communication, we also want smaller words and post-quantum security.

1.1 Our Results

We construct garbled RAMs that achieve $O(TW \cdot \log N)$ -bit communication, without the multiplicative security parameter. Our garbled RAM can be based on either the Circular Power-Decisional Diffie Hellman (CP-DDH) or the Circular Power-Ring Learning with Errors (CP-RLWE) assumptions, previously proposed by Ishai, Li, and Lin [ILL25b] to obtain succinct garbled circuits.

Theorem 1.1 (Garbled RAM, informal). *Assume the existence of pseudorandom functions (PRFs) in NC1, and assume any one of the following assumptions: (1) Circular Power-DDH in Paillier groups, or (2) Circular Power-DDH in prime-order groups, or (3) Circular Power-RLWE. There exists a garbled RAM scheme with communication $O(T \cdot (W + \lambda) \cdot \log N) + W^2 \cdot \text{poly}(\lambda)$ bits for any RAM program that runs in time T and on N -word memory, where each word consists of W bits. The security parameter λ is the bit-length of either a group or a ring element that depends on the assumption. The computation time of both the garbler and evaluator is $O(TW^2 \cdot \log N \cdot \text{poly}(\lambda))$.*

To compare the communication, an ideal scenario is oblivious RAM (ORAM) [Gol87, Ost90, GO96]. ORAM can be viewed as an *interactive protocol* of garbled RAM, where Bob passively holds the encrypted RAM memory, and Alice performs the RAM program through Bob, who retrieves from and updates the memory. To ensure that Bob learns only the output in GRAM, the memory accesses performed by Alice, which is the ORAM, must be computationally indistinguishable for all inputs. In garbled RAM, Bob’s evaluation implicitly simulates an ORAM scheme, and thus the ORAM lower bound applies to Bob’s evaluation time. The overhead of ORAM is well-studied and proven to be $\Theta(\log N)$ [GO96, WCS15, LN18, AKL⁺20, AKLS21], i.e., we need $\Theta(\log N)$ memory accesses to hide one original insecure access. Looking at our GRAM, when the word size $W \geq \lambda$ is sufficiently large, the communication is $O(TW \cdot \log N)$ bits, which is asymptotically the same as that of the interactive and weaker ORAM protocol. Hence, although not yet $T \cdot W$ bits, our $O(TW \cdot \log N)$ -bit scheme is a significant milestone.

Our construction is based on the techniques of Ishai, Li, and Lin [ILL25b], who constructed succinct garbled circuits from Homomorphic Secret Sharing (HSS) [BGI⁺18]. Their succinct garbled circuits

¹For smaller words of $W < \lambda \cdot \log^\epsilon N$, the construction achieves communication $O(T \cdot (W + \lambda \cdot \log^\epsilon N)(\lambda + \log N))$ bits. Alternatively, when garbling a RAM program that performs only $\Theta(\lambda)$ -bit arithmetic operations, their scheme achieves $O(TW \log N)$ communication for larger words of $W \geq \log^2 N \cdot \lambda$ bits [GGS26].

achieve one-bit communication per gate, and thus we were curious whether their techniques extend to GRAM. Our result shows that the HSS-based techniques indeed extend to the RAM model. Since RAM computation supports dynamic data accesses at unit cost while the circuit model does not, our work adds an additional technical gadget to the HSS toolbox; see the garbled **Route** below.

Our GRAM inherits the assumptions from succinct garbled circuits, namely CP-DDH or CP-RLWE. Moreover, we use a hash-then-extract construction that is modified from Li, Lin, and Lu [LLL26], and our use additionally requires circular variants of DDH or RLWE assumptions that are non-powered and over the same secret; the additional circular security is identical to that of encrypting a few secret-key bits circularly using the same encryption scheme (for the details, see Section 4.2). Our CP-RLWE-based scheme is potentially post-quantum secure.

The communication of our GRAM is asymptotically better than the results of Liu et al. [LLH26] and Gu et al. [GGS26], for any T, W, N such that $T \geq N$. Similar to the HSS-based succinct garbled circuits, this work is more on the theoretical feasibility, and we do not provide implementation. As succinct garbling is implemented very recently [ZLY⁺25, KLLL26], we believe our GRAM is implementable. Looking closely, our GRAM relies on a perfect ORAM to shave a super-constant factor, but it also introduces a large constant [CNS18, CSLN21]. A practical optimization can incur a super-constant asymptotic factor but get a smaller constant. We leave the optimization as an open question.

In Theorem 1.1 and throughout the paper, we abuse notation by using the security parameter λ to denote the key length associated with the assumption under consideration. Thus, for the same target security level, the value of λ depends on the underlying assumption. Namely, the GRAM of Liu et al. [LLH26] is based on the random oracle, so λ denotes the key size and is typically a few hundred bits in practice. In Zebra [GGS26] and in our DDH-based schemes, the assumption is either DCR, a variant of DCR, or a prime-order group, so λ is the bit-length of a group element, which is often a few thousand bits at a comparable security level. For our CP-RLWE-based scheme, λ is the bit-length of a ring element, which is often ten or even hundreds of KBs. Improving the concrete efficiency under each of these assumptions is an important future work.

Technical building block: garbled Route. An essential building block in our GRAM is a more efficient garbled Route gadget. Proposed in prior GRAMs [HKO23, GTRS25, GGS26, LLH26], the Route gadget permutes n input data items to output according to a given permutation $\pi = (\pi_1, \dots, \pi_n)$, i.e., for each $i \in [n]$, the input item a_i is assigned to the output b_{π_i} . The garbled Route works with garbled data items a_i but reveals the permutation π_i ; moreover, Route is required to be “eager evaluating.” That is, if both garbled values a_i and π_i are given, the output garbled value b_{π_i} can be evaluated, even if any other inputs are not yet ready.

Theorem 1.2 (Garbled Route, informal). *Assume the same assumption as per Theorem 1.1, and let λ be the same security parameter. There exists a garbled Route scheme with communication $O(\lambda + \ell)$ bits per item being routed, where n is the number of items being routed in the Route gadget, ℓ is the number of bits per item. In addition to the per-gadget communication, a one-time communication of $O(\ell^2 \cdot \text{poly}(\lambda))$ bits is needed but is reused among many instances of Route. The computation time per item is $O((\ell^2 + n\ell) \cdot \text{poly}(\lambda))$ for both the garbler and evaluator.*

Noticing that the communication is only $O(\ell + \lambda)$ bits when dynamically moving an ℓ -bit item, we find that Route gadget is of independent interest and is potentially enabling new optimizations or

applications.

Our Route takes asymptotically lower or equal communication than previous works for all n and ℓ . Previously, Zebra [GGS26] constructed Route and achieved $O((\lceil \ell/\lambda \rceil + n) \cdot \lambda)$ communication per item based on DCR, where λ is the bit-length of a DCR group member, and Zebra required that the ℓ -bit items are represented by $\Theta(\lambda)$ -bit integers. Hence, the Route of Zebra works with $\Theta(\lambda)$ -bit integers instead of bits, which incurs extra communication when bits are needed in the applications. Also recently, Liu et al. [LLH26] constructed a garbled Merge gadget based on random oracle, achieving $O(\ell \cdot n \cdot \lambda)$ communication. The functionality of Merge is a subset of Route—Merge sends items to two first-in-first-out piles, while Route sends items to an arbitrary permutation. Still, a garbled Merge would be sufficient for our GRAM as long as the Merge achieves $O(\ell + \lambda)$ -bit communication. If one found a more efficient Merge, we would likely get a more efficient GRAM; it is an important technical open question.

An optimization of fixed factor hashing [LLL26]. A technical challenge in garbled Route is that for each data item a_i , the garbler must commit to the output garbling labels at the destination π_i *without* knowing π_i . Zebra [GGS26] resolved this by having the garbler prepare routing material for *every* possible destination up front, but that incurred the multiplicative factor n in Zebra’s route. To this end, we let the evaluator *dynamically* and homomorphically compute the output garbling labels π_i using homomorphic secret sharing techniques. The dynamic computation incurs $O(\ell\lambda)$ -bit communication for each ℓ -bit item, which is unaffordable for our goal. Fortunately, we borrow a tool from Li, Lin, and Lu [LLL26], called fixed factor hashing (FFH), which compresses many garbling labels to a short hash. With that, it suffices to dynamically update the short hash according to π_i , and thus the communication is only the hash size. To the best of our knowledge, our work is the first application of FFH after Li, Lin, and Lu.

The Li-Lin-Lu fixed factor hashing takes $O(\lambda^2)$ bits per hash², where λ is the bit-length of a group or ring element, depending on the assumption. To achieve better communication, we lightly modified and optimized the hash size to a constant number of elements, which takes only $O(\lambda)$ bits. Our optimization could be useful in other scenarios, so we state it below. In our Route, the FFH extraction must output shares that are keyed by the same secret, so we need the assumptions below to be circular. In other applications, such as Li-Lin-Lu garbling, circular security is unnecessary.

Theorem 1.3 (Fixed factor hashing, informal). *Assume any one of the following assumptions: (1) circular DDH in Paillier groups, or (2) circular DDH in prime-order groups, or (3) circular RLWE. Let λ be the element size of the corresponding group or ring assumptions. There exists a fixed factor hashing that homomorphically compresses $\ell = \text{poly}(\lambda)$ authenticated homomorphic secret shares into $O(1)$ elements, and the authenticated shares can be homomorphically extracted. A reusable public data of $O(\ell^2 \cdot \text{poly}(\lambda))$ bits is needed.*

1.2 Related Works

In this work, we focus on the standard RAM model, in which the CPU performs boolean operations on two W -bit words in each time step. The prior work Zebra [GGS26] considers a different model, called *bounded arithmetic RAM*. In that model, each W -bit word is a vector of b -bit integers for some

²Recall that we denote λ as the bit-length of a group/ring element. This is used differently from the notation in Li, Lin, and Lu [LLL26] where λ denotes the exact security parameter.

$b = \Theta(\lambda)$ and thus consists of W/b coordinates. In each time step, the CPU performs coordinate-wise addition or multiplication on two such words. When $W \geq \lambda \cdot \log^2 N$, Zebra achieves $O(TW \cdot \log N)$ bits of communication for arithmetic RAM programs, thereby avoiding the multiplicative λ factor.

To see why Zebra incurs a λ factor for boolean RAMs [GGS26, Theorem 1.2], we succinctly recap their key idea here. Zebra garbles each b -bit integer into unit shares, enabling not only efficient bounded integer arithmetic but also efficient RAM accesses. However, to support boolean operations, it needs to bit-decompose each b -bit integer into a vector of b bits, incurring $O(b \cdot \lambda)$ -bit communication. The communication cost of bit-decomposition alone accounts for the λ factor. Eliminating this factor is explicitly stated as an open question in Zebra. In this work, we enable efficient RAM accesses directly on vectors of W bits, avoid bit-decomposition altogether, and resolve this open question.

Randomized encodings (RE) are closely related to garbled circuits [IK00, AIK04, App17]. In the RAM model, *succinct* randomized encodings are closely related to succinct garbled RAM [BGL⁺15]. Namely, a succinct RE encodes a RAM program P and an input x into an encoded program $\tilde{P}_x$, so that given $\tilde{P}_x$, the evaluator learns $P(x)$ but nothing more; the non-trivial succinctness requires that the encoding time must be much shorter than the computation time of $P(x)$. The succinctness is an extremely strong requirement, and thus earlier succinct REs are based on iO, and only recently, the work of Abram, Malavolta, and Roy [AMR25] proposed a scheme based on the decomposed learning with errors assumption. It remains open whether succinct RE or succinct GRAM can be based on weaker assumptions.

Our GRAM is based on the Circular Power-RLWE variant, which is probably stronger than the plain RLWE. Specifically, our variant of CP-RLWE assumption (see Definition 4.7) implies fully homomorphic encryptions (FHEs) [Gen09, BV11a, BGV12] and even RAM-FHE [HHWW19, LMW23]. A natural question is whether such CP-RLWE enables more efficient or simpler GRAMs via RAM-FHE and/or other RLWE-based primitives. To the best of our knowledge, it remains unclear how to obtain GRAM from FHE or even RAM-FHE.

2 Overview of Previous Techniques

In this section, we review previous techniques in garbled RAM and succinct garbled circuits. We suppose the readers are familiar with the garbling of boolean circuits [BHR12].

2.1 Garbled RAM from ORAM

We consider garbling schemes for programs in the following RAM model. The RAM program is defined by a circuit, often called CPU, that performs T steps of computation using a memory, which consists of N memory words, and each word is W bits. In each step, the CPU reads a word from any memory location to its local register, performs computation on two words in the register, and then writes a word to any memory location. The output of the computation is often defined to be stored in the memory. While some details are omitted, e.g., the size of the register, this model is sufficient for the discussion of garbled RAM. Indeed, because the CPU is a circuit, the CPU computation can be garbled by any garbled circuit scheme. Hence, the challenge of GRAM is to efficiently garble read and write accesses between CPU and memory.

In garbled circuits, an evaluator must perform the computation without learning any information about private inputs. In the case of RAM, the evaluator must determine which memory locations to access, and, if implemented naively, the access pattern may reveal information about the private inputs. Therefore, a secure garbled RAM should ensure that its access pattern is oblivious. A standard way to achieve this is to implement the RAM as an ORAM, which hides the memory locations accessed during the computation.

Tree-based ORAMs. Many GRAM schemes, including ours, use a tree-based ORAM [HKO22, PLS23, HKO23, GTRS25, GGS26, LLLH26], such as circuit ORAM [WCS15]. Tree-based ORAMs organize RAM into a binary tree, with each RAM address being associated with a root-to-leaf path in the tree. This structure is particularly useful for garbled RAM since the computation associated with each access can be decomposed into a sequence of small, per-node computations along the path, each of which can be implemented as a circuit. We provide an overview of tree-based ORAM and its structure which makes it convenient to garble.

Roughly speaking, tree-based ORAMs instantiate a balanced binary tree of $m = O(N)$ leaves to store N data elements. Each element is associated with a random leaf $p \in [m]$, also called position, and the element must be stored somewhere on the tree path from root to leaf p . The position p of any data element is resampled uniformly at random after each access of the element, and the accessed element is moved to the root node. To avoid overflow at the root node, an eviction procedure moves root elements toward their assigned leaf positions.

In the above procedure, a *random* tree path is accessed for each read or write operation on any data element. The random paths ensure a concentrated access pattern with overwhelming probability. That is, when performing Z operations of ORAM, the root must be visited Z times. Moreover, each of the left and right children of root will be visited $\Theta(Z/2) \pm \delta$ times with overwhelming probability for a sufficiently large $\delta = O(Z)$; one may choose $Z \geq \log^4 N$ and $\delta = Z/\log N$ to get a negligible failure probability in N . This concentration lets us bound the number of accesses that a circuit associated with a node must process, and this bound extends to all nodes.

We can now use this bound to implement the ORAM tree using circuits. For each tree node, we instantiate a circuit that takes $O(Z)$ input accesses from the parent node, outputs $O(Z/2)$ accesses to each of the left and right children nodes. The circuit then takes $O(Z/2)$ input accesses from each of the left and right children, and then outputs $O(Z)$ accesses back to the parent, supporting bidirectional access. The circuit also maintains the state of the tree node for at most $O(Z)$ visits. For any d , a node at tree depth d is visited $O(t/2^d)$ times in expectation over t ORAM accesses. Therefore, with overwhelming probability, a circuit at depth d can support $t_d = (Z - \delta) \cdot 2^d$ ORAM accesses. Thus, for every t_d accesses, we instantiate fresh circuits for all depth- d nodes and transfer each node's states to new circuits. That is, a single RAM access is represented by a collection of circuits across the ORAM tree. There are some details we leave out here; e.g., to ensure a seamless connection with the parent and children nodes, the new circuit overlaps by $O(\delta)$ in inputs and outputs. We refer to previous works for the details [HKO22, PLS23, HKO23].

Recall that a trivial but inefficient circuit implementation of a memory read could be to take a requested location of $\log N$ bits and all N memory words and then output the located W -bit word via a selection circuit. That leads to a huge garbling cost, and it motivates dynamic circuits. Technically speaking, the circuit in the above tree-ORAM implementation is a dynamic circuit by

the fact that they forward only the data required to the specified child but not both children. We review dynamic circuits next.

2.2 Dynamic Circuits with Route

Previous GRAM schemes address this inefficiency through the abstraction of *dynamic circuits* [HKO23]. Informally, in a dynamic circuit, the evaluation ordering of the gates depends not only on the circuit but also on the data being evaluated. Under the dynamic evaluation, gates whose inputs have not yet arrived remain unevaluated until those inputs arrive, allowing the computation to proceed only with the necessary gates for the execution. Reducing the number of gates also reduces the garbling cost.

In tree-based ORAM, access requests need to be forwarded from the parent to one of its children. Since the destination depends on the request, the circuit must be able to forward the request to the appropriate child node. This is the role of the special **Route** gate: to dynamically forward input items to specific output items according to the given targets.

More concretely, in addition to standard boolean gates (AND, OR, NOT), the Route gate defined below is allowed in dynamic circuits.³ A Route is parameterized by n and ℓ , and its functionality is:

- The input is a list of items $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$ and a list of targets $(\pi_1, \pi_2, \dots, \pi_n)$, where any item $\mathbf{x}_i$ is ℓ -bits, and any target π_i is $(\log n)$ -bits. We call each item to be routed as a *cable*, following prior work [GGS26]. We assume that n is a power of two for simplicity, and we require that the targets $(\pi_1, \pi_2, \dots, \pi_n)$ form a permutation of $[n]$.
- The output is the permuted cables, $(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n) = (\mathbf{x}_{\pi_1^{-1}}, \mathbf{x}_{\pi_2^{-1}}, \dots, \mathbf{x}_{\pi_n^{-1}})$, where π_i^{-1} is defined to be $j \in [n]$ such that $\pi_j = i$. The output is required to be *eager evaluating*, that is, the output value $\mathbf{y}_i = \mathbf{x}_{\pi_i^{-1}}$ shall be ready as soon as its corresponding input value $(\mathbf{x}_j, \pi_j = i)$ is ready.

Notice that dynamic circuits require gates, especially **Route**, to be eager evaluating: whenever the input values are known, we evaluate the gate immediately to get the output value. As a result, the eager evaluating property makes the ordering of evaluation input-dependent, which is unlike standard circuits, where the ordering is determined by circuit topology. To ensure the computation is well-defined, dynamic circuits require that each wire is assigned a value at most once for any input. Also, the permutation for each Route gate is determined by the inputs to the dynamic circuit.

The Route gate plays an essential role in a dynamic circuit simulating a tree-based ORAM. Specifically, **Route** is the only gate that allows the dynamic circuit to talk to one child without talking to the other. In the dynamic circuit, we use a Route for $n = Z$ cables and connect the first $Z/2$ outputs of Route to the left child and the remaining $Z/2$ outputs to the right child. With an appropriate permutation π_i , the dynamic circuit routes to the on-path child, while the other child circuit is

³There are other dynamic gates defined in previous works [HKO23, Hea24, GGS26, LLLH26], and Route implies some other gates, e.g., the “switch” gate [Hea24]. This work uses only Route. It is open whether our technique improves the efficiency of other gates.

saved for future accesses.⁴

Thus, it is crucial to garble *Route* efficiently. Due to the eager evaluating requirement, a garbled *Route* necessarily reveals the permutation π_i for all cables. Therefore, security requires that the garbled *Route* gate be simulatable given the permutation but no information about the data being routed. When *Route* is employed in the above manner, the permutation is fully determined by the tree node accesses assigned in the tree-based ORAM, which are independent of the data.

2.3 Succinct Garbled Circuits and HGate

Dynamic circuits also use many standard boolean gates in many places, such as to compute the targets as inputs to *Route*. Thus, boolean gates must be efficiently garbled to achieve efficient garbled RAMs. We will follow the succinct garbled circuits of Ishai, Li, and Lin [ILL25b], which we recall at a high level.

Let $\mathbf{x}$ be a vector of bits. Consider a (static) public gate g that computes on $\mathbf{x}$ and outputs another bit-vector $g(\mathbf{x}) = \mathbf{y}$. At a high level, the succinct garbling scheme represents each wire using additive secret shares. These shares are authenticated using a secret s , which the garbler randomly samples s from the underlying group or ring, depending on the cryptographic assumption. Additionally, the garbler samples a vector of group or ring elements $k_{\mathbf{x}}$, which serve as its share for each wire value. The evaluator's share is computed by multiplying the value $\mathbf{x}$ by the secret s for authenticity and adding the result coordinate-wise with the garbler's share. Thus, each party holds an additive share of $s \cdot \mathbf{x}$, denoted by $\langle s \cdot \mathbf{x} \rangle_G := k_{\mathbf{x}}$ for the garbler's share and $\langle s \cdot \mathbf{x} \rangle_E := k_{\mathbf{x}} + s \cdot \mathbf{x}$ as the evaluator's share.

Notice that the garbler's shares are always independent of the value $\mathbf{x}$ and s . Also, the shares $\langle s \cdot \mathbf{x} \rangle_{G/E}$ are the keys and tags of a Message Authentication Code (MAC), which authenticates the value $\mathbf{x}$. Thus, before garbling and evaluating the gate g , each party holds:

$$\text{Garbler: } (\langle s \cdot \mathbf{x} \rangle_G), \quad \text{Evaluator: } (\langle s \cdot \mathbf{x} \rangle_E, \mathbf{x}).$$

For ease of presentation, we begin with the privacy-free setting in which the evaluator holds $\mathbf{x}$ in the clear, and we will introduce privacy later. In this succinct garbling scheme, the goal is to obtain the corresponding shares of $\mathbf{y} = g(\mathbf{x})$ for each gate g . The main idea of this procedure is that the garbler can prepare some public data $\mathbf{pd}$ in advance such that $\mathbf{pd}$ securely encrypts the garbler's secrets s . This public data allow the evaluator to perform homomorphic operations which transform its input shares into the corresponding output shares while the garbler similarly computes their output shares. More specifically, the scheme provides a pair of procedures, HGate_G and HGate_E , to homomorphically and locally perform the g gate computation:

$$\begin{aligned} \text{Garbler:} \quad & (\langle s \cdot \mathbf{y} \rangle_G) \leftarrow \text{HGate}_G(\mathbf{pd}, g, \langle s \cdot \mathbf{x} \rangle_G), \\ \text{Evaluator:} \quad & (\langle s \cdot \mathbf{y} \rangle_E, \mathbf{y}) \leftarrow \text{HGate}_E(\mathbf{pd}, g, \langle s \cdot \mathbf{x} \rangle_E, \mathbf{x}). \end{aligned} \tag{1}$$

We abuse the notation g as the identity of (or the output wires of) the gate g . We remark that the gate g must be known to both parties in advance (otherwise, the scheme must incur additional communication to describe the gate). Also, g can be any gate that is computable by a (standard)

⁴As mentioned in Section 1.1, one may use a functionally weaker gate instead of *Route*, called *Merge* [LLH26], where the permutation is weakened to only left or right for each item.

boolean circuit of size $\text{poly}(\lambda)$ and depth $O(\log \lambda)$. This means that we can use HGate for more than just boolean gates, which will be useful for our Route later.

As mentioned, the above explanation is for the privacy-free setting. We present this as a warm-up that we will use to explain our contributions in Section 3.1. However, both our result and Ishai, Li, and Lin [ILL25b] need privacy. Below, we detail the modification to achieve privacy of inputs in the succinct garbling scheme.

To achieve privacy, we need to hide the wire values from the evaluator. Since HGate still requires the authenticated message from the shares as a plaintext input for correctness, the scheme additionally masks each wire value $\mathbf{x}$ using a one-time pad. To do so, the garbler samples a pseudorandom function (PRF) secret key $\mathbf{sk}$ to generate the masks. The garbler then generates bit-masks as $\mathbf{r} = \text{PRF}_{\mathbf{sk}}(\text{id})$, where id are identities of the wires whose values are $\mathbf{x}$. We denote the masked bit-vectors as $\bar{\mathbf{x}} = \mathbf{x} \oplus \mathbf{r}$ for a bit-mask $\mathbf{r}$. Thus, in the full garbling scheme, the parties hold shares for $s \cdot \bar{\mathbf{x}}$ where the garbler's share remains as $\langle s \cdot \bar{\mathbf{x}} \rangle_G := k_{\mathbf{x}}$ and the evaluator's share is $\langle s \cdot \bar{\mathbf{x}} \rangle_E := k_{\mathbf{x}} + s \cdot \bar{\mathbf{x}}$ over the addition in the corresponding group or ring. In full, they hold

$$\text{Garbler: } (\langle s \cdot \bar{\mathbf{x}} \rangle_G, \mathbf{r}), \quad \text{Evaluator: } (\langle s \cdot \bar{\mathbf{x}} \rangle_E, \bar{\mathbf{x}}).$$

Another difference is that the public data must now encrypt the secret key $\mathbf{sk}$ as well. This is so that the evaluator can evaluate the PRF without learning $\mathbf{sk}$. Since the inputs require authenticated shares, the evaluator can only correctly evaluate the PRF for the masks using the authenticated inputs and the evaluation of g . We highlight the differences from the privacy-free setting in blue:

$$\begin{aligned} \text{Garbler:} \quad & (\langle s \cdot \bar{\mathbf{y}} \rangle_G, \mathbf{c}_g) \leftarrow \text{HGate}_G(\text{pd}, \mathbf{sk}, g, \langle s \cdot \bar{\mathbf{x}} \rangle_G), \\ \text{Evaluator:} \quad & (\langle s \cdot \bar{\mathbf{y}} \rangle_E, \bar{\mathbf{y}}) \leftarrow \text{HGate}_E(\text{pd}, g, \langle s \cdot \bar{\mathbf{x}} \rangle_E, \bar{\mathbf{x}}, \mathbf{c}_g). \end{aligned} \tag{2}$$

In the setting where we require privacy, the evaluator is unable to evaluate g in the clear because the inputs are masked. This requires a message from the garbler to fix, which is a share of the value $\bar{\mathbf{y}}$, denoted as $\mathbf{c}_g$. This cipher is the only message sent from the garbler to the evaluator to compute their full share. The size of this cipher is the same size as $\mathbf{y}$ in bits, meaning that the communication cost of boolean gates is 1-bit per gate. Notice that the garbler can generate the masks of the bit-vectors for $\mathbf{x}$ and $\mathbf{y}$ using $\mathbf{sk}$ whenever needed. More details on the full construction of HGate can be found in Section 4.3.

3 Overview of Our Garbled RAM

This section provides an overview of our garbled RAM. The main technical improvement is our garbled Route that takes less communication. In Section 3.1, we show our conceptual ideas toward the improvement, where we use homomorphic secret sharing techniques, such as fixed factor hashing [LLL26]. In Section 3.2, we present the procedure of the garbled Route. Finally in Section 3.3, our garbled RAM is composed from Route, Circuit ORAM, and other building blocks using a proper parameterization.

We briefly recall that the Route gate is parameterized by n and ℓ . It takes n input cables $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$, each cable consists of ℓ bits, and n input control wires $(\pi_1, \pi_2, \dots, \pi_n)$ that specify a permutation in $[n]$. The output is the permuted cables $(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n)$ such that $\mathbf{y}_{\pi_i} = \mathbf{x}_i$. In addition, Route is required to have the eager evaluating property: the output $\mathbf{y}_{\pi_i}$ is obtained as

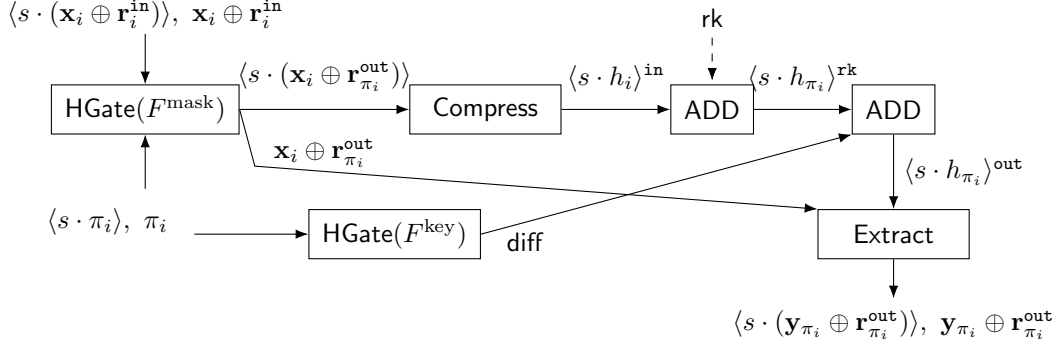


Figure 1: Route from input cable $\mathbf{x}_i$ to output cable $\mathbf{y}_{\pi_i}$, in the evaluator’s perspective. Each box is an algorithm, where ADD is addition over the group/ring. Each arrow is data, where the dashed arrow means rk is provided by the garbler. HGate implicitly uses garbler’s messages, and most of garbler’s procedures are symmetric except for ADD.

soon as the input $\mathbf{x}_i$ and control wire π_i are both ready, without waiting for any other input to arrive. Also recall that Route is the key component to garble a tree-based ORAM efficiently, so that we want to achieve $O(\lambda + \ell)$ -bit communication per cable.

3.1 Concepts for Garbled Route from Homomorphic Secret Sharing

At a glance, our Route is composed from homomorphic tools in order to manipulate shares between the garbler and evaluator. In this section, we provide the ideas necessary to see how all these tools connect to construct our Route, which is our main technical contribution. We provide a visual overview in Fig. 1 to be used as reference. Here, we show the evaluator’s view of the procedure, omitting some details and communication from the garbler. For a detailed outline of the procedure, see Section 3.2.

Recall that n is the number of cables being routed, ℓ is the number of bits per cable, and each cable is associated with a target $\pi_i \in [n]$ describing where it should be routed. For ease of presentation, we first describe a privacy-free construction in which the evaluator is given each value $\mathbf{x}_i$ directly rather than a masked value $\bar{\mathbf{x}}_i$; this lets us focus on the routing mechanism itself before reintroducing masking to achieve privacy. In the garbled Route, the evaluator holds shares for a data value $\mathbf{x}$ fully as $(\langle s \cdot \mathbf{x}_i \rangle_E, \mathbf{x}_i, \langle s \cdot \pi_i \rangle_E, \pi_i)$ for some $i \in [n]$, and the evaluator’s goal is to compute the corresponding output cable’s shares $(\langle s \cdot \mathbf{y}_{\pi_i} \rangle_E, \mathbf{y}_{\pi_i})$. However, since π_i is itself computed from private input within the program, the garbler does not, and cannot, know in advance what π_i is, so the HGate described in Section 2.3 does not help straightforwardly. Moreover, the requirement for eager evaluating is incompatible with the construction of HGate: in the scheme [ILL25b, Section 5], the evaluator’s shares of the output require *all* input shares to ensure authenticity using a homomorphic MAC. Using the MAC, we do not know how to authenticate the output when eagerly evaluating.

Update the evaluator’s shares dynamically. Our first step is to decouple the garbler’s output shares from any particular input, so that the garbler need not learn π_i . Specifically, suppose that the garbler’s shares, $\langle s \cdot \mathbf{x}_i \rangle_G$, and $\langle s \cdot \mathbf{y}_i \rangle_G$, are all generated using PRF and the secret key sk for

all input and output cables, $\mathbf{x}_i, \mathbf{y}_i$. Note that this $\mathbf{sk}$ is different from the $\mathbf{sk}$ in Section 2.3. That is,

$$\langle s \cdot \mathbf{x}_i \rangle_G \leftarrow \text{PRF}_{\mathbf{sk}}(\text{id}_{\text{Route}}, i, \text{in}, \text{tag}), \quad \langle s \cdot \mathbf{y}_i \rangle_G \leftarrow \text{PRF}_{\mathbf{sk}}(\text{id}_{\text{Route}}, i, \text{out}, \text{tag}), \quad (3)$$

where id_{Route} and i are identifiers of the Route and the cable, respectively. Similarly, in, out , and tag , are distinct identifiers to get pseudorandom shares. It “suffices” to deliver the differences

$$\text{diff}_{\text{tag}} = \langle s \cdot \mathbf{y}_{\pi_i} \rangle_G - \langle s \cdot \mathbf{x}_i \rangle_G$$

to the evaluator, so that the evaluator can compute $\langle s \cdot \mathbf{y}_{\pi_i} \rangle_E = \text{diff}_{\text{tag}} + \langle s \cdot \mathbf{x}_i \rangle_E$. This would be a strawman candidate.

We assume that the PRF is in NC1, which is already needed in the work of Ishai, et al. [ILL25b]. Observe that diff_{tag} can be computed given $\mathbf{sk}, i$, and π_i using PRF. Thus, we can build an NC1 circuit $F_i^{\text{key}}(\mathbf{sk}, \pi_i)$ that computes the desired differences. Then, employing HGate with the circuit F_i^{key} and the corresponding shares of π_i , we can obtain diff_{tag} . We omit a few details. For example, F_i^{key} is not exactly the gate g given to HGate, as HGate already implicitly uses $\mathbf{sk}$ from pd . Still, when the details are filled properly, such a strawman is indeed correct and secure.

Unfortunately, the strawman candidate is inefficient even in this privacy-free setting. Recall that our target communication is $O(\lambda + \ell)$ per cable in the privacy setting; the $O(\lambda)$ term is what we aim to achieve for eager-evaluating Route, independent of privacy, while the $O(\ell)$ term will come from masking for privacy. However, the strawman already requires $O(\ell \cdot \lambda)$ bits per cable communication. This is because the evaluator needs the cipher $\mathbf{c}_g$ sent from the garbler, and here g indicates the circuit F_i^{key} . However, $\mathbf{c}_g$ has the same bit-length as the desired outputs, which are diff_{tag} . Since $s \cdot \mathbf{y}_i$ consists of ℓ group or ring elements, where each group or ring element is $O(\lambda)$ bits, the diff_{tag} and thus the communication take $O(\ell \cdot \lambda)$ bits per cable. Jumping to implementing a tree-based ORAM using Route, that means $O(\ell \cdot \lambda) \cdot \log N$ bits of communication for each tree path of $\log N$ nodes, where $\ell = \Theta(W + \log N)$ is the cable size when we move W -bit words. So the multiplicative factor ℓ is undesirable.

Still, it is an important idea to garble a dynamic circuit, and we will improve it next. Dynamically evaluating the garbled wire labels is conceptually similar to the techniques used in all previous garbled RAMs (e.g., [GLOS15, HKO22]), but, to the best of our knowledge, this is the first time the idea works with succinct garbled circuits.

Compress the communication by fixed factor hashing. Our solution is inspired by the recent low-depth garbled circuit of Li, Lin, and Lu [LLL26]. We recall their idea at a high level. They followed the succinct garbled circuit [ILL25b] but improved the circuit depth of the garbler. To keep the garbler’s depth low and independent of the depth of the circuit being garbled, in their scheme, the garbler’s shares are sampled using a PRF, exactly the same as Equation (3). However, the evaluator still uses HGate_E and obtains the shares as in Equation (2). That is a mismatch between the garbler’s and evaluator’s shares. To resolve this, the garbler sends the *differences* between the two shares, so that the evaluator gets matching shares to continue with the next gate. There is a similar issue as in our construction for dynamic circuit: sending the differences takes λ -times communication, too much.

The resolution of Li, Lin, and Lu [LLL26] is conceptually simple but effective. When we want to send the differences between two vectors of shares all at once, we can instead compress the shares and then send the difference between the *compressed shares*. To elaborate, let $\mathbf{x}$ be a bit-vector, sample a random linear compressing function Hash , and compute the hash $h = \text{Hash}(\mathbf{x})$. They provide a pair of algorithms Compress and Extract with public data Hash.pd , where Hash.pd is prepared depending on the Hash function and secret s . Given the shares of $\mathbf{x}$, two parties can locally obtain a share of h :

$$\langle s \cdot h \rangle_{G/E} \leftarrow \text{Compress}(\text{Hash.pd}, \langle s \cdot \mathbf{x} \rangle_{G/E}),$$

where the subscript G/E denotes the garbler's or the evaluator's shares correspondingly. Notice that the hash value h is determined by the input $\mathbf{x}$. Here, neither party knows h , yet the shares $\langle s \cdot h \rangle_{G/E}$ authenticate h . The compressed shares can be expanded to re-authenticate the original value $\mathbf{x}$ with Extract , computed locally by each party:

$$\langle s \cdot \mathbf{x} \rangle'_G \leftarrow \text{Extract}_G(\text{Hash.pd}, \langle s \cdot h \rangle_G) \quad \text{and} \quad \langle s \cdot \mathbf{x} \rangle'_E \leftarrow \text{Extract}_E(\text{Hash.pd}, \langle s \cdot h \rangle_E, \mathbf{x}),$$

where $\langle s \cdot \mathbf{x} \rangle'_G$ is another share that depends on $\langle s \cdot h \rangle_G$, but is not exactly the original share $\langle s \cdot \mathbf{x} \rangle_G$. Regardless, the new share $\langle s \cdot \mathbf{x} \rangle'_E$ of the evaluator remains a valid authenticated share of the value $\mathbf{x}$. The value $\mathbf{x}$ is unchanged. Crucially, the Extract works correctly for *any* shares $\langle s \cdot h \rangle'_{G/E}$ of h that are potentially rerandomized, so that we can apply the same differencing trick at the level of the compressed hash. This technique is called *fixed factor hashing* by Li, Lin, and Lu.

To garble Route with the compression, we Compress the shares of the cable $\langle s \cdot \mathbf{x}_i \rangle_{G/E}$ into the share of the hash $\langle s \cdot h_i \rangle_{G/E}^{\text{in}}$, dynamically compute and reveal the differences between the shares on input cable i and the share of output cable π_i . With that, the evaluator obtains the desired share $\langle s \cdot h_j \rangle_E^{\text{out}}$ for the output cable $j = \pi_i$ such that the share matches the garbler's share $\langle s \cdot h_j \rangle_G^{\text{out}}$. Using $\mathbf{x}_i$ and $\langle s \cdot h_j \rangle_{G/E}^{\text{out}}$, both parties then Extract and obtain the shared values and tags on the output cable j . The communication is dominated by the differences between the shares of hashes, which equals the length len of the share, $\text{len} = |\langle s \cdot h_j \rangle_E^{\text{out}}|$. Directly applying Compress and Extract from [LLL26] in this way is feasible, but modifications are needed to reach our target communication.

The previous work [LLL26] removes the multiplicative factor ℓ but leaves the share length being $\text{len} = O(\lambda^2)$ bits because their context allowed more communication. Such a $\text{len} = O(\lambda^2)$ is too large for our target communication, so we depart from [LLL26] to bring len down to $O(\lambda)$ bits. Specifically, their construction leaves the compressed share $\langle s \cdot h \rangle$ as a $O(\lambda)$ -length vector of group or ring elements, whereas our hash h is a single scalar in group setting or constant length vector in lattice setting. This optimization lets us tighten len from $O(\lambda^2)$ to $O(\lambda)$ bits, and it is explained next.

Optimization of Compress and Extract . In the group setting, our idea is to simply move the bit-composition step from Extract into Compress . In Li-Lin-Lu, Extract receives the shares $\langle s \cdot \mathbf{h} \rangle$ of the bit-decomposition $\mathbf{h}$ of the hash value h , and it runs bit-composition to recover the share $\langle s \cdot h \rangle$ of the same hash value. Since bit composition is simply a linear combination and uses no key material beyond what Compress already has access to, we can instead perform this composition *before* communication: our Compress sums the bit-level shares directly, so the compressed share $\langle s \cdot h \rangle$ is already a single scalar, shortening it by a factor of λ .

Porting this idea to the lattice setting is not as immediate, since there is no clean analog of bit composition of $\langle s \cdot h \rangle$ there. Here, with RLWE assumption, “bit” is a loose term: what is bounded is the magnitude of the coefficients of the ring element, not literal bits. Namely, their **Compress** yielded the shares $\langle s \cdot \mathbf{h} \rangle$ of a vector $\mathbf{h}$. While $\mathbf{h}$ can be viewed as a decomposition of some scalar h , but their **Extract** used $(\langle s \cdot \mathbf{h} \rangle, \mathbf{x})$ directly to get the resulting shares, without went through any share $\langle s \cdot h \rangle$ for any h .

What we can still optimize is the length of this vector $\mathbf{h}$. To do that, we fine-tune the bound B on the coefficients of the ring elements. Let q be the modulus of the ring, and suppose $\log q = O(\lambda)$ for exposition. In Li-Lin-Lu, the bound B was taken to be 2, i.e., binary coefficients, and that requires $\mathbf{h}$ to be $O(\log q)$ -dimension, which incurs the $O(\lambda)$ factor. Had one taken an extremely large B close to q , one would get $\mathbf{h}$ of $O(1)$ dimensions, but the resulting error in homomorphic secret shares would be too large for **Extract** to recover correct shares. Our resolution is to choose an appropriate coefficient bound B so that $\log B = O(\log q)$ and $B < q^\epsilon$ for some constant $\epsilon > 0$. Such B is large enough to keep the dimension of $\mathbf{h}$ constant, yet small enough relative to q to preserve the correctness of **Extract**. More details about **Compress** and **Extract** can be found in Section 8.

Privacy via dynamic remasking. So far, we have discussed mostly the privacy-free setting, but in our full construction we will want to hide input values $\mathbf{x}$ via masking. As recalled in Section 2.3, HGate achieves privacy by masking each wire value $\mathbf{x}$ with a mask $\mathbf{r}$ generated by a PRF. In the context of Route, consider $\mathbf{x}_i$ as the wire values for a cable i . The garbler samples $\mathbf{r}_i^{\text{in}} \leftarrow \text{PRF}_{\text{sk}'}(\text{id}_{\text{wire}})$ for each input wire, and the evaluator only ever sees the masked value $\hat{\mathbf{x}}_i = \mathbf{x}_i \oplus \mathbf{r}_i^{\text{in}}$, together with shares of $\langle s \cdot \hat{\mathbf{x}}_i \rangle_G$. Route must follow this same approach, and the secret key of the PRF here is different from the one used to generate the garbler’s share because the outputs are different: sk outputs λ -sized outputs for shares and sk' outputs ℓ -bit outputs for masks. Unlike a static gate, however, the output wire’s identity for Route depends on π_i , which the garbler does not know in advance. This means that the garbler does not know how to apply the output mask $\mathbf{r}_{\pi_i}^{\text{out}}$ when garbling the dynamic circuit. We resolve this the same way we resolved the analogous issue for tags: let the HGate take as inputs the share $\langle s \cdot \pi_i \rangle$ and output the remasked value $\bar{\mathbf{x}}_i = \hat{\mathbf{x}}_i \oplus \mathbf{r}_{\pi_i}^{\text{out}} \oplus \mathbf{r}_i^{\text{in}}$. This step effectively removes the input mask and applies the proper output mask dynamically, and it contributes an additive $O(\ell)$ term to the per-cable communication. Finally, we put everything together and sketch the Route construction with more details below to include privacy of inputs.

3.2 Procedure of Garbled Route

To begin with, we suppose that both the garbler and evaluator hold the input shares of a masked $\hat{\mathbf{x}}_i = \mathbf{x}_i \oplus \mathbf{r}_i^{\text{in}}$ as $(\langle s \cdot \hat{\mathbf{x}}_i \rangle_{G/E} = \{\langle s \cdot \hat{x}_{i,t} \rangle_{G/E}\}_{t \in [\ell]}, \langle s \cdot \pi_i \rangle_{G/E})$, where the evaluator additionally holds $(\hat{\mathbf{x}}_i = \{\hat{x}_{i,t}\}_{t \in [\ell]}, \pi_i)$. We separately denote masked values $\hat{\mathbf{x}}$ and $\bar{\mathbf{x}}$ to represent whether the value x is encrypted by the mask value associated with an input wire or output wire, respectively. For the garbler, the shares are for all $i \in [n]$, but for the evaluator, there is only one known and fixed $i \in [n]$. The masks, $\mathbf{r}^{\text{in}}$, as well as the output value shares $\mathbf{r}^{\text{out}}$, are generated by $\text{PRF}_{\text{sk}'}$ as per the succinct garbled circuit [ILL25b]. Other shares in the input are derived by HGate the gates prior to the Route. In the following, we underline whenever the garbler sends messages, and all other operations are local.

1. Both parties performs HGate so that the evaluator obtains the remasked $\hat{\mathbf{x}}_i = \{\hat{x}_{i,t}\}_{t \in [\ell]}$ and

$\langle s \cdot \hat{\mathbf{x}}_i \rangle_E = \{\langle s \cdot \hat{x}_{i,t} \rangle_E\}_{t \in [\ell]}$. To do so, each party performs

Garb: $(\langle s \cdot \bar{x}_{i,t} \rangle_G, c_{i,t}^{\text{val}}) \leftarrow \text{HGate}_G(\text{pd}_{\text{sk}'}, F_{g,i,t}^{\text{mask}}, \{\langle s \cdot \pi_i \rangle_G, \langle s \cdot \hat{x}_{i,t} \rangle_G\})$ for all i, t ,

Eval: $(\langle s \cdot \bar{x}_{i,t} \rangle_E, \bar{x}_{i,t}) \leftarrow \text{HGate}_E(\text{pd}_{\text{sk}'}, F_{g,i,t}^{\text{mask}}, \{\pi_i, \hat{x}_{i,t}\}, \{\langle s \cdot \pi_i \rangle_E, \langle s \cdot \hat{x}_{i,t} \rangle_E\}, c_{i,t}^{\text{val}})$ for all t ,

where gate $F_{g,i,t}^{\text{mask}}(\text{sk}')$ computes $\hat{x}_{i,t} \oplus r_{\pi_i,t}^{\text{out}} \oplus r_{i,t}^{\text{in}}$, and the cipher $\mathbf{c}_i^{\text{val}} = \{c_{i,t}^{\text{val}}\}_{t \in [\ell]}$ for all i are sent to the evaluator.

- Both parties compress the shares, the garbler for all i , but the evaluator for the fixed i ,

$$\langle s \cdot h_i \rangle_{G/E}^{\text{in}} \leftarrow \text{Compress}(\text{Hash.pd}, \langle s \cdot \bar{\mathbf{x}}_i \rangle_{G/E}).$$

Notice that the garbler's share $\langle s \cdot h_i \rangle_G^{\text{in}}$ is derived by many steps and is hard to re-derive. Hence, the next step rekeys shares.

- Rekey the shared compression as follows. The garbler samples $\langle s \cdot h_i \rangle_G^{\text{rk}} \leftarrow \text{PRF}_{\text{sk}}(\text{ID}(g, \text{rk}, i))$, where rk is yet another constant for pseudorandomness, and then sends $\text{rk}_i = \langle s \cdot h_i \rangle_G^{\text{rk}} - \langle s \cdot h_i \rangle_G^{\text{in}}$ for all i to the evaluator. The evaluator computes $\langle s \cdot h_i \rangle_E^{\text{rk}} = \text{rk}_i + \langle s \cdot h_i \rangle_E^{\text{in}}$. The “rekey” terminology follows [LLL26].
- Both parties performs HGate so that the evaluator obtains diff_{tag} and thus $\langle s \cdot h_j \rangle_E^{\text{out}} = \langle s \cdot h_i \rangle_E^{\text{rk}} + \text{diff}_{\text{tag}}$ for $j = \pi_i$. That is, similar to Step 1, each party performs

$$\begin{aligned} \text{Garbler:} \quad & (\dots, \mathbf{c}_i^{\text{tag}}) \leftarrow \text{HGate}_G(\text{pd}_{\text{sk}}, F_{g,i}^{\text{key}}, \langle s \cdot \pi_i \rangle_G) \text{ for all } i, \\ & \langle s \cdot h_i \rangle_G^{\text{out}} = \text{PRF}_{\text{sk}}(\text{ID}(g, \text{out}, i)) \text{ for all } i, \\ \text{Evaluator:} \quad & (\dots, \text{diff}_{\text{tag}}) \leftarrow \text{HGate}_E(\text{pd}_{\text{sk}}, F_{g,i}^{\text{key}}, \langle s \cdot \pi_i \rangle_E, \pi_i, \mathbf{c}_i^{\text{tag}}), \\ & \langle s \cdot h_j \rangle_E^{\text{out}} = \langle s \cdot h_i \rangle_E^{\text{rk}} + \text{diff}_{\text{tag}}, \end{aligned}$$

where the main difference is that the gate $F_{g,i}^{\text{key}}$ here computes the subtraction

$$\text{diff}_{\text{tag}} = \text{PRF}_{\text{sk}}(\text{ID}(g, \text{out}, \pi_i)) - \text{PRF}_{\text{sk}}(\text{ID}(g, \text{in}, i)).$$

The cipher $\mathbf{c}_i^{\text{tag}}$ for all i are sent to the evaluator similarly.

- Both parties extract and obtain the shares of the output cable $j = \pi_i$ locally by

$$\begin{aligned} \langle s \cdot \bar{\mathbf{y}}_j \rangle_G & \leftarrow \text{Extract}_G(\text{Hash.pd}, \langle s \cdot h_j \rangle_G^{\text{out}}, t) \text{ for all } j, t, \text{ and} \\ \langle s \cdot \bar{\mathbf{y}}_j \rangle_E & \leftarrow \text{Extract}_E(\text{Hash.pd}, \langle s \cdot h_j \rangle_E^{\text{out}}, \bar{\mathbf{x}}_i, t) \text{ for all } t. \end{aligned}$$

The above procedure sketched our garbled Route. The communication is taken by messages $\mathbf{c}_i^{\text{val}}$, rk_i , and $\mathbf{c}_i^{\text{tag}}$, which takes ℓ , $O(\lambda)$ and $O(\lambda)$ bits correspondingly for each cable i , where we suppose that the size of a group or ring element is parameterized to $O(\lambda)$. Thus, the per cable communication is $O(\lambda + \ell)$ bits. The running time for both garbler and evaluator is dominated by HGate operations, which takes $n \cdot \text{poly}(\lambda)$ for each bit in $\mathbf{c}^{\text{val/tag}}$, and thus it is $n \cdot \ell \cdot \text{poly}(\lambda)$ per cable. Other operations, such as Compress and Extract , take time $O(\ell^2 \cdot \text{poly}(\lambda))$ per cable [LLL26]. The correctness follows similar to the prior works [ILL25b, LLL26].

Security. As the construction relies on HGate, we inherited the assumptions: the existence of a PRF in the class NC1 and any one of (1) CP-DDH in Paillier groups, or (2) CP-DDH in prime-order groups, or (3) CP-RLWE. Due to the use of **Compress** and **Extract**, we also inherit their assumptions [LLL26]—since they consider boolean circuits of bounded depth, they applied the standard versions of RLWE or DDH in a leveled fashion. However, as **Route** will be used in garbled RAM, we cannot assume bounded depth, and we need to additionally assume the circular versions of DDH or RLWE. That is, in the public data **Hash.pd**, we use the secret s to encrypt messages depending on s (e.g., bits of s) and we assume such a circular encryption hides s . The circular encryption differs from CP-DDH or CP-RLWE as per [ILL25b] because the CP-DDH or CP-RLWE assumptions use s^2 to encrypt messages depending on s . We need both s -circular (for **Extract**) and s^2 -circular (for HGate). With the assumptions, proving security is similar to prior works. It is because the messages are just pseudorandom shares and are secure when the public data (**pd**, **Hash.pd**) are proven to be securely hiding s and sk , as in the prior works.

We do not know whether our assumption is implied by the previous CP-DDH or CP-RLWE. We conjecture that some modification of our construction might remove one of the two circularities, but we did not find a clear way. Also, although circular RLWE implies strong primitives, such as FHE or even RAM-FHE, it remains unclear how to obtain a succinct garbled RAM from the primitives.

Putting garbled Route in an entire dynamic circuit. The above procedure of garbling **Route** seemed to require that all input shares be prepared prior to garbling the **Route** gate. If that was the case, it would be a huge issue since a dynamic circuit is allowed to form a cyclic graph—in GRAM constructions, being cyclic is essential [HKO23].

Fortunately, we observe that in the garbling procedure of **Route**, the output shares are actually *independent* of the input shares! Indeed, the output shares depend only on the public data and $\langle s \cdot h_i \rangle_G^{\text{out}}$, which is generated using PRF in Step 4. As long as the shares are generated first, all remaining gates can be garbled accordingly; for details, see Algorithm 3. The evaluation procedure remains the same. That is also how the Li-Lin-Lu garbling achieves a low depth.

3.3 Garbled RAMs from Garbled Route

To construct garbled RAM using our garbled **Route**, we instantiate the framework in Section 2.2. That is, a tree-based ORAM is compiled into a dynamic circuit, whose standard boolean gates are garbled via HGate and whose **Route** gates are garbled via the scheme of Section 3.1. Still, we need to fill in some parameters. Following Zebra [GGS26], we will use Circuit ORAM, a tree-based ORAM scheme [WCS15], but the recursion is modified. Figure 2 illustrates the resulting construction at a high level. Next, by describing the construction and efficiency of the main ORAM tree, we show the role of an efficient **Route**, i.e., how does **Route** enable proper parameters in the framework.

Main ORAM tree from Route. We follow the structure of Circuit ORAM. Let $Z = \text{poly log } N$ to ensure measure concentration. We instantiate a complete binary tree of N/Z leaves. Each leaf node stores at most Z words, and the root node also stores at most Z words. Each non-root internal node stores a small constant number of words, e.g., 4 or 5 words. Besides storing words, each node also supports some functionalities required during Circuit ORAM’s fetch and eviction procedures.

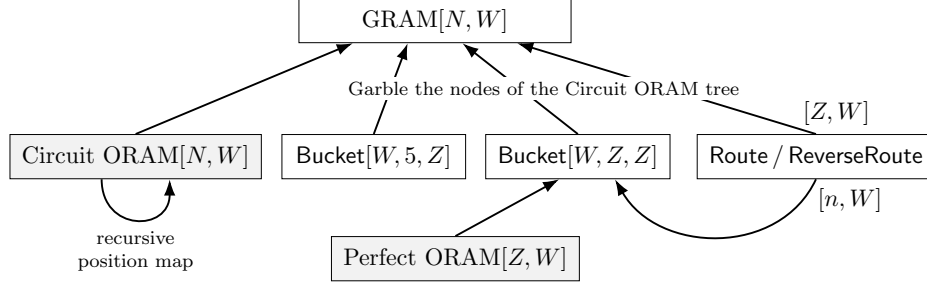


Figure 2: An illustration of the construction of $\text{GRAM}[N, W]$, with memory size N and word size W . Gray boxes denote plain and ungarbled primitives; white boxes denote the garbled gadgets. Circuit ORAM is a tree-like data structure, and we use **Bucket** and **Route/ReverseRoute** to garble each of its nodes. $\text{Bucket}[a, b, c]$ denotes its word size a , its capacity b , and the number c of operations it supports. $(\text{Reverse})\text{Route}[a, b]$ denotes the number a of cables and the number b of wires per cable. We set $Z = \text{poly } \log N$ and $n \leq O(Z)$ to be powers of 2.

We defer the controlling of the fetching and eviction, but we focus on the connection between nodes next.

To garble the above tree of Circuit ORAM, each node is associated with a sequence of dynamic circuits gadgets, as mentioned in Section 2.2. Each of the gadgets, regardless of its associated node being root, internal, or leaf, takes at most $\Theta(Z)$ accesses, with an expectation of Z accesses. For each access on a node, its parent node passes down $O(W)$ bits for a word and $O(\log N)$ bits for the position. No matter the controlling, the node gadget reads its storage and then passes down a proper word and proper position to one of its two children. After the information reaches the leaf, the same path passes and computes information reversely from leaf to the root. As mentioned, for a RAM program that performs T memory accesses, we prepare a sequence of $O(T/Z)$ dynamic circuit gadgets for the root node and similarly $O(T/(2^d \cdot Z))$ dynamic circuit gadgets for each depth d nodes.

Notice that in the node gadgets, the controlling decides which direction to pass information, i.e., which of the left or right child, and which of up and down the tree; we defer its dynamic circuit instantiation. Once the direction is decided, the node gadget invokes **Route** to send the information to the next node gadget. Since each node gadget takes $\Theta(Z)$ accesses, the garbled **Route** permutes $\Theta(Z)$ cables, each is $O(W + \log N)$ bits. Applying our garbled **Route**, the communication is $O(W + \log N + \lambda)$ per node gadget, and it sums to $O((W + \log N + \lambda) \cdot \log N)$ per tree path. We next instantiate the controlling.

Garbling tree-node storage with **Bucket.** In the node gadget, the controlling can be instantiated mostly using standard boolean gates due to the simplicity of Circuit ORAM. Recall that a node stores either a constant number or $O(Z)$ number of words. Let **Bucket** be the data structure storing the words. The controlling is to search, store, or replace a word in **Bucket**. When the node stores at most a constant number of words, the **Bucket** and thus the node gadget is entirely instantiated using standard boolean gates, which can be efficiently garbled with **HGate**. However, for the root and leaf nodes, the node stores $O(Z)$ words, and we do not want to pay the overhead of Z per access. To efficiently search, store, or replace words in such node gadgets, we construct **Bucket**

with capacity Z using perfect ORAM [CNS18] and **Route**. Similar to the role of **Route** in Circuit-ORAM-based dynamic circuit, our **Route** provides dynamic activation when the perfect ORAM is instantiated as a dynamic circuit. We refer to Section 6 for the detailed **Bucket** construction. The per-operation cost of such **Bucket** is $(W + \log N + \lambda) \cdot \text{poly}(\log Z)$ in communication. Because there are only two root and leaf nodes per path, the $\text{poly} \log Z$ factor is dominated by $\log N$.

This summarizes the garbling of the main ORAM tree at a high level, assuming the recursive position map is garbled properly. The per-access communication in bits is

$$O((W + \log N + \lambda) \cdot (\log N + \text{poly} \log Z)) = O((W + \log N + \lambda) \cdot \log N). \quad (4)$$

Recursive ORAM trees. It remains to garble the recursive position map of Circuit ORAM, which we describe below. For each tree in the recursive position map, we choose the data block size to be $B = \lambda^\epsilon \cdot \log N$ bits for some constant fraction $\epsilon > 0$ such that $\lambda^\epsilon \geq 2$ and $B \leq \lambda$. Our choice of B ensures that the recursion reduces the number of words by the factor λ^ϵ . Thus by $N = \text{poly}(\lambda)$, there are at most $\log_{\lambda^\epsilon} N = O(1)$ number of recursive trees. For each garbling of a recursive tree, the communication is similar to Equation (4) and is bounded by $O((B + \log N + \lambda) \cdot \log N)$, where we set W to be B . Therefore, summing over all recursion, the total communication per access is

$$O((W + B + \log N + \lambda) \cdot \log N) = O((W + \lambda) \cdot \log N),$$

by $W \geq \log N$ and $B = O(\lambda)$, as claimed in Theorem 1.1.

A note on the size of Route. Recall that our garbled **Route** takes $O(\lambda + \ell)$ bits of communication per cable, where the communication is independent of the number n of cables being routed. It is tempting to exploit such efficient communication by employing large n . Unfortunately, the per-cable *computation* of our garbled **Route** consists of a multiplicative factor n , and thus it is undesirable to instantiate garbled **Route** with large n . We thus choose $n = Z = \text{poly} \log N$ in our scheme. Still, because computation may be less expensive than communication, employing moderately large **Route** may be desirable in other scenarios.

4 Preliminary

We use bold letters $\mathbf{x}$ to denote a vector and plain letter x to denote an element. We use $\mathbf{x} \odot \mathbf{y}$ to denote component-wise multiplication between two vectors. For an integer value within a range specified in the context, we use $\text{Bits}(x)$ to denote its bit-representation as a Boolean vector; we sometimes abuse notation and denote $x[i] = \text{Bits}(x)[i]$. For a vector $\mathbf{x}$, $|\mathbf{x}|$ denotes the dimension of $\mathbf{x}$. For any $n \in \mathbb{N}$, $[n] = \{0, 1, 2, \dots, n-1\}$; for any positive real number r , let $\lceil r \rceil = \lceil r \rceil$. For two ensembles of distributions $\mathcal{X} = \{X_\lambda\}_{\lambda \in \mathbb{N}}$ and $\mathcal{Y} = \{Y_\lambda\}_{\lambda \in \mathbb{N}}$, we write $\mathcal{X} \approx_c \mathcal{Y}$ if they are computationally indistinguishable, and $\mathcal{X} \approx_s \mathcal{Y}$ if they are statistically close. For group element g , $\text{GroupGen}(g)$ denotes the group generated by g . We use $(x \_ y)$ in our algorithms to denote local material to the left (x), and messages being sent/received on the right (y) based on the context.

4.1 Definition of Garbling

Throughout this paper, we consider garbled circuits when the circuit is *public*.

Definition 4.1 (Garbling Scheme). *A garbling scheme contains a tuple of efficient algorithms:*

- $\text{Garb}(1^\lambda, C)$: for security parameter λ and a circuit C with input size $|\mathbf{x}|$ and output size $|\mathbf{y}|$, outputs a garbled circuit $\tilde{C}$ with input key functions $\{K_i\}_{i \in [|\mathbf{x}|]}$. Each key function K_i maps an input x_i to a label L_i with length bounded by $O(\lambda)$.
- $\text{Eval}(C, \tilde{C}, \{L_i\}_{i \in [|\mathbf{x}|]})$: for a circuit C , garbled circuit $\tilde{C}$, and input labels L_i , outputs an evaluation of circuit C .

Correctness: *A garbling scheme is correct if, for polynomial-time circuit C and security parameter λ , there exists a negligible function $\text{negl}(\lambda)$ such that, for any valid input $\mathbf{x}$,*

$$\Pr \left[\text{Eval}(C, \tilde{C}, \{L_i\}_{i \in [|\mathbf{x}|]}) = C(\mathbf{x}) \mid (\tilde{C}, \{K_i\}) \leftarrow \text{Garb}(1^\lambda, C), \forall i \in [|\mathbf{x}|] L_i = K_i(x_i) \right] \geq 1 - \text{negl}(\lambda).$$

Security: *A garbling scheme is secure if there exists an efficient simulator Sim such that, for any sequence of circuits $\{C_\lambda\}_{\lambda \in \mathbb{N}}$ with respective valid inputs $\{\mathbf{x}_\lambda\}_{\lambda \in \mathbb{N}}$,*

$$\left\{ \text{Sim}(1^\lambda, C_\lambda, C_\lambda(\mathbf{x}_\lambda)) \right\}_\lambda \approx_c \left\{ (\tilde{C}, \{L_i\}_{i \in [|\mathbf{x}_\lambda|]}) \mid (\tilde{C}, \{K_i\}) \leftarrow \text{Garb}(1^\lambda, C_\lambda), \forall i \in [|\mathbf{x}|] L_i = K_i(x_i) \right\}_\lambda.$$

4.2 Hardness Assumptions

Here we detail the cryptographic assumptions we use. In particular, we rely on the Circular Power-DDH, or Circular Power-RingLWE assumptions. For Paillier-based assumptions, the homomorphic tools we use require a particular “small exponent” assumption [ADOS22] which lets us sample the secret exponent from a smaller distribution. This is required for correctness in the constructions that we borrow from Ishai, Li, and Lin [ILL25b], so we state the small exponent variant (Definition 4.5).

Definition 4.2 (Paillier Groups). *Paillier groups are defined by their generation algorithm:*

- $\text{Gen}(1^\lambda, 1^\zeta)$: samples two λ -bit primes $p = 2p' + 1$, $q = 2q' + 1$, where p' and q' are also primes. It outputs $(\mathcal{N} = pq, \zeta)$ describing the group $G = \mathbb{Z}_{\mathcal{N}^{\zeta+1}}^*$.

Lemma 4.3 (Facts about Paillier Groups [Pai99, DJ01]). *Let $G = \mathbb{Z}_{\mathcal{N}^{\zeta+1}}^*$ be a Paillier groups sampled by $\text{Gen}(1^\lambda, 1^\zeta)$.*

- G has a subgroup $F = \{(1 + \mathcal{N})^x : x \in \mathcal{N}^\zeta\}$ where finding x can be efficiently solved.
- G has a subgroup H that is isomorphic to $\mathbb{Z}_{\mathcal{N}}^*$, and $G = F \times H$.
- For a random element $g \in G$ such that the Jacobi symbol of $g \bmod \mathcal{N}$ is 1, $\text{GroupGen}\langle g \rangle$ contains F except with negligible probability. Let $g \leftarrow \text{Samp}(\mathcal{N}, \zeta)$ denote the sampling of such a g with Jacobi symbol 1.

Definition 4.4 (Prime-order groups). *Prime-order groups are defined by their generation algorithm:*

- $\text{Gen}(1^\lambda)$: outputs (G, p, g) where G is the group description of prime order $p > 2^\lambda$, and g is a parameter of G .

We adapt the circular power-DDH assumptions under Paillier and prime order groups from Ishai, Li, and Lin [ILL25b, Definition 7], where the only difference is an additional circular ciphertext is given to the adversary using a new a'_i as seen below. We view this extension as mild since it adheres to a similar precedence of clique-security seen in Boneh et al. [BHHO08], which establishes that providing multiple ciphertexts of related secret-dependent messages does not break security of DDH.

Definition 4.5 (Generalized Circular Power-DDH assumption on Paillier Groups [ILL25a]). *The generalized Circular Power-DDH (CP-DDH) assumptions hold on Paillier groups if, for any efficiently computable function $f : \{0, 1\}^* \rightarrow \mathbb{Z}_{\mathcal{N}^{\zeta+1}}^*$,*

$$\left\{ \begin{array}{l} \text{pp}, g, g^s, g^{s^2}, g^{a_i}, g^{sa_i}, g^{s^2 a_i} (1 + \mathcal{N})^{s[i]}, \\ g^{a'_i}, g^{sa'_i} (1 + \mathcal{N})^{f(s[i])} \\ \text{for } i \in [\log \mathcal{N}] \end{array} \middle| \begin{array}{l} \text{pp} \leftarrow \text{Pai.Gen}(1^\lambda, 1^\zeta), \\ g \leftarrow \text{Pai.Samp}(\text{pp}), s, \{a_i, a'_i\}_{i \in [\log \mathcal{N}]} \leftarrow [\mathcal{N}] \end{array} \right\}_\lambda$$

$$\approx_c \left\{ \begin{array}{l} \text{pp}, g, g^s, g^d, g^{a_i}, g^{b_i}, g^{c_i}, g^{a'_i}, g^{b'_i} \\ \text{for } i \in [\log \mathcal{N}] \end{array} \middle| \begin{array}{l} \text{pp} \leftarrow \text{Pai.Gen}(1^\lambda, 1^\zeta), \\ g \leftarrow \text{Pai.Samp}(\text{pp}), s, d, \{a_i, b_i, c_i, a'_i, b'_i\}_{i \in [\log \mathcal{N}]} \leftarrow [\mathcal{N}^{\zeta+2}] \end{array} \right\}_\lambda.$$

Notice that s is a small exponent in $\mathcal{N}$ (which is smaller than the group).

Definition 4.6 (Generalized Circular Power-DDH assumption on prime-order groups [ILL25a]). *The generalized CP-DDH holds on prime-order groups if, for any efficiently computable function $f : \{0, 1\}^* \rightarrow \mathbb{Z}_p$,*

$$\left\{ \begin{array}{l} \text{pp}, g, g^s, g^{s^2}, g^{a_i}, g^{sa_i}, g^{s^2 a_i + s[i]}, \\ g^{a'_i}, g^{sa'_i + f(s[i])} \\ \text{for } i \in [\log p] \end{array} \middle| \begin{array}{l} \text{pp} = (G, p, g) \leftarrow \text{Pri.Gen}(1^\lambda), \\ s, \{a_i\}_{i \in [\log p]} \leftarrow \mathbb{Z}_p \end{array} \right\}_\lambda$$

$$\approx_c \left\{ \begin{array}{l} \text{pp}, g, g^s, g^d, g^{a_i}, g^{b_i}, g^{c_i}, g^{a'_i}, g^{b'_i} \\ \text{for } i \in [\log p] \end{array} \middle| \begin{array}{l} \text{pp} = (G, p, g) \leftarrow \text{Pri.Gen}(1^\lambda), \\ s, d, \{a_i, b_i, c_i, a'_i, b'_i\}_{i \in [\log p]} \leftarrow \mathbb{Z}_p \end{array} \right\}_\lambda.$$

We will use a variant of Ring Learning-With-Errors (RingLWE) assumption, which is similar to the circular power RingLWE of Ishai, Li, and Lin [ILL25b, Definition 9], but the only difference is that an additional circular ciphertext is given to the adversary; see the pair $(\mathbf{a}', s \cdot \mathbf{a}' + \mathbf{e}_3 + f(s) \cdot \Delta)$ below. The circular ciphertext, without the circular power tuple of s^2 , has been the base of bootstrapping fully homomorphic encryptions [BV11b]. We thus believe our generalization is mild.

Definition 4.7 (Generalized Circular Power RingLWE assumption). *The generalized Circular Power RingLWE (CP-RLWE) assumption holds for security parameter λ with respect to ring $\mathcal{R}(\lambda)$, moduli $p(\lambda), q(\lambda)$ such that $\Delta = \frac{q}{p}$, error and secret distributions $\mathcal{D}_{\text{err}}(\lambda), \mathcal{D}_s(\lambda)$ if, for every $m(\lambda)$ and for any efficiently computable function $f : \{0, 1\}^* \rightarrow \mathcal{R}_p^m$,*

$$\left\{ \begin{array}{l} \mathbf{a}, s \cdot \mathbf{a} + \mathbf{e}_1, s^2 \cdot \mathbf{a} + \mathbf{e}_2 + s \cdot \Delta, \\ \mathbf{a}', s \cdot \mathbf{a}' + \mathbf{e}_3 + f(s) \cdot \Delta \\ \text{(over } \mathcal{R}_q) \end{array} \middle| \begin{array}{l} s \leftarrow \mathcal{D}_s, \\ \mathbf{a}, \mathbf{a}' \leftarrow \mathcal{R}_q^m, \\ \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \leftarrow \mathcal{D}_{\text{err}}^m \end{array} \right\}_\lambda \approx_c \{ \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{a}', \mathbf{b}' \leftarrow \mathcal{R}_q^m \}_\lambda.$$

4.3 Homomorphic Tools

Our work relies heavily on many homomorphic tools from homomorphic secret-sharing (HSS) literature [BGI16, BKS19, ADOS22, ILL25a, ILL25b, LLL26]. In this section, we give definitions for the tools we use. For brevity, we refer the readers to Ishai, Li, and Lin [ILL25b] for a more detailed summary of the tools here with their respective assumptions.

To describe these tools, we use notation commonly seen in HSS literature. For a value x , we write $\langle x \rangle_P$ to denote additive shares of x held by parties $P \in \{G, E\}$, where P in our context is either the garbler G or the evaluator E . Thus, additive shares are defined as $x = \langle x \rangle_G + \langle x \rangle_E$. We say additive shares of x are authenticated by a global secret s if $\langle s \cdot x \rangle_G = k$ is a MAC key, and $\langle s \cdot x \rangle_E = k + s \cdot x$ is a tag authenticating x , where the evaluator does not know the secret s . We use $\langle s \cdot \mathbf{x} \rangle_P$ to denote a series of additive shares authenticating each value x_i , where $\mathbf{x} = (x_1, x_2, \dots, x_\ell)$ for $i \in [\ell]$.

Definition 4.8 (Algebraic Homomorphic MACs [ILL25a]). *An algebraic homomorphic MAC (aHMAC) scheme is defined by a tuple of efficient algorithms:*

- **KeyGen**($1^\lambda, s \in [2^\lambda]$): takes a global secret s and outputs a secret key $\mathbf{sk}$ and an evaluation key $\mathbf{evk}$.
- **Auth**($\mathbf{sk}, x \in \{0, 1\}, \text{id} \in \{0, 1\}^\lambda$): takes the secret key $\mathbf{sk}$, a bit to be authenticated x , and an id for the bit. Outputs a tag $\langle s \cdot x \rangle_E = \langle s \cdot x \rangle_G + s \cdot x$.
- **EvalTag**($\mathbf{evk}, C, \mathbf{x}, \langle s \cdot \mathbf{x} \rangle_E$): takes an evaluation key $\mathbf{evk}$, a circuit $C : \{0, 1\}^{|\mathbf{x}|} \rightarrow \{0, 1\}^{|\mathbf{y}|}$, valid input $\mathbf{x}$, and their tags $\langle s \cdot \mathbf{x} \rangle_E$. It outputs tags $\langle s \cdot \mathbf{y} \rangle_E$ authenticating $C(\mathbf{x})$.
- **EvalKey**($\mathbf{sk}, C, \text{id}$): takes secret key $\mathbf{sk}$, the circuit C , and the identities id for the inputs, and outputs MAC keys $\langle s \cdot \mathbf{y} \rangle_G \in \mathbb{Z}^{|\mathbf{y}|}$.

δ -Correctness: Let $x_i \in \{0, 1\}$ for $i \in [\ell]$ and let $\mathbf{x} = (x_0, x_2, \dots, x_{\ell-1})$. An aHMAC scheme is correct with some error bound $\delta(\lambda)$ if, for some negligible function $\text{negl}(\lambda)$,

$$\Pr \left[\langle s \cdot \mathbf{y} \rangle_E = \langle s \cdot \mathbf{y} \rangle_G + s \cdot C(\mathbf{x}) \mid \begin{array}{l} (\mathbf{sk}, \mathbf{evk}) \leftarrow \text{KeyGen}(1^\lambda, s) \\ \langle x_i \rangle_E \leftarrow \text{Auth}(\mathbf{sk}, x_i, \text{id}_i) \\ \langle s \cdot \mathbf{y} \rangle_E \leftarrow \text{EvalTag}(\mathbf{evk}, C, \mathbf{x}, \langle s \cdot \mathbf{x} \rangle_E) \\ \langle s \cdot \mathbf{y} \rangle_G \leftarrow \text{EvalKey}(\mathbf{sk}, C, \text{id}) \end{array} \right] \geq 1 - \delta(\lambda) - \text{negl}(\lambda).$$

Selective security: An aHMAC scheme is secure if there exists an efficient simulator Sim such that, for every sequence of global secrets $\{s = s_\lambda\}_\lambda$, inputs $\{\mathbf{x} = \mathbf{x}_\lambda\}_\lambda$, and $\{\text{id} = \text{id}_\lambda\}_\lambda$,

$$\left\{ \text{Sim}(1^\lambda, 1^{|\mathbf{y}|}) \right\}_\lambda \approx_c \left\{ \mathbf{evk}, \langle s \cdot \mathbf{x} \rangle_E \mid \begin{array}{l} (\mathbf{sk}, \mathbf{evk}) \leftarrow \text{KeyGen}(1^\lambda, s) \\ \langle s \cdot x_i \rangle_E \leftarrow \text{Auth}(\mathbf{sk}, x_i, \text{id}_i) \end{array} \right\}_\lambda.$$

Definition 4.9 (Extended Homomorphic Secret Sharing [BKS19]). *An extended homomorphic secret sharing (HSS) scheme is defined by a tuple of efficient algorithms:*

- **ExtEval_P**($\text{pd}_\mathbf{y}, F, \mathbf{w}_P^x, \mathbf{v}_P^x$): for party $P \in \{G, E\}$, takes as input public data $\text{pd}_\mathbf{y}$ with respect to some $\mathbf{y}$, an NC1 circuit $F : \{0, 1\}^{|\mathbf{y}|} \rightarrow \{0, 1\}^{|\mathbf{x}|}$, and two vectors $\mathbf{w}_P^x, \mathbf{v}_P^x$. It outputs a pair of integers $w_P^z, v_P^z \in \mathbb{Z}$.

δ -Correctness: An extended HSS scheme is correct with some error bound $\delta(\lambda)$ if, for secret s with additive shares $(\langle s \cdot \mathbf{x} \rangle_P, \langle \mathbf{x} \rangle_P)$ and for some negligible function $\text{negl}(\lambda)$,

$$\Pr \left[\begin{array}{c} w_E^z = w_G^z + s \cdot z \\ v_E^z = v_G^z + z \\ \text{(over } \mathbb{Z}) \end{array} \middle| \begin{array}{c} w_P^z, v_P^z \leftarrow \text{ExtEval}_P(\text{pd}_{\mathbf{y}}, F, \langle s \cdot \mathbf{x} \rangle_P, \langle \mathbf{x} \rangle_P) \\ z := \text{InnerProd}(\mathbf{x}, F(\mathbf{y})) \\ \text{(over } \mathbb{Z}) \end{array} \right] \geq 1 - \delta(\lambda) - \text{negl}(\lambda).$$

Security: An extended HSS scheme is secure if the public data $\text{pd}_{\mathbf{y}}$ computationally hides $\mathbf{y}$ and the encodings $\langle s \cdot \mathbf{x} \rangle_P, \langle \mathbf{x} \rangle_P$ computationally hide inputs $\mathbf{x}$ and secret s except with negligible probability in λ .

We will use a fixed factor hashing by Li, Lin, and Lu [LLL26] in a non-black-box-way. Thus, we define the syntax and correctness here, and later we will prove the security of our scheme directly using the DDH/RingLWE assumptions.

Definition 4.10 (Fixed factor hashing [LLL26]). A fixed factor hashing scheme is defined by a tuple of efficient algorithms for hash function $\text{Hash}(\mathbf{x}) := \sum_i b_i \cdot \mathbf{x}[i]$, where b_i are public hash coefficients:

- **Setup**($1^\lambda, s, s'$): takes a security parameter λ and a pair of secrets s, s' . Outputs public data pd containing hash information b_i and extraction hints.
- **Compress**($\text{pd}, \langle s \cdot \mathbf{x} \rangle_P$): takes as input public data pd with authenticated share $\langle s \cdot \mathbf{x} \rangle_P$ for party $P \in \{G, E\}$. Outputs authenticated hash share $\langle s \cdot h \rangle_P$ for $h = \text{Hash}(\mathbf{x})$.
- **Extract** $_P(\text{pd}, \langle s \cdot h \rangle_P, j, \langle \mathbf{x} \rangle_P)$: takes as input public data pd with authenticated hash share $\langle s \cdot h \rangle_P$ and additive share $\langle \mathbf{x} \rangle_P$, where party G 's share $\langle \mathbf{x} \rangle_G$ is empty string by construction. Outputs the new authenticated share for the j -th index $\langle s' \cdot x_j \rangle'_P$.

δ -Correctness: A fixed factor hashing scheme is correct with some error bound $\delta(\lambda)$ if, for secrets s, s' , additive shares $\langle s \cdot \mathbf{x} \rangle_P, \langle \mathbf{x} \rangle_P$, and some negligible function $\text{negl}(\lambda)$,

$$\Pr \left[\begin{array}{c} \langle s' \cdot x_j \rangle'_E = \langle s' \cdot x_j \rangle'_G + s' \cdot x_j \\ \text{for all } j \end{array} \middle| \begin{array}{c} \text{pd} \leftarrow \text{Setup}(1^\lambda, s, s') \\ \langle s \cdot h \rangle_P \leftarrow \text{Compress}(\text{pd}, \langle s \cdot \mathbf{x} \rangle_P) \\ \langle s' \cdot x_j \rangle'_P \leftarrow \text{Extract}_P(\text{pd}, \langle s \cdot h \rangle_P, j, \langle \mathbf{x} \rangle_P) \end{array} \right] \geq 1 - \delta(\lambda) - \text{negl}(\lambda).$$

The above definition is slightly stronger than that of Li, Lin, and Lu [LLL26, Construction 5.15]. The main difference is that our hash h is a scalar value, where as the previous work left the hash as a short vector. That makes our shares $\langle s \cdot h \rangle_P$ shorter and saves communication. Another difference is that we use the same secret $s' = s$, which makes our constructions rely on the circular security of DDH/RingLWE later. We alter the construction slightly from the previous work, and the alteration is deferred to Section 8.

4.4 Homomorphic Gate

A core component for our work is built from the homomorphic tools, HGate, by Ishai, Li, and Lin [ILL25b]. It combines aHMAC and HSS in order to homomorphically evaluate authenticated shares on a Boolean circuit that takes $O(\log \lambda)$ -bit input and $O(\log \lambda)$ circuit depth. Each HGate takes communication equal to the bit-length of the circuit's output. It is an important part of

the construction of our routing gate, so we describe it and show the protocol in Algorithm 1 for completeness.

The main idea of **HGate** is to allow the garbler and evaluator to get authenticated shares of some function. As written in previous works [ILL25b, LLL26], this function is typically a Boolean operation, i.e., **AND**, **XOR**, **NOT**. The main challenge is maintaining the consistency of the masking logic throughout the circuit. That is, for a garbling scheme to be considered secure, we can not allow the evaluator to learn the true inputs to the circuit. This is solved by having the garbler select a mask associated with each wire in advance that acts as a one-time pad to each wire value such that, for wire value $x \in \{0, 1\}$ and it's associated mask $r_x \in \{0, 1\}$, the evaluator receives $\bar{x} = x \oplus r_x$. This means that **HGate** needs to handle these masks while performing homomorphic operations.

To address this, Ishai, Li, and Lin [ILL25b] opt to select the mask for each wire via a PRF. This PRF takes as input a wire id and outputs a bit mask according to a secret key sk . Since the goal is to place a new mask on the output of a particular gate, we need a way to allow the evaluator to evaluate the PRF without learning sk . This is where they use the HSS extension technique seen in Definition 4.9, where the input public data encrypts sk . Encrypting the secret key for the PRF allows the evaluator to use HSS in order to evaluate a function that takes sk as input without learning it. The following function allows each party to compute gate operations as expected for masked input $\bar{\mathbf{x}}$:

$$F_g^{\text{hgate}}(\text{sk}) = \{\text{PRF}_{\text{sk}}(\text{ID}(g, \text{out})) \oplus \mathbf{g}(\mathbf{v} \oplus \text{PRF}_{\text{sk}}(\text{ID}(g, \text{in})))\}_{\mathbf{v} \in \{0,1\}^{|\mathbf{x}|}},$$

where g is identifier for the gate in the circuit, $\mathbf{g}$ is the operation of the gate, $\text{ID}(g, \text{in})$ are the IDs of the inputs wires of the gate, and $\text{ID}(g, \text{out})$ is the ID of the output wire. This function successfully simulates all possible inputs, $\mathbf{v}$, with their associated masks and places a new output mask that the garbler knows.

The remaining piece is that we need to be able to select the correct entry from this function's truth table for the particular inputs the evaluator receives. Here is where they use aHMAC as in Definition 4.8. Specifically, the idea is to use shares of a one-hot vector as input to the HSS function to do the selection. The function used in aHMAC is a one-hot function $\mathbf{1}_{\mathbf{v}}$ where, for some $\mathbf{v} \in \{0, 1\}^{|\mathbf{x}|}$, outputs 1 iff the input is $\mathbf{v}$ (i.e., $\mathbf{1}_{\mathbf{v}}(i) = 0$ for all $i \in \{0, 1\}^{|\mathbf{x}|}$ where $i \neq \mathbf{v}$, but $\mathbf{1}_{\mathbf{v}}(i) = 1$ if $i = \mathbf{v}$). Since it is evaluated via aHMAC, the output is authenticated shares, which we refer to all together as a one-hot encoding.

Combining the ideas above is what gets the 1-bit per gate succinct garbling as seen in Ishai, Li, and Lin [ILL25b], where the garbler needs to send their unauthenticated output of the HSS to the evaluator so that they are able to recover the clear masked output bit $\bar{z}$, where z is the plaintext output of the function F . We use this **HGate** not only for Boolean operations in our dynamic circuit, but also in our **Route** gates, which requires a slightly more modular algorithm for **HGate** that is reflected in Algorithm 1 in which the function F to be evaluated via HSS is given as input, and the input/output sizes are determined by this function F . We also leave out the masked notation in the figure for this reason. The public data pd_{sk} includes public data for both aHMAC and HSS, where the HSS public data encodes a the PRF secret key sk .

Algorithm 1. HGate [ILL25b]

HGate.Garble($\text{pd}_{\text{sk}}, F, \langle s \cdot \mathbf{x} \rangle_G$) :

1. For $\mathbf{v} \in \{0, 1\}^{|\mathbf{x}|}$, compute one-hot encoding shares

$$\langle s \cdot \mathbb{1}_{\mathbf{v}}(\mathbf{x}) \rangle_G \leftarrow \text{EvalKey}(\text{aHMAC.pd}, \mathbb{1}_{\mathbf{v}}, \langle s \cdot \mathbf{x} \rangle_G).$$

2. Compute shares on function F

$$(\langle s \cdot z \rangle_G, \langle z \rangle_G) \leftarrow \text{ExtEval}_G \left(\text{HSS.pd}_{\text{sk}}, F, \{ \langle s \cdot \mathbb{1}_{\mathbf{v}}(\mathbf{x}) \rangle_G, 0 \}_{\mathbf{v} \in \{0, 1\}^{|\mathbf{x}|}} \right).$$

3. Output $(\langle s \cdot z \rangle_G \ominus \langle z \rangle_G)$.

HGate.Eval($\text{pd}_{\text{sk}}, F, \mathbf{x}, \langle s \cdot \mathbf{x} \rangle_E \ominus \langle z \rangle_G$) :

1. For $\mathbf{v} \in \{0, 1\}^{|\mathbf{x}|}$, compute one-hot encoding shares

$$\langle s \cdot \mathbb{1}_{\mathbf{v}}(\mathbf{x}) \rangle_E \leftarrow \text{EvalTag}(\text{aHMAC.pd}, \mathbb{1}_{\mathbf{v}}, \langle s \cdot \mathbf{x} \rangle_E, \mathbf{x}).$$

2. Compute shares on the function F

$$(\langle s \cdot z \rangle_E, \langle z \rangle_E) \leftarrow \text{ExtEval}_E \left(\text{HSS.pd}_{\text{sk}}, F, \{ \langle s \cdot \mathbb{1}_{\mathbf{v}}(\mathbf{x}) \rangle_E, \mathbb{1}_{\mathbf{v}}(\mathbf{x}) \}_{\mathbf{v} \in \{0, 1\}^{|\mathbf{x}|}} \right).$$

3. Set

$$z \leftarrow \langle z \rangle_G + \langle z \rangle_E.$$

4. Output $(z, \langle s \cdot z \rangle_E)$.

HGate.Sim($\text{pd}_{\text{sk}}, F, \mathbf{x}, \langle s \cdot \mathbf{x} \rangle_E, z$) :

1. For $\mathbf{v} \in \{0, 1\}^{|\mathbf{x}|}$, compute one-hot encoding shares

$$\langle s \cdot \mathbb{1}_{\mathbf{v}}(\mathbf{x}) \rangle_E \leftarrow \text{EvalTag}(\text{aHMAC.pd}, \mathbb{1}_{\mathbf{v}}, \langle s \cdot \mathbf{x} \rangle_E, \mathbf{x}).$$

2. Compute shares on the function F

$$(\langle s \cdot z \rangle_E, \langle z \rangle_E) \leftarrow \text{ExtEval}_E \left(\text{HSS.pd}_{\text{sk}}, F, \{ \langle s \cdot \mathbb{1}_{\mathbf{v}}(\mathbf{x}) \rangle_E, \mathbb{1}_{\mathbf{v}}(\mathbf{x}) \}_{\mathbf{v} \in \{0, 1\}^{|\mathbf{x}|}} \right).$$

3. Set

$$\langle z \rangle_G \leftarrow z - \langle z \rangle_E,$$

4. Output $(z, \langle s \cdot z \rangle_E \ominus \langle z \rangle_G)$.

Theorem 4.11 (Garbling Boolean Gates). *Let λ be the security parameter. Under any of the following assumption: (1) Circular Power-DDH in Paillier groups, (2) a variant of Circular Power-DDH in prime-order groups, or (3) Circular Power-RLWE, there exists a secure garbled boolean gate with communication cost only 1 bit per gate, where λ depends on the assumption used; in addition, the garbling takes a one-time communication of $\text{poly}(\lambda)$ bits for any number of boolean gates. The computation time is $\text{poly}(\lambda)$ per gate for both the garbler and evaluator.*

We remark that the above F_g^{hgate} was designed to evaluate the corresponding boolean gate g , but

we will use HGate with other functions F that do not correspond to any boolean gate.

4.5 Dynamic Circuit

We define a dynamic circuit similar to Zebra [GGS26] but with the relevant gates. In general, a dynamic circuit should allow eager evaluation which allows certain gates to evaluate even without all input wires being set to better suit the RAM computation model.

Definition 4.12 (Dynamic Circuit). *A dynamic circuit C consists of wires connecting the following types of gates:*

- *Standard boolean gates: Given an operation $\text{op} \in \{\text{AND}, \text{XOR}, \text{NOT}\}$ and shares of encodings for values x, y , outputs shares of encoding for the evaluation $\text{op}(x, y)$, whereas $\text{op} = \text{NOT}$ takes only 1-bit input.*
- *Route For n cables consisting of ℓ wires, a routing gate takes as input cables $\mathbf{x}_0, \dots, \mathbf{x}_{n-1}$ and control wires $\pi_0, \dots, \pi_{n-1}$ with output cables $\mathbf{y}_0, \dots, \mathbf{y}_{n-1}$. For $i \in [n]$, if $\mathbf{x}_i$ and π_i are set, then $\mathbf{y}_{\pi_i}$ is set to the value on cable $\mathbf{x}_i$.*
- *ReverseRoute For n cables consisting of ℓ wires, a reverse routing gate takes as input cables $\mathbf{y}_0, \dots, \mathbf{y}_{n-1}$ and control wires $\pi_0, \dots, \pi_{n-1}$ with output cables $\mathbf{x}_0, \dots, \mathbf{x}_{n-1}$. For $i \in [n]$, if π_i and $\mathbf{y}_{\pi_i}$ are set, then $\mathbf{x}_i$ is set to the value on cable $\mathbf{y}_{\pi_i}$.*

Remark 4.13 (Route and ReverseRoute). Notice that ReverseRoute differs from Route since, in ReverseRoute, the control π_i dynamically chooses the *input* $\mathbf{y}_{\pi_i}$, while in Route, it chooses the *output* dynamically. Still, as observed in prior works [GGS26, LLLH26], their constructions are symmetric, so we focus on Route in our work.

5 Constructing Garbled Dynamic Circuits

In this section, we describe the construction for Route and how to compose it to achieve a garbled dynamic circuit as defined by Definition 4.12.

5.1 Routing Gate

Recall that a Route gadget takes up to n input cables with ℓ wires in each cable, n control wires, and n output cables. Both parties hold public data pd which depends on the assumption used. They also hold authenticated additive shares $\langle s \cdot x_{i,t} \rangle_P$ that encode wire data $x_{i,t}$, where $i \in [n]$ are cables, $t \in [\ell]$ are wires in each cable, and $P \in \{G, E\}$ represent the party that holds the share. The evaluator additionally holds wire data $x_{i,t}$ and control wire data π_i in the clear. Each party has authenticated shares $\langle s \cdot \pi_i \rangle_P$ for each control wire. In the final construction, each wire data $x_{i,t}$ is masked by a one-time pad. To simplify the presentation, we write $x_{i,t}$ directly and address the masks later. Recall the ultimate goal of the Route is to redirect each input cable to a new output cable based on the control wire for that input cable. We want to preserve authenticated additive shares between the garbler and the evaluator with the correct data on each wire. Notably, our Route is eager evaluating, as the routing by the evaluator can be performed one cable at a time depending on which wires are set.

Our routing gate construction saves communication in two main ways: compression of data wires and local computation of routing material. The high level idea is to compress the wires for each cable into a succinct representation independent of the number of wires per cable. This way, when the parties perform homomorphic evaluation to get the routing material for each cable based on the control wire values, the garbler only needs to send material for each cable rather than for each wire, whose size is the same as an authenticated share. Each party should then be able to recover authenticated shares for each wire after the compressed cables have been routed.

For the compression, we borrow fixed factor hashing from the low-depth succinct garbling work of Li, Lin, and Lu [LLL26] as in Definition 4.10. Recall that, given the authenticated shares for each wire value in a cable and the public data `Hash.pd`, the parties compute

$$G : \langle s \cdot h_i \rangle_G \leftarrow \text{Compress}(\text{Hash.pd}, \{\langle s \cdot x_{i,t} \rangle_G\}_{t \in [\ell]}),$$

$$E : \langle s \cdot h_i \rangle_E \leftarrow \text{Compress}(\text{Hash.pd}, \{\langle s \cdot x_{i,t} \rangle_E\}_{t \in [\ell]}).$$

An important feature of these compressed hashes is that they support linear operations homomorphically. This allows us to essentially rekey the evaluators tags while they are compressed by removing the original key and replacing it with the key associated with the new cable as per the control wires. When we later try to retrieve the wire authenticated shares, this rekeying process will be reflected in both parties' shares.

Once compressed, the next step is to get the routing material locally with as little communication as possible. To do so, we use `HGate` (Algorithm 1) from the succinct garbling framework of Ishai, Li, and Lin [ILL25b]. As mentioned previously, these homomorphic gates use `aHMAC` and `HSS` (Definitions 4.8 and 4.9) in order to locally compute authenticated shares of some output based on material encoded in the public data and the function being evaluated. The public data encodes a PRF secret key that enables the evaluator to homomorphically evaluate PRF-derived functions without revealing the key itself. By construction, the communication required per `HGate` is only as large as the size of the output, as is necessary for the evaluator to receive the value being authenticated in the clear for future computation.

As for how we use `HGate` in `Route`, we start with the shares from the previous gates for each wire and compress them. To take advantage of the local computation advantages from `HGate`, we need the keys on each compressed tag to line up with the PRF that will be used in the routing function for `HSS`. To do this, the garbler simply uses their secret key `sk` and the cable ID for each input cable to evaluate the PRF, then takes the difference between this value and the compressed keys they computed for the cable. This needs to be sent to the evaluator so that they are able to switch the key on their compressed tags, requiring only $O(\lambda)$ communication per cable.

Specifically, the PRF is used in the function

$$F_{g,i}^{\text{key}}(\text{sk}) = \{\text{PRF}_{\text{sk}}(\text{ID}(g, \text{out}, \mathbf{v})) - \text{PRF}_{\text{sk}}(\text{ID}(g, \text{in}, i))\}_{\mathbf{v} \in \{0,1\}^{\log n}}, \quad (5)$$

where g is an ID for the gate, and $i \in [n]$ represents input cables. $\mathbf{v}$ represents all the possible output cables for the routing gate, and is built into the function. The secret key `sk` is used for the PRF to output $O(\lambda)$ -sized strings to be used as keys. All together, since the evaluator rekeyed their tags to line up with the PRF, then we can use this function in `HSS` to allow the evaluator to retrieve routing material for each cable.

All that is left is to select the proper entry from the truth table of this function, which is easily solved in **HGate**. To put it in context of **Route**, the aHMAC in **HGate** allows each party to compute an authenticated one-hot encoding based on the control wires π_i :

$$\begin{aligned} G : \langle s \cdot \mathbb{1}_{\mathbf{v}}(\pi_i) \rangle_G &\leftarrow \text{EvalKey}(\text{aHMAC.pd}, \mathbb{1}_{\mathbf{v}}, \langle s \cdot \pi_i \rangle_G), \\ E : \langle s \cdot \mathbb{1}_{\mathbf{v}}(\pi_i) \rangle_E &\leftarrow \text{EvalTag}(\text{aHMAC.pd}, \mathbb{1}_{\mathbf{v}}, \langle s \cdot \pi_i \rangle_E, \pi_i). \end{aligned}$$

Similarly to in Section 4.4, $\mathbb{1}_{\mathbf{v}}(\pi_i)$ is an indicator polynomial that outputs 1 if the one-hot representation of π_i is equal to $\mathbf{v}$, and 0 otherwise. Since it is evaluated via aHMAC, we get a one-hot encoding, which is used as input to HSS to select the necessary routing material from all possible material in $F_{g,i}^{\text{key}}$:

$$\begin{aligned} G : \langle \text{rt}_i \rangle_G &\leftarrow \text{ExtEval}_G(\text{HSS.pd}_{\text{sk}}, F_{g,i}^{\text{key}}, (\langle s \cdot \mathbb{1}_{\mathbf{v}}(\pi_i) \rangle_G, \langle \mathbb{1}_{\mathbf{v}}(\pi_i) \rangle_G)_{\mathbf{v} \in \{0,1\}^{\log n}}), \\ E : \langle \text{rt}_i \rangle_E &\leftarrow \text{ExtEval}_E(\text{HSS.pd}_{\text{sk}}, F_{g,i}^{\text{key}}, (\langle s \cdot \mathbb{1}_{\mathbf{v}}(\pi_i) \rangle_E, \langle \mathbb{1}_{\mathbf{v}}(\pi_i) \rangle_E)_{\mathbf{v} \in \{0,1\}^{\log n}}), \end{aligned}$$

where the garbler's share of $\mathbb{1}_{\mathbf{v}}(\pi_i)$ is always the trivial share $\langle \mathbb{1}_{\mathbf{v}}(\pi_i) \rangle_G = 0$ since they have no π_i to use to evaluate. Notice that in the above, rt_i denotes the routing material for cable i , which depends on the value π_i , and thus it dynamically depends on the input of the dynamic circuit. Putting it clearly, this combination of homomorphic evaluation leads to **HGate** evaluating

$$\text{rt}_i = \sum_{\mathbf{v} \in \{0,1\}^{\log n}} \mathbb{1}_{\mathbf{v}}(\pi_i) \cdot F_{g,i,\mathbf{v}}^{\text{key}}(\text{sk}), \quad (6)$$

where the subscript $\mathbf{v}$ on the function denotes the $\mathbf{v}$ -th position of the function's truth table. Lastly, the garbler needs to send their share of rt_i for each cable in order for the evaluator to get the full routing material, since the evaluator only computes an additive share of the function evaluation. This only costs communication $O(\lambda)$ bits per cable.

Once the compressed tags have been routed properly to align with the new key $\langle s \cdot h_{\pi_i} \rangle_G^{\text{out}}$, each party uses extract to recover authenticated shares for each wire in the routed cable. Since rekeying is a linear operation, then the extracted authenticated shares between each party will still be additive shares, regardless of where each data value was routed to. We can extract an authenticated share for each wire $t \in [\ell]$ in a cable,

$$\begin{aligned} G : \langle s \cdot y_{\pi_i,t} \rangle_G &\leftarrow \text{Extract}_G(\text{Hash.pd}, \langle s \cdot h_{\pi_i} \rangle_G^{\text{out}}, t), \\ E : \langle s \cdot y_{\pi_i,t} \rangle_E &\leftarrow \text{Extract}_E(\text{Hash.pd}, \langle s \cdot h_{\pi_i} \rangle_E^{\text{out}}, t, \{x_{i,t}\}_{t \in [\ell]}). \end{aligned}$$

Dynamically setting wire masks. One final piece to the puzzle that we did not address yet is how to handle the masks on each value. As mentioned in Section 4.4, each wire in the circuit has a predetermined mask value computed by a PRF with a secret key sk' . This sk' will allow the PRF to produce outputs of 1-bit in length. To compose our **Route** with **HGate** as Boolean operations in our dynamic circuit, we need to maintain that the mask on each wire value lines up with the PRF evaluations on sk' . Since we need to remask the data values, then we also need authenticated shares that line up with these new masked bits. Luckily, this is easy to fix with another call to **HGate** with a different set of HSS public data and an additional input. The function to evaluate is

$$F_{g,i,t}^{\text{mask}}(\text{sk}') = \{\mathbf{v}_2 \oplus \text{PRF}_{\text{sk}'}(\text{ID}(g, \text{out}, \mathbf{v}_1, t)) \oplus \text{PRF}_{\text{sk}'}(\text{ID}(g, \text{in}, i, t))\}_{\mathbf{v}_1 \in \{0,1\}^{\log n}, \mathbf{v}_2 \in \{0,1\}}, \quad (7)$$

where $t \in [\ell]$ is the wire in cable i . Here, $\mathbf{v} = (\mathbf{v}_1, \mathbf{v}_2)$ is a vector that we split into two parts. The first part, $\mathbf{v}_1$, is the same as in the HGate above, where it represents the control wire π_i . The second part, $\mathbf{v}_2$, represents the masked bit $\hat{x}_{i,t} = x_{i,t} + r_{i,t}$. Thus, the inputs to this are

$$(\langle s \cdot \bar{x}_{i,t} \rangle_G \leftarrow \langle \bar{x}_{i,t} \rangle_G) \leftarrow \text{HGate.Garble}(\text{pd}_{\text{sk}'}, F_{g,i,t}^{\text{mask}}, \{\langle s \cdot \pi_i \rangle_G, \langle s \cdot \hat{x}_{i,t} \rangle_G\}),$$

$$(\bar{x}_{i,t}, \langle s \cdot \bar{x}_{i,t} \rangle_E) \leftarrow \text{HGate.Eval}(\text{pd}_{\text{sk}'}, F_{g,i,t}^{\text{mask}}, \{\pi_i, \hat{x}_{i,t}\}, \{\langle s \cdot \pi_i \rangle_E, \langle s \cdot \hat{x}_{i,t} \rangle_E\} \leftarrow \langle \bar{x}_{i,t} \rangle_G),$$

where $\bar{x}_{i,t}$ represents the newly masked representation of $x_{i,t}$. We also use pd'_{sk} as input to HGate to denote the aHMAC and HSS public data associated with sk' , while pd_{sk} is the aHMAC and HSS public data associated with sk later. The garbler once again needs to send their share of $\bar{x}_{i,t}$ to reveal it to the evaluator in the clear, which costs an additional $O(\ell)$ per cable.

Algorithm 2 details the steps for the protocol of routing gates. Notice that, in the garbling procedure, we describe `Route.Garble` to garble all n cables at once. In contrast, the evaluating procedure describes `Route.EvalCable` for routing individual cables $i \in [n]$. The difference is necessary because the garbler did not know which cable would be evaluated in which order, and because the evaluator needs to fulfill the eager evaluating requirement for each individual cable.

Theorem 5.1 (Garbling Routing Gates). *Let $n, \ell \in \mathbb{N}$ with security parameter λ . Under any of the following assumption: (1) Circular Power-DDH in Paillier groups, (2) a variant of Circular Power-DDH in prime-order groups, or (3) Circular Power-RLWE, there exists a secure garbled routing gate on n cables of ℓ wires each that realizes any routing permutation of cables with communication cost $O(\lambda + \ell)$ bits per cable, where λ depends on the assumption used; in addition, the garbling takes a one-time communication of $O(\ell^2 \cdot \text{poly}(\lambda))$ bits for any number of routing instances. The computation time is $O((\ell^2 + n\ell) \cdot \text{poly}(\lambda))$ per cable for both the garbler and evaluator.*

Remark 5.2. Since the `ReverseRoute` is constructed symmetrically to `Route`, the bounds in Theorem 5.1 apply to `ReverseRoute` as well. In particular, for the same parameters n and ℓ , it has the same communication and computation complexity.

Algorithm 2. Route

`Route.Garble`($\text{pd}, \text{sk}, \{\langle s \cdot \hat{x}_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_G\}_{i \in [n]})$:

1. For $i \in [n]$ and $t \in [\ell]$, compute remasked shares with $F_{g,i,t}^{\text{mask}}$ defined as in Equation 7

$$(\langle s \cdot \bar{x}_{i,t} \rangle_G \leftarrow \langle \bar{x}_{i,t} \rangle_G) \leftarrow \text{HGate.Garble}(\text{pd}_{\text{sk}'}, F_{g,i,t}^{\text{mask}}, \{\langle s \cdot \pi_i \rangle_G, \langle s \cdot \hat{x}_{i,t} \rangle_G\}).$$

2. For $i \in [n]$, compute compressions

$$\langle s \cdot h_i \rangle_G \leftarrow \text{Compress}(\text{Hash.pd}, \{\langle s \cdot \bar{x}_{i,t} \rangle_G\}_{t \in [\ell]}).$$

3. For $i, j \in [n]$, sample target hashes

$$\begin{aligned} \langle s \cdot h_i \rangle'_G &\leftarrow \text{PRF}_{\text{sk}}(\text{ID}(g, \text{in}, i)), \\ \langle s \cdot h_j \rangle_G^{\text{out}} &\leftarrow \text{PRF}_{\text{sk}}(\text{ID}(g, \text{out}, j)). \end{aligned}$$

4. For $i \in [n]$, set rekeying materials

$$\text{rk}_i \leftarrow \langle s \cdot h_i \rangle'_G - \langle s \cdot h_i \rangle_G.$$

5. For $i \in [n]$, compute routing shares with $F_{g,i}^{\text{key}}$ defined as in Equation 5

$$(\langle s \cdot \text{rt}_i \rangle_G \leftarrow \langle \text{rt}_i \rangle_G) \leftarrow \text{HGate.Garble}(\text{pd}_{\text{sk}}, F_{g,i}^{\text{key}}, \langle s \cdot \pi_i \rangle_G).$$

6. For each $j \in [n]$ and $t \in [\ell]$, compute output shares

$$\langle s \cdot \bar{y}_{j,t} \rangle_G \leftarrow \text{Extract}_G(\text{Hash.pd}, \langle s \cdot h_j \rangle_G^{\text{out}}, t).$$

7. Output $(\{\langle s \cdot \bar{y}_{j,t} \rangle_G\}_{j \in [n], t \in [\ell]} \leftarrow \{\langle \bar{x}_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}, \{\text{rk}_i\}_{i \in [n]}, \{\langle \text{rt}_i \rangle_G\}_{i \in [n]}).$

Route.EvalCable(pd, $i \in [n]$, $\{\hat{x}_{i,t}\}_{t \in [\ell]}$, π_i , $\{\langle s \cdot x_{i,t} \rangle_E\}_{t \in [\ell]}$, $\langle s \cdot \pi_i \rangle_E \leftarrow \{\langle \bar{x}_{i,t} \rangle_G\}_{t \in [\ell]}$, rk_i , $\langle \text{rt}_i \rangle_G$) :

1. For $t \in [\ell]$, compute remasked shares with $F_{g,i,t}^{\text{mask}}$ defined as in Equation 7

$$(\bar{x}_{i,t}, \langle s \cdot \bar{x}_{i,t} \rangle_E) \leftarrow \text{HGate.Eval}(\text{pd}_{\text{sk}'}, F_{g,i,t}^{\text{mask}}, \{\pi_i, \hat{x}_{i,t}\}, \{\langle s \cdot \pi_i \rangle_E, \langle s \cdot \hat{x}_{i,t} \rangle_E\} \leftarrow \langle \bar{x}_{i,t} \rangle_G).$$

2. Compute compression and rekey

$$\langle s \cdot h_i \rangle_E \leftarrow \text{Compress}(\text{Hash.pd}, \{\langle s \cdot \bar{x}_{i,t} \rangle_E\}_{t \in [\ell]}),$$

$$\langle s \cdot h_i \rangle'_E \leftarrow \langle s \cdot h_i \rangle_E + \text{rk}_i.$$

3. Compute routing share with $F_{g,i}^{\text{key}}$ defined as in Equation 5

$$(\text{rt}_i, \langle s \cdot \text{rt}_i \rangle_E) \leftarrow \text{HGate.Eval}(\text{pd}_{\text{sk}}, F_{g,i}^{\text{key}}, \pi_i, \langle s \cdot \pi_i \rangle_E \leftarrow \langle \text{rt}_i \rangle_G).$$

4. Set route as

$$\langle s \cdot h_{\pi_i} \rangle_E^{\text{out}} \leftarrow \langle s \cdot h_i \rangle'_E + \text{rt}_i.$$

5. For each $t \in [\ell]$, compute output shares

$$\langle s \cdot \bar{y}_{\pi_i,t} \rangle_E \leftarrow \text{Extract}_E(\text{Hash.pd}, \langle s \cdot h_{\pi_i} \rangle_E^{\text{out}}, t, \{\bar{x}_{i,t}\}_{t \in [\ell]}).$$

6. For each $t \in [\ell]$, set

$$\bar{y}_{\pi_i,t} \leftarrow \bar{x}_{i,t}$$

7. Output $(\langle s \cdot \bar{y}_{\pi_i,t} \rangle_E, \bar{y}_{\pi_i,t}).$

The correctness is easy to see from the correctness of each of the pieces we use, particularly the HGate and fixed factor hashing. Based on the functions used in each HGate, the garbler will get fixed authenticated shares for each output wire of the Route. The evaluator will get an authenticated share based on the fixed share that the garbler holds, which allows them to route a wire based on the control wires. By construction of Compress and Extract, using these does not affect the correctness, so the result of these protocols will keep the additive authenticated relationship of the shares.

For completeness, we show how we use HGate to compute Boolean operations, which is the same

as in Ishai, Li, and Lin [ILL25b]:

$$F_g^{\text{hgate}}(\text{sk}') = \{\text{PRF}_{\text{sk}'}(\text{ID}(g, \text{out})) \oplus \mathbf{g}(\mathbf{v} \oplus \text{PRF}_{\text{sk}'}(\text{ID}(g, \text{in})))\}_{\mathbf{v} \in \{0,1\}^2},$$

where g is the gate ID, $i, j \in [n]$ represent input/output cables of the gate respectively, $t, u \in [\ell]$ represent the input/output wires of the gate respectively, and $\mathbf{g} \in \text{AND}, \text{XOR}$ denotes the operation of the gate.

Additionally, for PRF secret keys sk, sk' , the public data is sampled as

$$\begin{aligned} \text{aHMAC.pd} &\leftarrow \text{aHMAC.Setup}(\text{pp}, s, s), \text{Hash.pd} \leftarrow \text{Hash.Setup}(\text{pp}, s, s), \\ \text{HSS.pd}_{\text{sk}} &\leftarrow \text{HSS.Setup}(\text{pp}, s, \text{sk}), \text{HSS.pd}_{\text{sk}'} \leftarrow \text{HSS.Setup}(\text{pp}, s, \text{sk}'), \end{aligned} \quad (8)$$

$$\text{pd} = (\text{aHMAC.pd}, \text{HSS.pd}_{\text{sk}}, \text{HSS.pd}_{\text{sk}'}, \text{Hash.pd}),$$

where the sampling method depends on the assumption, which can be seen in the next sections on the instantiations under each assumption.

5.2 Garbling Dynamic Circuits

Algorithms 1 and 2 describe the garbling of individual Boolean gates and Route gates. As written, the garbling algorithms depend on *all* shares of the gate's input. When the circuit was an acyclic graph, as in standard circuits, such input dependencies are not an issue. However, for dynamic circuits, the circuit graph can be *cyclic*, and indeed, we built cyclic dynamic circuits to simulate GRAM [HKO23]. Hence, we must be able to garble all dynamic circuits, even if they are cyclic.

To this end, observe that the garbler is able to compute the output shares of Route. It is because in Algorithm 2, the shares $\langle s \cdot \bar{y}_{j,t} \rangle_G$ depends only on the PRF evaluation $\langle s \cdot h_j \rangle_G^{\text{out}}$, which is independent of the input shares. Also, with the output shares of Route, we are able to garble the remaining gates, including the Route gates. Algorithm 3 describes the detailed garbling procedure. We omit the evaluation of the garbling as the ordering is identical to the ordering of the corresponding non-garbled dynamic circuit.

Algorithm 3. Garbling Dynamic Circuit with Route

Input: the public data pd as per Equation 8, the PRF secret keys sk and sk' , and a dynamic circuit C .

1. For each input cable i , uniformly sample (according to the corresponding assumption) the garbler's shares $\langle s \cdot \bar{x} \rangle_G$ for each wire $t \in [\ell]$, where ℓ is the number of wires in each cable.
2. For all other unique cables j in the circuit, sample a bit mask $r_{j,u}$ from $\text{PRF}_{\text{sk}'}$.
3. For each Route $\in C$, let n be the number of cables and ℓ be the size of each cable. Perform the following steps:
 - (a) Compute the garbler's target shares $\langle s \cdot h_i \rangle'_G$ and $\langle s \cdot h_j \rangle_G^{\text{out}}$ using PRF_{sk} for all input cables $i \in [n]$ and output cables $j \in [n]$ as per Step 3 of Route.Garble.
 - (b) Extract the garbler's shares $\langle s \cdot \bar{y}_{j,t} \rangle_G$ for all output cables $j \in [n]$ and all wires $t \in [\ell]$ as per Step 6 of Route.Garble.
4. For all $g \in C$, perform:

- (a) If g is a Boolean gate (not a Route), g is not yet garbled, and all the shares of g 's inputs are prepared, garble it using `HGate.Garble` as per Algorithm 1.
- (b) If g is a Route, g is not yet garbled, and all the shares of g 's inputs are prepared, perform the remaining steps of `Route.Garble`, excluding Steps 3 and 6.

The above is repeated until no such g is found.

Because the shares of all inputs of C and all outputs of all Routes are all prepared in Step 3, this step is well-defined and efficiently computable, and it garbles all gates.

5. Output the garbled circuit $\hat{C}$ consisting of `pd` and the messages output from each gate.

5.3 Instantiations Under Paillier Groups

In this section, we instantiate the routing protocol under Paillier groups. Given PRF secret keys $\text{sk}, \text{sk}' \in \{0, 1\}^\lambda$, public data `pd` is sampled as follows:

1. $\text{pp} = (\mathcal{N}, \zeta) \leftarrow \text{Pai.Gen}(1^\lambda, 1^\zeta)$.

2. Sample

$$g \leftarrow \text{Pai.Samp}(\text{pp}), \text{seed} \leftarrow \{0, 1\}^\lambda, s \leftarrow [\mathcal{N}], \mathbf{r} \leftarrow [\mathcal{N}]^{\lceil \log \mathcal{N} \rceil}, \mathbf{r}', \mathbf{r}'', \mathbf{t}', \mathbf{t}'' \leftarrow [\mathcal{N}]^\lambda, \\ \mathbf{a}_\beta \leftarrow [\mathcal{N}]^{\lceil \log \mathcal{N} \rceil} \text{ for } \beta \in [\ell], b_\alpha \leftarrow [\mathcal{N}], \mathbf{v}_\alpha \leftarrow [\mathcal{N}]^{\lceil \log \mathcal{N} \rceil} \text{ for } \alpha \in [\ell].$$

3. Given a PRF secret key $\text{sk}, \text{sk}' \in \{0, 1\}^\lambda$, set public data `pd` as

$$\begin{aligned} \text{pd} = & (\text{pp}, \text{seed}, g^{\mathbf{r}}, g^{\mathbf{r}s}, g^{\mathbf{r}s^2} \odot (1 + \mathcal{N})^{\text{Bits}(s)}, \\ & g^{\mathbf{r}'}, g^{\mathbf{r}'s} \odot (1 + \mathcal{N})^{\text{Bits}(\text{sk})}, g^{\mathbf{r}''s}, g^{\mathbf{r}''} \odot (1 + \mathcal{N})^{\text{Bits}(\text{sk})}, \\ & g^{\mathbf{t}'}, g^{\mathbf{t}'s} \odot (1 + \mathcal{N})^{\text{Bits}(\text{sk}')}, g^{\mathbf{t}''s}, g^{\mathbf{t}''} \odot (1 + \mathcal{N})^{\text{Bits}(\text{sk}')}, \\ & \{(g^{b_\alpha})^{\mathbf{a}_\beta s} \odot (1 + \mathcal{N})^{-\mathbf{u}_\beta[\alpha] \cdot \text{Bits}(s)}\}_{\alpha \in [\ell], \beta \in [\ell]}, \{g^{\mathbf{a}_\beta}\}_{\beta \in [\ell]}, \\ & \{g^{\mathbf{v}_\alpha}\}_{\alpha \in [\ell]}, \{g^{\mathbf{v}_\alpha s} \odot (1 + \mathcal{N})^{s \cdot b_\alpha}\}_{\alpha \in [\ell]}, \{g^{b_\alpha}\}_{\alpha \in [\ell]}), \end{aligned} \quad (9)$$

where $\mathbf{u}_\beta$ is a unit vector whose β -th element is 1.

Lemma 5.3 (Correctness of Routing Gate under Paillier Groups). *Let λ be a security parameter and `pd` be sampled as in Equation 9, where $\text{pp} = (\mathcal{N}, \zeta)$ is generated from $\text{Pai.Gen}(1^\lambda, 1^2)$ with secret exponent $s \in [\mathcal{N}]$. For each $i \in [n]$ and $t \in [\ell]$, let $\pi_i \in [n]$ be a routing, and $x_{i,t} \in \{0, 1\}$ be input wire values with $\hat{x}_{i,t} \in \{0, 1\}$ being masked wire values. There exists a negligible function $\text{negl}(\lambda)$ such that $(\langle s \cdot \bar{y}_{\pi_i, t} \rangle_G, \langle s \cdot \bar{y}_{\pi_i, t} \rangle_E)$ is a correct garbling of the output wire $y_{\pi_i, t} = x_{i,t}$, where*

$$\begin{aligned} \{\langle s \cdot \bar{y}_{\pi_i, t} \rangle_G\}_{i \in [n], t \in [\ell]} & \leftarrow \text{Route.Garble}(\text{pd}, \text{sk}, \{\langle s \cdot \hat{x}_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_G\}_{i \in [n]}), \\ \{\langle s \cdot \bar{y}_{\pi_i, t} \rangle_E\}_{i \in [n], t \in [\ell]} & \leftarrow \text{Route.Eval}(\text{pd}, \{\hat{x}_{i,t}\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot x_{i,t} \rangle_E\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_E\}_{i \in [n]} \cup \\ & \quad \{\text{rk}_i, \langle \text{rt}_i \rangle_G\}_{i \in [n]}, \{\langle r_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}), \end{aligned}$$

with failure probability less than or equal to $\text{negl}(\lambda)$.

Proof. The correctness follows directly from `Compress`, `ExtractP`, `EvalKey`, `EvalTag`, and `ExtEvalP`. $\square$

Lemma 5.4 (Security of Routing Gate under Paillier Groups). *Under the same setting as Lemma 5.3, there exists a simulator Route.Sim that statistically simulates G 's message in Route.Garble . There exists a negligible function $\text{negl}(\lambda)$ such that the statistical distance between the distributions*

$$\begin{aligned} & \text{Route.Garble}(\text{pd}, \text{sk}, \{\langle s \cdot \hat{x}_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_G\}_{i \in [n]}), \\ & \text{Route.Sim}(\text{pd}, \{x_{i,t}\}_{i \in [n], t \in [\ell]}, \{\pi_i\}_{i \in [n]}, \{\langle s \cdot \hat{x}_{i,t} \rangle_E\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_E\}_{i \in [n]}, \\ & \quad \{\bar{x}_{\pi_i, u}\}_{\pi_i \in [n], u \in [\ell]}, \{\langle s \cdot h_i \rangle'_E\}_{i \in [n]}, \{\langle s \cdot h_{\pi_i} \rangle_{\pi_i \in [n]}), \end{aligned}$$

is less than or equal to $\text{negl}(\lambda)$.

Proof. Security follows from the simulator Route.SimCable shown in Algorithm 4. $\square$

We split the simulator algorithm into two functions to account for the dynamic property of the circuits we are garbling. Recall that, based on the permutation of each routing gate, we may have some cables that are inactive, meaning they do not have control wires or input wires set for some cables. To handle this, we have the simulator perform a different protocol based on whether the cable is active or not to simulate the garbler's message. If all wires associated with a cable are set, the simulator runs Route.SimCable . If not, then they run Route.SimNoCable . Combining these conditions can be seen as the whole protocol for Route.Sim . Notice that the simulator needs extra inputs. These are required to call HGate.Sim , as well as calculate the rekeying material. More details on the simulator can be found in the proof of Theorem 5.6.

Algorithm 4. Routing Simulator

$\text{Route.SimCable}(\text{pd}, i \in [n], \{\hat{x}_{i,t}\}_{t \in [\ell]}, \pi_i, \{\langle s \cdot x_{i,t} \rangle_E\}_{t \in [\ell]}, \langle s \cdot \pi_i \rangle_E, \{\bar{z}_{\pi_i, t}\}_{t \in [\ell]}, \langle s \cdot h_i \rangle'_E, \langle s \cdot h_{\pi_i} \rangle_E^{\text{out}})$:

1. For $t \in [\ell]$, compute remasking material with $F_{g,i,t}^{\text{mask}}$ defined as in Equation 7

$$\begin{aligned} & (\bar{x}_{i,t}, \langle s \cdot \bar{x}_{i,t} \rangle_E, \neg \langle \bar{x}_{i,t} \rangle_G) \\ & \leftarrow \text{HGate.Sim}(\text{pd}_{\text{sk}'}, F_{g,i,t}^{\text{mask}}, \{\pi_i, \hat{x}_{i,t}\}, \{\langle s \cdot \pi_i \rangle_E, \langle s \cdot \hat{x}_{i,t} \rangle_E\}, \bar{z}_{\pi_i, t}). \end{aligned}$$

2. Compute compression and rekey material

$$\begin{aligned} \langle s \cdot h_i \rangle_E & \leftarrow \text{Compress}(\text{Hash.pd}, \{\langle s \cdot \bar{x}_{i,t} \rangle_E\}_{t \in [\ell]}), \\ \text{rk}_i & \leftarrow \langle s \cdot h_i \rangle'_E - \langle s \cdot h_i \rangle_E. \end{aligned}$$

3. Compute routing material

$$\text{rt}_i \leftarrow \langle s \cdot h_{\pi_i} \rangle_E^{\text{out}} - \langle s \cdot h_i \rangle'_E$$

4. Compute routing share with $F_{g,i}^{\text{key}}$ defined as in Equation 5

$$(\text{rt}_i, \langle s \cdot \text{rt}_i \rangle_E \neg \langle \text{rt}_i \rangle_G) \leftarrow \text{HGate.Sim}(\text{pd}_{\text{sk}}, F_{g,i}^{\text{key}}, \pi_i, \langle s \cdot \pi_i \rangle_E, \text{rt}_i).$$

5. Set route as

$$\langle s \cdot h_{\pi_i} \rangle_E^{\text{out}} \leftarrow \langle s \cdot h_i \rangle'_E + \text{rt}_i.$$

6. For each $t \in [\ell]$, compute output shares

$$\langle s \cdot \bar{y}_{\pi_i, t} \rangle_E \leftarrow \text{Extract}_E(\text{Hash.pd}, \langle s \cdot h_{\pi_i} \rangle_E^{\text{out}}, t, \{\bar{x}_{i, t}\}_{t \in [\ell]}).$$

7. For each $t \in [\ell]$, set

$$\bar{y}_{\pi_i, t} \leftarrow \bar{x}_{i, t}$$

8. Output $(\langle s \cdot \bar{y}_{\pi_i, t} \rangle_E, \bar{y}_{\pi_i, t} \ominus \{\langle \bar{x}_{i, t} \rangle_G\}_{t \in [\ell]}, \text{rk}_i, \langle \text{rt}_i \rangle_G)$.

Route.SimNoCable(pd, $i \in [n]$, $\{\langle s \cdot x_{i, t} \rangle_G\}_{t \in [\ell]}, \langle s \cdot \pi_i \rangle_G, \langle s \cdot h_i \rangle'_G$) :

1. For $t \in [\ell]$, compute remasking material with $F_{g, i, t}^{\text{mask}}$ defined as in Equation 7

$$(\langle s \cdot \bar{x}_{i, t} \rangle_G \ominus \langle \bar{x}_{i, t} \rangle_G) \leftarrow \text{HGate.Garble}(\text{pd}_{\text{sk}'}, F_{g, i, t}^{\text{mask}}, \{\langle s \cdot \pi_i \rangle_G, \langle s \cdot \hat{x}_{i, t} \rangle_G\})$$

2. Compute compression and rekey material

$$\langle s \cdot h_i \rangle_G \leftarrow \text{Compress}(\text{Hash.pd}, \{\langle s \cdot \bar{x}_{i, t} \rangle_G\}_{t \in [\ell]}),$$

$$\text{rk}_i \leftarrow \langle s \cdot h_i \rangle'_G - \langle s \cdot h_i \rangle_G.$$

3. Compute routing share with $F_{g, i}^{\text{key}}$ defined as in Equation 5

$$(\langle s \cdot \text{rt}_i \rangle_G, \ominus \langle \text{rt}_i \rangle_G) \leftarrow \text{HGate.Garble}(\text{pd}_{\text{sk}}, F_{g, i}^{\text{key}}, \langle s \cdot \pi_i \rangle_G).$$

4. Output $(\{\langle \bar{x}_{i, t} \rangle_G\}_{t \in [\ell]}, \text{rk}_i, \langle \text{rt}_i \rangle_G)$.

Remark 5.5. To complete the security proof of a dynamic garbling scheme, we must give the simulator the permutation π for each routing gate. This is a requirement since these permutations represent the physical accesses of the underlying ORAM. Since these accesses are public in the ORAM model, then we must reveal them here as well and prove that it reveals nothing about the true inputs. Specifically, these physical access patterns allow the simulator to replicate the dynamic circuit's execution order without knowing the inputs. This is important so that the simulator may construct a path consistent with the real garbled program.

Theorem 5.6 (Dynamic Garbling Scheme under Paillier Groups). *Under the CP-DDH assumption for Paillier groups, the garbling scheme for dynamic circuits, which consists of Boolean gates HGate and Route, is correct and secure.*

Proof. The proof follows the simulation strategy of Ishai, Li, and Lin [ILL25b] with modifications to account for the routing functionality. The correctness follows directly from Lemma 5.3 and Lemma 16 of Ishai, Li, and Lin [ILL25b]. We continue to argue for security by altering the garbling scheme as the simulator. The simulator should take as input the security parameter λ , the dynamic circuit being garbled C , and the permutation π for each routing gate.

- Sample Paillier public data $\widetilde{\text{pd}}$ as

$$\begin{aligned}
\text{pp} &= (\mathcal{N}, \zeta) \leftarrow \text{Pai.Gen}(1^\lambda, 1^2), g \leftarrow \text{Pai.Samp}(\text{pp}), \text{seed} \leftarrow \{0, 1\}^\lambda, \\
s &\leftarrow [\mathcal{N}], \mathbf{a}, \mathbf{b}, \mathbf{c} \leftarrow [\mathcal{N}^3]^{\log \mathcal{N}}, \mathbf{a}', \mathbf{b}', \mathbf{a}'', \mathbf{b}'', \mathbf{c}', \mathbf{d}', \mathbf{c}'', \mathbf{d}'' \leftarrow [\mathcal{N}]^\lambda, \\
\mathbf{e}'_\beta &\leftarrow [\mathcal{N}^3]^{\log \mathcal{N}} \text{ for } \beta \in [\ell], e'_\alpha \leftarrow [\mathcal{N}^3], \mathbf{e}_\alpha, \mathbf{f}_\alpha \leftarrow [\mathcal{N}]^{\lceil \log \mathcal{N} \rceil} \text{ for } \alpha \in [\ell], \\
\mathbf{f}'_{\alpha, \beta} &\leftarrow [\mathcal{N}^3]^{\log \mathcal{N}} \text{ for } \alpha, \beta \in [\ell],
\end{aligned} \tag{10}$$

$$\begin{aligned}
\widetilde{\text{pd}} &= (\text{pp}, \text{seed}, g^{\mathbf{a}}, g^{\mathbf{b}}, g^{\mathbf{c}}, g^{\mathbf{a}'}, g^{\mathbf{b}'}, g^{\mathbf{a}''}, g^{\mathbf{b}''}, g^{\mathbf{c}'}, g^{\mathbf{d}'}, g^{\mathbf{c}''}, g^{\mathbf{d}''}, \\
&\{g^{\mathbf{f}'_{\alpha, \beta}}\}_{\alpha \in [\ell], \beta \in [\ell]}, \{g^{\mathbf{e}'_\beta}\}_{\beta \in [\ell]}, \{g^{\mathbf{e}_\alpha}\}_{\alpha \in [\ell]}, \{g^{\mathbf{f}_\alpha}\}_{\alpha \in [\ell]}, \{e'_\alpha\}_{\alpha \in [\ell]}).
\end{aligned}$$

- Given the permutation π for each routing gate, parse the ordering of gates in the dynamic circuit to be simulated in execution order.
- Sample labels $\widetilde{l}_{i,t} \leftarrow [\mathcal{N}^3]$ and a masked bit $\widetilde{x}_{i,t} \leftarrow \{0, 1\}$ for each input wire t of each input cable i and sample a masked bit $x_{j,u} \leftarrow \{0, 1\}$ for every other wire u of every active cable j in C , where whether a wire is active or not is based on execution order.
- For each $\text{Route} \in C$, for input cable $i \in [n]$, randomly sample a target input hash label $\widetilde{l}'_i \leftarrow [\mathcal{N}^3]$. For each output cable $j \in [n]$, sample target output hash label $\widetilde{l}^{\text{out}}_j \leftarrow [\mathcal{N}^3]$.
- For each gate $g \in C$ in execution order, with labels $\widetilde{l}_{i,t}$ and masks $\widetilde{x}_{i,t}$ for cable $i \in \text{InCables}(g)$ and wire $t \in \text{InWires}(g)$ in each cable, run the respective simulator algorithm:
 - If g is a Boolean gate HGate , run the simulator according to Algorithm 1 with F_g^{hgate} representing a Boolean operation,

$$(\widetilde{l}_{j,u} \leftarrow \langle \widetilde{x}_{j,u}^g \rangle_G) \leftarrow \text{HGate.Sim}(\widetilde{\text{pd}}_{\text{sk}'}, F_g^{\text{hgate}}, \{\widetilde{x}_{i,t}\}_{t \in \text{InWires}(g)}, \{\widetilde{l}_{i,t}\}_{t \in \text{InWires}(g)}, \widetilde{x}_{j,u}),$$

where $\widetilde{l}_{j,u}$ is the output label for wire $u \in \text{OutWire}(g)$ and cable $j \in \text{OutCable}(g)$. $\widetilde{x}_{j,u}$ is the masked bit assigned to wire the output wire.

If the input wires are not active as per the execution order and the labels are prepared, instead run

$$(\widetilde{l}_{j,u} \leftarrow \langle \widetilde{x}_{j,u}^g \rangle_G) \leftarrow \text{HGate.Garble}(\widetilde{\text{pd}}_{\text{sk}'}, F_g^{\text{hgate}}, \{\widetilde{l}_{i,t}\}_{t \in \text{InWires}(g)}).$$

- If g is a routing gate Route , derive $\pi_i \in [n]$ for each input cable $i \in [n]$ for a routing gate of input size n according to the execution order. If the input wires and control wire for a cable $i \in [n]$ are set, run the simulator as in Algorithm 4 by Lemma 5.4,

$$\begin{aligned}
&(\{\widetilde{l}_{\pi_i, u}\}_{u \in [\ell]}, \{\widetilde{x}_{\pi_i, u}\}_{u \in [\ell]} \leftarrow \{\langle \widetilde{x}_{i,t}^g \rangle_G\}_{t \in [\ell]}, \widetilde{\text{rk}}_i^g, \langle \widetilde{\text{rt}}_i^g \rangle_G) \\
&\leftarrow \text{Route.SimCable}(\widetilde{\text{pd}}, i, \{\widetilde{x}_{i,t}\}_{t \in [\ell]}, \pi_i, \{\widetilde{l}_{i,t}\}_{t \in [\ell]}, \widetilde{l}_{\pi_i}, \{\widetilde{x}_{j,u}\}_{u \in [\ell]}, \widetilde{l}'_i, \widetilde{l}^{\text{out}}_{\pi_i}),
\end{aligned}$$

where ℓ is the number of wires per cable and $\widetilde{l}_{\pi_i, u}$ are the output labels for output wire $u \in [\ell]$ in each output cable $j \in [n]$, where π_i is the same set as j .

If the input wires or control wires for a particular cable are not active as per the execution order and the labels are prepared, instead run

$$(\{\langle \tilde{x}_{i,t}^g \rangle_G\}_{t \in [\ell]}, \tilde{rk}_i^g, \langle \tilde{rt}_i^g \rangle_G) \leftarrow \text{Route.SimNoCable}(\widetilde{\text{pd}}, i \in [n], \{\tilde{l}_{i,t}\}_{t \in [\ell]}, \tilde{l}_{\pi_i}, \tilde{l}'_i).$$

For any inactive output cables $j \in [n]$, compute the labels of their wires $u \in [\ell]$ as

$$(\tilde{l}_{j,u}) \leftarrow \text{Extract}_G(\text{Hash.pd}, \tilde{l}_j^{\text{out}}, u)$$

Go through each gate $g \in C$ until they have all been simulated.

- Let $\tilde{\mathbf{z}}$ be the masked bits assigned to output wires of the circuit C and compute the masks for output wires as $\tilde{\mathbf{o}} = \tilde{\mathbf{z}} + \mathbf{z}$.
- Output the simulated garbling $\tilde{C} = (\widetilde{\text{pd}}, \{\langle \tilde{x}_{j,u}^g \rangle_G\}_{\text{HGate} \in C}, \{\{\langle \tilde{x}_{i,t}^g \rangle_G\}_{i \in [n], t \in [\ell]}, \{\langle \tilde{rk}_i^g \rangle_G\}_{i \in [n]}, \{\langle \tilde{rt}_i^g \rangle_G\}_{i \in [n]}\}_{\text{Route} \in C}, \tilde{\mathbf{o}})$ and the input labels $\{\tilde{l}_{i,t}\}$.

Hyb₀ This is the real distribution of $\hat{C}$ and $\{l_{i,t}\}$.

Hyb₁ Modify the construction of the garbler's message for each gate by using HGate.Sim and Route.SimCable in execution order. For each gate $g \in C$, properly compute the gate logic to get each wire value $x_{j,u}$ and mask the wire value as $\bar{x}_{j,u} = x_{j,u} \oplus \text{PRF}_{\text{sk}'}(\text{ID}(g, \text{out}, j, u))$, where j is the cable that the wire u belongs to. Additionally, properly compute the target hashes for each routing gate as $\langle s \cdot h_i \rangle'_G \leftarrow \text{PRF}_{\text{sk}}(\text{ID}(g, \text{in}, i))$ and $\langle s \cdot h_j \rangle_G^{\text{out}} \leftarrow \text{PRF}_{\text{sk}}(\text{ID}(g, \text{out}, j))$ for each input/output cable $i, j \in [n]$ in each routing gate.

For $\text{HGate} \in C$, define $l_{i,t} = k_{i,t} + s \cdot \bar{x}_{i,t}$ to be the input labels. With labels $\tilde{l}_{i,t}$ and masked wire values $\tilde{x}_{i,t}$ for $t \in \text{InWires}(g)$, run the simulator for homomorphic gates:

$$(\tilde{l}_{j,u} \leftarrow \langle \tilde{x}_u^g \rangle_G) \leftarrow \text{HGate.Sim}(\text{pd}_{\text{sk}'}, C, \{\tilde{x}_{i,t}\}_{t \in \text{InWires}(g)}, \{\tilde{l}_{i,t}\}_{t \in \text{InWires}(g)}, \bar{x}_{j,u}),$$

where $\tilde{l}_{j,u}$ is the output label for wire $u \in \text{OutWire}(g)$ for cable j and $\bar{x}_{j,u}$ is the masked wire values assigned to the wire. If the input wires are not active as per the execution order and the labels are prepared, instead run

$$(\tilde{l}_{j,u} \leftarrow \langle \tilde{x}_{j,u}^g \rangle_G) \leftarrow \text{HGate.Garble}(\widetilde{\text{pd}}_{\text{sk}'}, F_g^{\text{hgate}}, \{\tilde{l}_{i,t}\}_{t \in \text{InWires}(g)}).$$

For $\text{Route} \in C$, define $l_{i,t} = k_{i,t} + s \cdot \bar{x}_{i,t}$ be the input labels for cable $i \in [n]$, where n is the input size of the routing gate. Derive π_i for each cable from the execution order. With labels $\tilde{l}_{i,t}$ and masked wire values $\tilde{x}_{i,t}$ for $t \in g_{\text{in}}$, run the simulator for routing gates:

$$\begin{aligned} & (\{\tilde{l}_{\pi_i,u}\}_{t \in [\ell]}, \{\tilde{x}_{\pi_i,u}\}_{t \in [\ell]} \leftarrow \{\langle \tilde{x}_{i,t}^g \rangle_G\}_{t \in [\ell]}, \tilde{rk}_i^g, \langle \tilde{rt}_i^g \rangle_G) \\ & \leftarrow \text{Route.SimCable}(\text{pd}, i, \{\tilde{x}_{i,t}\}_{t \in [\ell]}, \pi_i, \{\tilde{l}_{i,t}\}_{t \in [\ell]}, \tilde{l}_{\pi_i}, \{\bar{x}_{j,u}\}_{j \in [n], u \in [\ell]}, \langle s \cdot h_i \rangle'_G, \langle s \cdot h_{\pi_i} \rangle_G^{\text{out}}), \end{aligned}$$

where ℓ is the number of wires per cable and $\tilde{l}_{j,u}$ are the output labels for output wire $u \in [\ell]$ in each output cable $j \in [n]$. If the input wires or control wires for a particular cable are not active as per the execution order and the labels are prepared, instead run

$$(\{\langle \tilde{x}_{i,t}^g \rangle_G\}_{t \in [\ell]}, \tilde{rk}_i^g, \langle \tilde{rt}_i^g \rangle_G) \leftarrow \text{Route.SimNoCable}(\widetilde{\text{pd}}, i \in [n], \{\tilde{l}_{i,t}\}_{t \in [\ell]}, \tilde{l}_{\pi_i}, \tilde{l}_{\text{out}_i}).$$

For any inactive output cables $j \in [n]$, compute the labels of their wires $u \in [\ell]$ as

$$(\tilde{l}_{j,u}) \leftarrow \text{Extract}_G(\widetilde{\text{Hash.pd}}, \tilde{l}_j^{\text{out}}, u)$$

Go through each gate $g \in C$ until they have all been simulated. By the simulation security of **HGate** (Lemma 17 of Ishai, Li, and Lin [ILL25b]) and **Route** (Lemma 5.4), the resulting distribution is statistically indistinguishable from that of Hyb_0 . Thus, $\text{Hyb}_0 \approx \text{Hyb}_1$.

Hyb₂ Randomly sample the labels as $\tilde{l}_{i,t} \leftarrow [\mathcal{N}^3]$ instead of computing them as $l_{i,t} = k_{i,t} + s \cdot \bar{x}_{i,t}$. The resulting distributions are statistically close, so $\text{Hyb}_1 \approx \text{Hyb}_2$.

Hyb₃ Instead of computing the mask on the wires using the PRF, compute it as $\tilde{\mathbf{o}} = \tilde{\mathbf{z}} + \mathbf{z}$, where $\tilde{\mathbf{z}}$ are the masked wire values on the output wires. These ways of computing the masks are equivalent, so $\text{Hyb}_2 \equiv \text{Hyb}_3$.

Hyb₄ Replace the honestly generated public data for $\widetilde{\text{pd}}$ according to Equation 10. By Claim 1 of Ishai, Li, and Lin [ILL25b] and Lemma 8.2, the resulting distributions are computationally indistinguishable. Thus, $\text{Hyb}_3 \approx_c \text{Hyb}_4$.

Hyb₅ Replace the pseudorandom masked wire values $\bar{x}_{i,t}$ with randomly sampled $\tilde{x}_{i,t} \leftarrow \{0, 1\}$. Modify the construction of the garbler's message for routing gates such that, for each **Route** $\in C$ and each input cable $i \in [n]$, randomly sample a target input hash label $\tilde{l}_i' \leftarrow [\mathcal{N}^3]$ rather than from a PRF at the beginning of the protocol. Do similar for output cables $j \in [n]$, sampling target output hash labels $\tilde{l}_j^{\text{out}} \leftarrow [\mathcal{N}^3]$. Simulate the message as

$$\begin{aligned} & (\{\tilde{l}_{\pi_i, u}\}_{t \in [\ell]}, \{\tilde{x}_{\pi_i, u}\}_{t \in [\ell]} - \{\langle \tilde{x}_{i,t}^g \rangle_G\}_{t \in [\ell]}, \tilde{\mathbf{rk}}_i^g, \langle \tilde{\mathbf{rt}}_i^g \rangle_G) \\ & \leftarrow \text{Route.SimCable}(\text{pd}, i, \{\tilde{x}_{i,t}\}_{t \in [\ell]}, \pi_i, \{\tilde{l}_{i,t}\}_{t \in [\ell]}, \tilde{l}_{\pi_i}, \{\tilde{x}_{j,u}\}_{j \in [n], u \in [\ell]}, \tilde{l}_i', \tilde{l}_{\pi_i}^{\text{out}}). \end{aligned}$$

By the security of PRF, $\text{Hyb}_4 \approx_c \text{Hyb}_5$. Observe that this hybrid is distributed identically to the simulator output.

Hyb₅ is exactly the same as the simulated distribution of the garbled circuit required by Definition 4.1. By hybrid argument, we have that $\text{Hyb}_0 \approx_c \text{Hyb}_5$ and the dynamic garbling scheme is secure. $\square$

5.4 Instantiations Under Prime-Order Groups

In this section, we instantiate the routing protocol under prime-order groups. Given a PRF secret key $\text{sk}, \text{sk}' \in \{0, 1\}^\lambda$, public data pd is sampled as follows:

1. $\text{pp} = (G, p, g) \leftarrow \text{Pri.Gen}(1^\lambda)$.

2. Sample

$$\begin{aligned} \text{seed} & \leftarrow \{0, 1\}^\lambda, s \leftarrow \mathbb{Z}_p, \mathbf{r} \leftarrow \mathbb{Z}_p^{[\log p]}, \mathbf{r}', \mathbf{t}', \mathbf{R}, \mathbf{T} \leftarrow \mathbb{Z}_p^{\lambda \times [\log p]}, \\ \mathbf{a}_\beta & \leftarrow \mathbb{Z}_p^{[\log p]} \text{ for } \beta \in [\ell], b_\alpha \leftarrow \mathbb{Z}_p, \mathbf{v}_\alpha \leftarrow \mathbb{Z}_p^{[\log p]} \text{ for } \alpha \in [\ell]. \end{aligned}$$

3. Given a PRF secret key $\text{sk}, \text{sk}' \in \{0, 1\}^\lambda$, set public data pd as

$$\begin{aligned} \text{pd} = & (\text{pp}, \text{seed}, g^{\mathbf{r}}, g^{\mathbf{r}s}, g^{\mathbf{r}s^2 + \text{Bits}(s)}, \\ & g^{\mathbf{r}'s}, g^{\mathbf{r}'s^2 + \text{Bits}(\text{sk})}, g^{\mathbf{R}s}, g^{\mathbf{R}s^2 + \text{Bits}(\text{sk}) \cdot \text{Bits}(s)}, \\ & g^{\mathbf{t}'s}, g^{\mathbf{t}'s^2 + \text{Bits}(\text{sk}')}, g^{\mathbf{T}s}, g^{\mathbf{T}s^2 + \text{Bits}(\text{sk}') \cdot \text{Bits}(s)}, \\ & \{(g^{b_\alpha})^{\mathbf{a}_\beta s} / g^{\mathbf{u}_\beta[\alpha] \cdot \text{Bits}(s)}\}_{\alpha \in [\ell], \beta \in [\ell]}, \{g^{\mathbf{a}_\beta}\}_{\beta \in [\ell]}, \\ & \{g^{\mathbf{v}_\alpha}\}_{\alpha \in [\ell]}, \{g^{\mathbf{v}_\alpha s + s \cdot b_\alpha}\}_{\alpha \in [\ell]}, \{g^{b_\alpha}\}_{\alpha \in [\ell]}), \end{aligned} \quad (11)$$

where $\mathbf{u}_\beta$ is a unit vector whose β -th element is 1.

Lemma 5.7 (Correctness of Routing Gate under Prime-Order Groups). *Let λ be a security parameter and pd be sampled as in Equation 11, where $\text{pp} = (G, p, g)$ is generated from $\text{Pri.Gen}(1^\lambda)$ with secret exponent $s \in \{0, 1\}^{\lceil \log p \rceil}$. Let $\delta = 1/(\text{poly}(\lambda) \cdot |C|)$ be an error bound. For each $i \in [n]$ and $t \in [\ell]$, let $\pi_i \in [n]$ be a routing, and $x_{i,t} \in \{0, 1\}$ be input wire values with $\hat{x}_{i,t} \in \{0, 1\}$ being masked wire values. There exists a negligible function $\text{negl}(\lambda)$ such that $(\langle s \cdot \bar{y}_{\pi_i, t} \rangle_G, \langle s \cdot \bar{y}_{\pi_i, t} \rangle_E)$ is a correct garbling of the output wire $y_{\pi_i, t} = x_{i,t}$, where*

$$\begin{aligned} \{\langle s \cdot \bar{y}_{\pi_i, t} \rangle_G\}_{i \in [n], t \in [\ell]} & \leftarrow \text{Route.Garble}(\text{pd}, \text{sk}, \{\langle s \cdot \hat{x}_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_G\}_{i \in [n]}), \\ \{\langle s \cdot \bar{y}_{\pi_i, t} \rangle_E\}_{i \in [n], t \in [\ell]} & \leftarrow \text{Route.Eval}(\text{pd}, \{\hat{x}_{i,t}\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot x_{i,t} \rangle_E\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_E\}_{i \in [n]} \cup \\ & \quad \{\text{rk}_i, \langle \text{rt}_i \rangle_G\}_{i \in [n]}, \{\langle r_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}), \end{aligned}$$

with failure probability less than or equal to $\delta(\lambda) + \text{negl}(\lambda)$.

Proof. The correctness follows directly from Compress , Extract_P , EvalKey , EvalTag , and ExtEval_P . $\square$

Lemma 5.8 (Security of Routing Gate under Prime-Order Groups). *Under the same setting as Lemma 5.7, there exists a simulator Route.Sim that statistically simulates G 's message in Route.Garble . There exists a negligible function $\text{negl}(\lambda)$ such that the statistical distance between the distributions*

$$\begin{aligned} & \text{Route.Garble}(\text{pd}, \text{sk}, \{\langle s \cdot \hat{x}_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_G\}_{i \in [n]}), \\ & \text{Route.Sim}(\text{pd}, \{x_{i,t}\}_{i \in [n], t \in [\ell]}, \{\pi_i\}_{i \in [n]}, \{\langle s \cdot \hat{x}_{i,t} \rangle_E\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_E\}_{i \in [n]}, \\ & \quad \{\bar{x}_{\pi_i, u}\}_{\pi_i \in [n], u \in [\ell]}, \{\langle s \cdot h_i \rangle'_E\}_{i \in [n]}, \{\langle s \cdot h_{\pi_i} \rangle_{\pi_i \in [n]}), \end{aligned}$$

is less than or equal to $\text{negl}(\lambda)$.

Proof. Security follows from the simulator Route.SimCable shown in Algorithm 4. $\square$

Theorem 5.9 (Dynamic Garbling Scheme under Prime-Order Groups). *Under the CP-DDH assumption for prime-order groups, the garbling scheme compiled from HGate and Route is correct and secure.*

Proof. The proof follows the simulation strategy of Ishai, Li, and Lin [ILL25b] with modifications to account for the routing gate functionality. We continue to argue for security. Sample prime-order public data $\widetilde{\text{pd}}$ as

$$\begin{aligned} \text{pp} &= (G, p, g) \leftarrow \text{Pri.Gen}(1^\lambda), \text{seed} \leftarrow \{0, 1\}^\lambda, \\ \mathbf{a}, \mathbf{b}, \mathbf{c} &\leftarrow \mathbb{Z}_p^{\lceil \log p \rceil}, \mathbf{a}', \mathbf{b}', \mathbf{c}', \mathbf{d}' \leftarrow \mathbb{Z}_p^\lambda, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \leftarrow \mathbb{Z}_p^{\lambda \times \lceil \log p \rceil}, \\ \mathbf{e}'_\beta &\leftarrow \mathbb{Z}_p^{\lceil \log p \rceil} \text{ for } \beta \in [\ell], \mathbf{e}'_\alpha \leftarrow \mathbb{Z}_p, \mathbf{e}_\alpha, \mathbf{f}_\alpha \leftarrow \mathbb{Z}_p^{\lceil \log p \rceil} \text{ for } \alpha \in [\ell], \\ \mathbf{f}'_{\alpha, \beta} &\leftarrow \mathbb{Z}_p^{\lceil \log p \rceil} \text{ for } \alpha, \beta \in [\ell], \end{aligned} \quad (12)$$

$$\begin{aligned} \widetilde{\text{pd}} &= (\text{pp}, \text{seed}, g^{\mathbf{a}}, g^{\mathbf{b}}, g^{\mathbf{c}}, g^{\mathbf{a}'}, g^{\mathbf{b}'}, g^{\mathbf{A}}, g^{\mathbf{B}}, g^{\mathbf{C}}, g^{\mathbf{d}'}, g^{\mathbf{C}}, g^{\mathbf{D}}, \\ &\{g^{\mathbf{f}'_{\alpha, \beta}}\}_{\alpha \in [\ell], \beta \in [\ell]}, \{g^{\mathbf{e}'_\beta}\}_{\beta \in [\ell]}, \{g^{\mathbf{e}_\alpha}\}_{\alpha \in [\ell]}, \{g^{\mathbf{f}_\alpha}\}_{\alpha \in [\ell]}, \{e'_\alpha\}_{\alpha \in [\ell]}). \end{aligned}$$

The security proof uses a similar hybrid argument to that of Theorem 5.6 aside from the error probability with each call of HGate.Sim . This changes the statistical difference between Hyb_0 and Hyb_1 to $\delta(\lambda) + \text{negl}(\lambda)$ according to Lemma 5.7. Additionally, the simulated public data $\widetilde{\text{pd}}$ is sampled as in Equation 12, which requires the use of Claim 9 of Ishai, Li, and Lin [ILL25b]. $\square$

5.5 Instantiations Under Lattices

In this section, we instantiate the routing protocol under lattices. Public parameters are chosen as follows:

1. A polynomial ring $\mathcal{R}(\lambda) = \mathbb{Z}[X]/(X^{n(\lambda)} + 1)$, moduli $p(\lambda), q(\lambda)$, and error distributions $\mathcal{D}_{\text{err}}, \mathcal{D}_s$, where $n \leq \text{poly}(\lambda)$ is a power-of-two, $p \geq \lambda^{\omega(1)}$, and $q = p \cdot \Delta$ for some $\Delta \geq p \cdot \lambda^{\omega(1)}$, and $\mathcal{D}_{\text{err}}, \mathcal{D}_s$ have coefficients bounded by $\text{poly}(\lambda)$. Let pp be the following public parameters.

$$\text{pp} = (\mathcal{R}, p, q, \mathcal{D}_{\text{err}}, \mathcal{D}_s) \quad (13)$$

2. Sample

$$\begin{aligned} \text{seed} &\leftarrow \{0, 1\}^\lambda, s, r_1, r_2, t_1, t_2 \leftarrow \mathcal{D}_s, a \leftarrow \mathcal{R}_q, \mathbf{a}' \leftarrow \mathcal{R}_q^\lambda, \\ e_1, e_2 &\leftarrow \mathcal{D}_{\text{err}}, \mathbf{e}', \mathbf{e}'_1, \mathbf{e}'_2, \mathbf{e}'_3, \mathbf{e}'_4, \mathbf{e}''_1, \mathbf{e}''_2, \mathbf{e}''_3, \mathbf{e}''_4 \leftarrow \mathcal{D}_{\text{err}}^\lambda, \mathbf{b} := s\mathbf{a}' + \mathbf{e}', \\ \mathbf{r}_\beta &\leftarrow \mathcal{R}_q^{\lceil \log q / \log B \rceil} \text{ for } \beta \in [\ell], b_\alpha \leftarrow \mathcal{R}_q \text{ for } \alpha \in [\ell], \delta \leftarrow \mathcal{R}_q, \gamma \leftarrow \mathcal{R}_p^{\lceil \log q / \log B \rceil}, \\ \mathbf{e}_\beta &\leftarrow \mathcal{D}_{\text{err}}^{\lceil \log q / \log B \rceil} \text{ for } \beta \in [\ell], e_{\alpha, \beta} \leftarrow \mathcal{D}_{\text{err}} \text{ for } \alpha, \beta \in [\ell]. \end{aligned}$$

3. Given a PRF secret key $\text{sk}, \text{sk}' \in \{0, 1\}^\lambda$, set public data pd as

$$\begin{aligned} \text{pd} &= (\text{pp}, \text{seed}, a, sa + e_1, s^2a + e_2 - s\Delta, \\ &r_1\mathbf{a}' + \mathbf{e}'_1 + \text{Bits}(\text{sk})\Delta, r_1\mathbf{b} + \mathbf{e}''_1, \\ &r_2\mathbf{a}' + \mathbf{e}'_2, r_2\mathbf{b} + \mathbf{e}''_2 + \text{Bits}(\text{sk})\Delta, \\ &t_1\mathbf{a}' + \mathbf{e}'_3 + \text{Bits}(\text{sk}')\Delta, t_1\mathbf{b} + \mathbf{e}''_3, \\ &t_2\mathbf{a}' + \mathbf{e}'_3, t_2\mathbf{b} + \mathbf{e}''_3 + \text{Bits}(\text{sk}')\Delta, \\ &\{(s \cdot \mathbf{r}_\beta + \mathbf{e}_\beta)^\top \cdot b_\alpha + e_{\alpha, \beta} - s \cdot \mathbf{u}_\beta[\alpha] \cdot \Delta\}_{\alpha \in [\ell], \beta \in [\ell]}, \\ &\{\mathbf{r}_\beta\}_{\beta \in [\ell]}, \{b_\alpha\}_{\alpha \in [\ell]}, \delta, \gamma), \end{aligned} \quad (14)$$

where $\mathbf{u}_\beta$ is a unit vector whose β -th element is 1.

Lemma 5.10 (Correctness of Routing Gate under Lattices). *Let λ be a security parameter and pd be sampled as in Equation (14), where pp is generated from as in Equation (13) with secret $s \in \mathcal{D}_s$. For each $i \in [n]$ and $t \in [\ell]$, let $\pi_i \in [n]$ be a routing, and $x_{i,t} \in \{0, 1\}$ be input wire values with $\hat{x}_{i,t} \in \{0, 1\}$ being masked wire values. There exists a negligible function $\text{negl}(\lambda)$ such that $(\langle s \cdot \bar{y}_{\pi_i,t} \rangle_G, \langle s \cdot \bar{y}_{\pi_i,t} \rangle_E)$ is a correct garbling of the output wire $y_{\pi_i,t} = x_{i,t}$, where*

$$\begin{aligned} \{\langle s \cdot \bar{y}_{\pi_i,t} \rangle_G\}_{i \in [n], t \in [\ell]} &\leftarrow \text{Route.Garble}(\text{pd}, \text{sk}, \{\langle s \cdot \hat{x}_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_G\}_{i \in [n]}), \\ \{\langle s \cdot \bar{y}_{\pi_i,t} \rangle_E\}_{i \in [n], t \in [\ell]} &\leftarrow \text{Route.Eval}(\text{pd}, \{\hat{x}_{i,t}\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot x_{i,t} \rangle_E\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_E\}_{i \in [n]} \cup \\ &\quad \{\text{rk}_i, \langle \text{rt}_i \rangle_G\}_{i \in [n]}, \{\langle r_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}), \end{aligned}$$

with failure probability less than or equal to $\text{negl}(\lambda)$.

Proof. The correctness follows directly from Compress , Extract_P , EvalKey , EvalTag , and ExtEval_P . $\square$

Lemma 5.11 (Security of Routing Gate under Lattices). *Under the same setting as Lemma 5.10, there exists a simulator Route.Sim that statistically simulates G 's message in Route.Garble . There exists a negligible function $\text{negl}(\lambda)$ such that the statistical distance between the distributions*

$$\begin{aligned} &\text{Route.Garble}(\text{pd}, \text{sk}, \{\langle s \cdot \hat{x}_{i,t} \rangle_G\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_G\}_{i \in [n]}), \\ &\text{Route.Sim}(\text{pd}, \{x_{i,t}\}_{i \in [n], t \in [\ell]}, \{\pi_i\}_{i \in [n]}, \{\langle s \cdot \hat{x}_{i,t} \rangle_E\}_{i \in [n], t \in [\ell]}, \{\langle s \cdot \pi_i \rangle_E\}_{i \in [n]}, \\ &\quad \{\bar{x}_{\pi_i,u}\}_{\pi_i \in [n], u \in [\ell]}, \{\langle s \cdot h_i \rangle'_E\}_{i \in [n]}, \{\langle s \cdot h_{\pi_i} \rangle_{\pi_i \in [n]}), \end{aligned}$$

is less than or equal to $\text{negl}(\lambda)$.

Proof. Security follows from the simulator Route.SimCable shown in Algorithm 4. $\square$

Theorem 5.12 (Dynamic Garbling Scheme under Lattices). *Under the CP-RLWE assumption, the garbling scheme compiled from HGate and Route is correct and secure.*

Proof. The proof follows the simulation strategy of Ishai, Li, and Lin [ILL25b] with modifications to account for the routing gate functionality. The correctness follow directly from Lemma 5.10 and Lemma 16 of Ishai, Li, and Lin [ILL25b]. We continue to argue for security.

Sample lattice public parameters $\text{pp} = (\mathcal{R}, p, q, \mathcal{D}_{\text{err}}, \mathcal{D}_s)$ as in Equation (13) and sample public data $\widetilde{\text{pd}}$ as

$$\begin{aligned} \text{seed} &\leftarrow \{0, 1\}^\lambda, a, b, c \leftarrow \mathcal{R}_q, \mathbf{b}', \mathbf{c}', \mathbf{b}'', \mathbf{c}'', \mathbf{d}', \mathbf{f}', \mathbf{d}'', \mathbf{f}'' \leftarrow \mathcal{R}_q^\lambda, \\ \mathbf{g}_\beta &\leftarrow \mathcal{R}_q^{\lceil \log q / \log B \rceil} \text{ for } \beta \in [\ell], g_\alpha \leftarrow \mathcal{R}_q \text{ for } \alpha \in [\ell], \\ \mathbf{g}'_{\alpha,\beta} &\leftarrow \mathcal{R}_q^{\lceil \log q / \log B \rceil} \text{ for } \alpha, \beta \in [\ell], h \leftarrow \mathcal{R}_q, \mathbf{h}' \leftarrow \mathcal{R}_p^{\lceil \log q / \log B \rceil}, \\ \widetilde{\text{pd}} &= (\text{pp}, \text{seed}, a, b, c, \mathbf{b}', \mathbf{c}', \mathbf{b}'', \mathbf{c}'', \mathbf{d}', \mathbf{f}', \mathbf{d}'', \mathbf{f}'', \{\mathbf{g}'_{\alpha,\beta}\}_{\alpha \in [\ell], \beta \in [\ell]}, \{\mathbf{g}_\beta\}_{\beta \in [\ell]}, \{g_\alpha\}_{\alpha \in [\ell]}, h, \mathbf{h}'). \end{aligned} \tag{15}$$

The security proof uses a similar hybrid argument to that of Theorem 5.6 the simulated public data $\widetilde{\text{pd}}$ being sampled as in Equation (15), which requires the use of Claim 5 of Ishai, Li, and Lin [ILL25b] and Lemma 8.4.

□

6 Garbling Bucket using Route

In Section 5, we introduced the `Route` and `ReverseRoute` gadgets used throughout the garbled RAM.⁵ In this section, we describe the other gadget: `Bucket`. The bucket is designed to be a building block of Circuit ORAM [WCS15, Algorithms 1–4 and 6]. That is, a bucket was designed to implement a tree node of Circuit ORAM. Below, we summarize the operations required by Circuit ORAM.

`Bucket`[w, α, n, t]: A bucket is parametrized by (w, α, n, t) , and it stores at most n data entries, where each entry is a tuple of address, value, and position, and each entry’s address is unique. Moreover, w is the bit-length of each entry, $\alpha < w/2$ is the bit-length of an address or a position, and t is the number of times the following operations can be performed. A bucket supports `ReadAndRemove`, `IsEmpty`, `GetDeepest`, `HasEmptySlot`, and `Insert` operations.

- `ReadAndRemove(addr)`: It looks for the address `addr` in the bucket. That is, if the tuple $(\text{addr}, \text{val}, \text{pos})$ for some (val, pos) is in the internal storage of this bucket, remove and output the tuple; else, output ‘not found’.
- `IsEmpty()`: It checks whether the bucket is empty. That is, if there is no tuple in the internal storage of this bucket, output 1; else, output 0.
- `GetDeepest(pos*, phase ∈ {prep, evict})`: It looks for the tuple in the bucket that can be moved into the deepest location on the root-to-leaf path denoted by `pos*`. That is, it searches for the tuple $(\text{addr}, \text{val}, \text{pos})$ in the internal storage of this bucket whose position label `pos` shares the longest common prefix with `pos*`. If `phase = prep`, then output the length of the common prefix; else if `phase = evict`, remove the tuple $(\text{addr}, \text{val}, \text{pos})$ from the internal storage and output it. If there is no such tuple found, output ‘not found’. If two or more tuples have position labels with the same longest common prefix with `pos*`, we break tie arbitrarily, e.g., choose the tuple with the smaller address.
- `HasEmptySlot()`: It checks whether the bucket has an empty slot. That is, if there exists an empty slot in the internal storage of this bucket, output 1; else, output 0.
- `Insert(addr, val, pos)`: It places the input tuple into the bucket. That is, if the bucket has an empty slot, insert the input tuple into this slot; else, output ‘full’.

To garble buckets containing only a constant number n of real entries, a straightforward implementation with standard boolean gates suffices. However, for buckets with more entries, directly scanning all n entries incurs undesirable overhead that is linear in the bucket size, $n \cdot w$. To avoid this cost, we implement the buckets more efficiently using our `ReverseRoute` and a perfect ORAM [CNS18].

⁵Only `ReverseRoute` is used in this section, but we use `Route` in the section title for comprehensiveness.

6.1 Background: Perfect ORAM

Perfect ORAM is an ORAM scheme that achieves perfect simulation for any memory access. Specifically, for any $n \in \mathbb{N}$, any access pattern of a perfect ORAM on any n -element array is always identically distributed [DMN11]. In contrast, for tree-based ORAMs [SvDS⁺13, WCS15], the distributions between different accesses may be distinguished with a probability negligible in n , which may be undesirable theoretically when n is not large enough due to concentration bounds. The best known perfect ORAMs [CNS18, CSLN21] achieve $\text{poly log } n$ overhead in the number of accesses. We will employ a perfect ORAM to instantiate a **Bucket** of $n = \text{poly log } N$ elements, so that each operation on **Bucket** takes only $\text{poly log } n = \text{poly log log } N$ cost. Here, N denotes the memory size of the overall RAM in the final garbled RAM construction. Accordingly, each **addr** or **pos** carries $\alpha = O(\log N)$ bits. Our garbled **Bucket** uses our garbled **ReverseRoute** to reduce communication cost. To show the use of **ReverseRoute**, we summarize the perfect ORAM scheme of Chan, Nayak, and Shi [CNS18] below.

The perfect ORAM scheme is based on the hierarchical paradigm [GO96]. In the paradigm, the data structure consists of $L = \Theta(\log n)$ levels, where level i stores 2^i data elements. Within each level i , the 2^i elements are shuffled uniformly at random, and the shuffling is mixed with 2^i dummy elements. To fetch an element with a given $\text{idx} \in [n]$, the scheme “searches” from the smallest level to the largest, where the search will be specified below. If found, the element is moved to the smallest level, and we touch a dummy element in each of the remaining larger levels. For every level $i \in [L]$, for every 2^i steps, each level smaller than 2^i are shuffled to level i , so that we can always move an element to the smallest level. It remains to specify an efficient and secure search for the element in each level.

The idea of Chan, Nayak, and Shi [CNS18] is to store the position of each $\text{idx} \in [n]$ recursively using the same perfect ORAM scheme. Namely, for each idx , we need to remember its level $i \in [L]$ and its position $j \in [2^i]$ in the level i . By grouping the positions of every *two* idx ’s into *one* data element, the problem size is reduced from n to $n/2$. Repeating this process yields $O(\log n)$ hierarchical ORAMs. There are other details: when shuffling smaller levels into a larger level, the positions of the shuffled elements must be efficiently updated in the recursive perfect ORAM. The details are less important because the updates always access pre-determined locations and can be implemented by Boolean circuits. For instance, the shuffling is implemented by the Waksman permutation [Wak68], where the garbler prepares the permutation, and updating the positions is implemented by sorting networks [Bat68, AKS83], which is a circuit and fulfills obliviousness. A detailed description of perfect ORAM is provided by Appendix A.1

To implement the above perfect ORAM using a dynamic circuit, following the idea of Zebra [GGS26, Section 8], we mechanically verify that almost all procedures can be implemented by (static) boolean circuits. The only exception is that, within each level, we want to read an element at a location, but the location is determined by the input of the perfect ORAM dynamically. The trick is that the perfect ORAM scheme guarantees that each element in the shuffled array, whether data or dummy, is read at *most once* (before the array is re-shuffled). Such an array is called One-Time Memory (OTM) in previous works [PLS23, GGS26]. Next, we describe the functionality of OTM, show that OTM can be efficiently implemented by **ReverseRoute**, and thus give a dynamic circuit implementation of perfect ORAM and **Bucket**.

6.2 Garbling a Perfect ORAM

Our circuit gadget `ReverseRoute` directly yields an OTM. More formally, an OTM consists of two procedures:

- `OTM.Buildn(A)`: Given an array A of n elements, return $\text{otm} = A$.
- `OTM.Readn(otm, i)`: Given an otm and index $i \in [n]$, return the i -th entry of the data array stored in otm .

Implementing an OTM using `ReverseRoute` is simple. `OTM.Buildn(A)` is just renaming the array, so there is no need to implement anything. To implement `OTM.Readn`, we instantiate a `ReverseRoute` gadget with n cables and each cable consists of ℓ wires, where ℓ is the bit-length of an OTM element. Namely, for all $i \in [n]$, the array element $\text{otm}[i]$ is connected to the i th input cable of `ReverseRoute`. With that, given $j \in [n]$, the j th invocation of `OTM.Readn(otm, i)` simply performs `ReverseRoute` with the control π_j set to $\pi_j = i$ and connects the j th output cable $\mathbf{x}_j$ to the OTM output. This completes the dynamic circuit of OTM.

Implementing the perfect ORAM of Chan, Nayak, and Shi [CNS18] as a dynamic circuit is tedious but mechanical. As summarized in Section 6.1, the algorithm consists of three partial steps: (a) compute which location to read in the current level, (b) read a location in the OTM of the current level, and (c) rebuild a level, which merges and shuffles smaller levels into a data structure and updates the corresponding position maps in a batched fashion. Notice that both (a) and (c) partial steps can be efficiently implemented by standard (non-dynamic) boolean circuits; the circuit size is asymptotically the same as the number of bits computed by the ORAM. Also, step (b), reading an OTM, can be efficiently implemented by `ReverseRoute`. The above can be verified mechanically with the detailed procedures in Appendix A.1. With that, we claim the following.

Lemma 6.1. *For any $n, w, T \in \mathbb{N}$ such that $T \geq n$, let $w_{\text{meta}} = O(\log n)$ be bit-length of metadata entries of the perfect ORAM. There is a dynamic circuit implementation of a perfect ORAM with space n supporting T accesses to w -bit data entries; moreover, the dynamic circuit consists of*

- $O(\lceil \frac{T}{2^i} \rceil)$ number of `ReverseRoute` $[n_i, w + w_{\text{meta}}]$ gadgets, and $O(\lceil \frac{T}{2^i} \rceil \cdot \log n)$ number of `ReverseRoute` $[n_i, w_{\text{meta}}]$ gadgets, where $n_i = O(2^i)$ for all $i \in \{1, \dots, \log n\}$.
- $O(T \cdot ((w + w_{\text{meta}}) \log^2 n + \log^4 n))$ number of standard boolean gates.

Proof. Recall that `ReverseRoute` $[n', \ell]$ denotes a reverse-route gadget that routes n' cables, each carrying a w -bit element.

In the perfect ORAM, each level in the main ORAM and the recursive ORAMs is implemented by an OTM using `ReverseRoute`. For each $i \in \{1, \dots, \log n\}$, an OTM with $n_i = O(2^i)$ entries is rebuilt every 2^i accesses in each ORAM. The main ORAM incurs a `ReverseRoute` $[n_i, w + w_{\text{meta}}]$. For each recursive ORAM, the bit-length of data is $O(\log n)$, so each rebuild incurs a `ReverseRoute` $[n_i, w_{\text{meta}}]$. There are $O(\log n)$ recursive ORAMs. In T total accesses, $O(\lceil \frac{T}{2^i} \rceil)$ number of `ReverseRoute` $[n_i, w + w_{\text{meta}}]$ and $O(\lceil \frac{T}{2^i} \rceil \cdot \log n)$ number of `ReverseRoute` $[n_i, w_{\text{meta}}]$ are incurred for every $i \in \{1, \dots, \log n\}$.

A coordinated rebuilding procedure at level i consists of rebuilding one level of size $O(2^i)$ in the actual data ORAM together with the corresponding levels in all recursive ORAMs. In the actual

data ORAM, each entry has bit-length $w + w_{\text{meta}}$, and in each recursive ORAM, each entry has bit-length $w_{\text{meta}} = O(\log n)$.

Each rebuild of $O(2^i)$ entries is implemented using a constant number of oblivious permutations, oblivious sorting operations, and linear scans over an array of 2^i entries in the actual data ORAM, and $O(\log n)$ such operations across all recursive ORAMs. For an array of 2^i entries of bit-length b , each such operation requires $O(b \cdot i \cdot 2^i)$ number of boolean gates [Wak68, Bat68, AKS83].

Therefore, one coordinated rebuild at level i incurs $O((w + w_{\text{meta}}) \cdot i \cdot 2^i)$ boolean gates for the main ORAM, and $O(i \cdot 2^i \cdot \log^2 n)$ boolean gates for all $\log n$ instances of recursive ORAMs. A level of size 2^i is rebuilt once every 2^i accesses. Thus, over T accesses, the total contribution of level i is $O(\lceil \frac{T}{2^i} \rceil \cdot (w + w_{\text{meta}} + \log^2 n) \cdot i \cdot 2^i)$. Summing over all $i \in [\log n]$ yields $O(T \cdot ((w + w_{\text{meta}}) \cdot \log^2 n + \log^4 n))$. $\square$

Combining the communication cost of garbling `ReverseRoute` and boolean gates, we obtain the following corollary.

Corollary 6.2. *For any n, w , there exists a garbling scheme for the dynamic circuit implementation of a perfect ORAM with space n , such that achieves $O(T \cdot ((w + \lambda) \cdot \log^2 n + \log^4 n))$ bits of communication for $T \geq n$ accesses to w -bit data entries. Meanwhile, the computation time is $O(nw + w^2 \log n + n \log^2 n) \cdot T \cdot \text{poly}(\lambda)$.*

Proof. Recall that by Theorem 5.1 and Remark 5.2, each `ReverseRoute` $[n', \ell]$ takes $O(n'(\lambda + \ell))$ bits communication and $n'\ell(n' + \ell) \cdot \text{poly}(\lambda)$ computation time for all n' cables. Each standard boolean gate implemented by `HGate` only incurs one bit communication and $O(\text{poly}(\lambda))$ computation time as Theorem 4.11. The total bandwidth incurred by perfect ORAM over T accesses is

$$\begin{aligned} & O\left(\sum_{i=1}^{\log n} \left\lceil \frac{T}{2^i} \right\rceil \cdot 2^i(\lambda + w + w_{\text{meta}}) + \sum_{i=1}^{\log n} \left\lceil \frac{T}{2^i} \right\rceil \log n \cdot 2^i(\lambda + w_{\text{meta}}) + T((w + w_{\text{meta}}) \log^2 n + \log^4 n)\right) \\ &= O(T \log n \cdot (\lambda + w + w_{\text{meta}}) + T \log^2 n \cdot (\lambda + \log n) + T((w + w_{\text{meta}}) \log^2 n + \log^4 n)) \\ &= O(T \log^2 n \cdot (\lambda + w + w_{\text{meta}}) + T \log^4 n) \\ &= O(T \log^2 n \cdot (\lambda + w) + T \log^4 n). \end{aligned}$$

Let $\ell_1 = w + w_{\text{meta}}$, and let $\ell_2 = w_{\text{meta}}$. The computation time is

$$\begin{aligned} & O\left(\sum_{i=1}^{\log n} \left\lceil \frac{T}{2^i} \right\rceil \cdot 2^i(w + w_{\text{meta}})(2^i + w + w_{\text{meta}}) + \sum_{i=1}^{\log n} \left\lceil \frac{T}{2^i} \right\rceil \log n \cdot 2^i w_{\text{meta}}(2^i + w_{\text{meta}})\right) \cdot \text{poly}(\lambda) \\ & \quad + O(T((w + w_{\text{meta}}) \log^2 n + \log^4 n) \text{poly}(\lambda)) \\ &= O\left(T \cdot \ell_1(n + \log n \cdot \ell_1) + T \log n \cdot \ell_2 \cdot (n + \log n \cdot \ell_2) + T \ell_1 \log^2 n + T \log^4 n\right) \cdot \text{poly}(\lambda) \\ &= O((w + w_{\text{meta}})^2 \log n + (w + w_{\text{meta}})n + w_{\text{meta}}^2 \log^2 n + w_{\text{meta}}n \log n + \log^4 n) \cdot T \cdot \text{poly}(\lambda) \\ &= O(nw + n \log^2 n + w^2 \log n) \cdot T \cdot \text{poly}(\lambda) \end{aligned}$$

The last equality follows by $w_{\text{meta}} = O(\log n)$ and then $\text{poly}(\log n) = O(n)$. $\square$

Finally, to implement a garbling scheme of a $\text{Bucket}[w, \alpha, n, t]$, we simply instantiate a perfect ORAM of $\Theta(n)$ entries, each entry is $\Theta(w)$ bits. Through the perfect ORAM, we build two textbook binary search trees, one sorts the addresses addr , and another sorts the positions pos , and both points to the list of all tuples. The Bucket operations ReadAndRemove , GetDeepest , and Insert are instantiated using the binary search trees. Notice that for any α -bit pos^* , $\text{GetDeepest}(\text{pos}^*)$ can be achieved by $O(\log \alpha)$ membership queries in the position binary search tree; Thus, GetDeepest takes $O(\log n \cdot \log \alpha)$ accesses to perfect ORAM. The operations IsEmpty and HasEmptySlot can be realized by keeping extra counters of $O(\log n)$ bits. Hence, garbling the Bucket translates to garbling the perfect ORAM with $T = O(t \cdot \text{poly}(\log n, \log \alpha))$ accesses.

Corollary 6.3. *For any w, α, n, t such that $t \geq n$ and $w > 2\alpha$, there exists a garbling scheme of $\text{Bucket}[w, \alpha, n, t]$ such that it takes $O(t \cdot (w + \lambda) \cdot \text{poly}(\log n, \log \alpha))$ bits of communication and $O(t \cdot w^2 n \cdot \text{poly}(\log n, \log \alpha, \lambda))$ computation time.*

7 Garbled RAM from Route and Bucket

In this section, we show that a garbled RAM can be realized by garbling a dynamic circuit, which uses gadgets, Route , ReverseRoute , and Bucket , introduced in Section 5 and Section 6, respectively. First, we recall their functionalities and parameters.

- $\text{Route}[n, \ell]$: Route receives n input cables $\{\mathbf{x}_i\}_{i \in [n]}$, each contains ℓ wires, and n control cables $\{\pi_i\}_{i \in [n]}$, each consists of $\log n$ wires. The outputs of a Route are n cables, $\{\mathbf{y}_i\}_{i \in [n]}$. The Route can eagerly evaluate an input cable to its destination output according to the control cable. That is, for any $i \in [n]$, once the input cable $\mathbf{x}_i$ and its control cable π_i are determined, then the π_i -th output cable $\mathbf{y}_{\pi_i}$ is set to the value on cable $\mathbf{x}_i$.
- $\text{ReverseRoute}[n, \ell]$: ReverseRoute receives n input cables $\{\mathbf{y}_i\}_{i \in [n]}$, each contains ℓ wires, and n input control cables $\{\pi_i\}_{i \in [n]}$, each consists of $\log n$ wires. The outputs of a ReverseRoute are n cables $\{\mathbf{x}_i\}_{i \in [n]}$, each consists of ℓ wires. The ReverseRoute can eagerly evaluate an input cable to its destination output according to the control cable. That is, for any $i \in [n]$, once the input cable $\mathbf{y}_{\pi_i}$ and its control cable π_i are determined, then the i -th output cable $\mathbf{x}_i$ is set to the value on cable $\mathbf{y}_{\pi_i}$.
- $\text{Bucket}[w, \alpha, n, t]$: Bucket is parametrized by the bit-length w of each entry, the bit-length $\alpha < w/2$ of each address or position label in an entry, the maximal number n of real entries stored in a bucket, and the number t of operations supported. Bucket supports ReadAndRemove , IsEmpty , GetDeepest , HasEmptySlot , and Insert operations as described in Section 6.

7.1 Simulate RAM Programs using Dynamic Circuits

Prior works [HKO23, GTRS25, GGS26] showed how to compile a RAM program with word size W into a Boolean dynamic circuit efficiently. Let T be the running time of the RAM program, and N be the RAM space, where $T \geq N$. In each time step, the RAM program reads from one memory word, computes on a constant number of words, and writes to one memory word; memory reads and writes are jointly called memory accesses. In this subsection, we explain the construction of a dynamic circuit that simulates Circuit ORAM [WCS15].

In Circuit ORAM, memory is organized as a binary tree with N/Z leaves for some parameter Z to

be chosen later. Each memory word is randomly assigned to one of the root-to-leaf paths, called the position of the word. For each access, Circuit ORAM performs one fetch and one eviction. Given a logical address `addr` and the position `pos` of the word to be accessed, the fetch stage looks for `addr` in each node on the root-to-leaf path `pos`, outputs the found word, and then moves the word to the root node with the updated position and potentially the newly written value. The eviction scans two random root-to-leaf paths; each scan walks between the root and the leaf three times, and when moving from one node to another, at most one word and $O(\log N)$ bits are carried. In the ORAM literature, a tree node is often interchangeably called a bucket due to its storage functionality. However, in this paper, we use nodes to realize and connect parts of the tree, and we use buckets to realize only the storage within each node.

In the actual Circuit ORAM, each logical access is simulated by traversing one path from the root to a uniformly random leaf in the tree (i.e., fetching) and then traversing two root-to-leaf paths deterministically [WCS15, Algorithm 6, EvictDeterministic]. For simplicity, we focus on the uniformly random paths next and hypothetically suppose that there is no other traversal; we will address the deterministic traversals later in Remark 7.1.

Simulating tree nodes of Circuit ORAM. Our simulation of Circuit ORAM using dynamic circuits is essentially the same as Zebra [GGS26, Section 6.1], and we reproduce it below for completeness. The running time T of the RAM program is segmented into $\lceil T/N \rceil$ periods, each period consisting of $T' = N$ memory accesses. For each period, we build a new Circuit ORAM tree by sorting and shuffling the previous period's data, which takes a boolean circuit of size $O((W + \log N) \cdot N \log N)$ by sorting networks [AKS83, Goo14]. We suppose that T'/Z is a power of 2 w.l.o.g.; if not, we choose a larger period T' so that T'/Z is a power of 2 and $T' < 2N$. Henceforth, we suppose that $T' = N$ is a power of 2 multiplied by Z and consider exactly N memory accesses.

For each tree node of Circuit ORAM, a sequence of dynamic-circuit gadgets is instantiated, where each gadget is called a micro node, μNode , for short. Let $D = \log \frac{N}{Z}$ be the tree depth. Each tree node is identified by the node depth d , and the i th node in the same depth of the tree, where $i \in [2^d]$, $d = 0$ is the root, and $d = D$ is a leaf. For each node (d, i) , there is a sequence of $t_d := N/(2^d \cdot Z)$ number of μNode gadgets; we denote the sequence by $\mu\text{Node}_{d,i} = (\mu\text{Node}_{d,i,0}, \mu\text{Node}_{d,i,1}, \dots, \mu\text{Node}_{d,i,t_d-1})$, where $\mu\text{Node}_{d,i,j}$ for some $j \in [t_d]$, the j th gadget in the sequence. For some constant fraction $\delta > 0$, each μNode receives a random number of requests from two parent μNode 's of the same sequence, where the number is in the range $Z \pm \delta Z$ with overwhelming probability. Moreover, the μNode sends $Z/2 \pm \delta Z$ requests to each of its left and right child μNode with overwhelming probability. The number of output *cables* per micro node is $(1+3\delta) \cdot Z$, which is more than the number $(1+2\delta) \cdot Z$ of output *requests*, which is in turn more than the number $(1+\delta) \cdot Z$ of its input requests and thus cables. That is, for each child, we intentionally overlap the two parents' $2\delta \cdot Z$ outputs. Both the δ fraction and the overlap ensure that the measure concentration holds. Within the same sequence (d, i) , every two μNode gadgets are also connected so that their requests can be sent to the same child nodes seamlessly. See Figures 3 and 4.

To see why the input is $(1+\delta) \cdot Z$ whereas the output is $(1+3\delta) \cdot Z$ cables with overlap, consider a sequence of $2^d \cdot Z$ random tree paths. With overwhelming probability, each node at depth d is visited $Z \pm \delta Z$ times. The $(1+\delta) \cdot Z$ input cables ensure that the micro node handles all the input visits. Next, each such node visits each of its children $Z/2 \pm \frac{1}{2}\delta \cdot Z$ times. The $(1+3\delta) \cdot Z$ output cables and the $(1+\delta) \cdot Z$ input cables of the child implies that there are $2\delta \cdot Z$ overlapped cables

on the children's input, where the overlapping starts from $(\frac{1}{2} - \frac{1}{2}\delta) \cdot Z$ to $(\frac{1}{2} + \frac{3}{2}\delta) \cdot Z$. With that, whenever the micro node visits any of its child for $(\frac{1}{2} \pm \frac{1}{2}\delta) \cdot Z$ times, the last visit lies in the overlap interval, from $(\frac{1}{2} - \frac{1}{2}\delta) \cdot Z$ to $(\frac{1}{2} + \frac{3}{2}\delta) \cdot Z$. Consequently, the parent's succeeding micro node can continue the communication with the same child. The details are elaborated below.

Connections between micro nodes. Formally, let $\delta > 0$ be a constant fraction, and let $Z = \text{poly log } N$, to be chosen later. Since Circuit ORAM always moves a W -bit word and $O(\log N)$ -bit metadata between tree nodes, let $w = W + O(\log N)$ be the number of wires in a cable. Each micro node $\mu\text{Node}_{d,i,j}$ is connected to other micro nodes by w -bit cables as follows:

- Parent micro nodes. If $d = 0$, $\mu\text{Node}_{0,0,j}$ is a root, and it takes Z input cables from the CPU circuit and Z output cables to the CPU circuit. Otherwise, $d > 0$, $\mu\text{Node}_{d,i,j}$ is non-root, and it takes $(\frac{1}{2} + \frac{3}{2}\delta) \cdot Z$ input cables from and output cables to each of the two parents $\mu\text{Node}_{d-1,\lfloor i/2 \rfloor, 2j}$ and $\mu\text{Node}_{d-1,\lfloor i/2 \rfloor, 2j+1}$; moreover, the last $2\delta Z$ cables of the first parent $\mu\text{Node}_{d-1,\lfloor i/2 \rfloor, 2j}$ and the first $2\delta Z$ cables of the second parent $\mu\text{Node}_{d-1,\lfloor i/2 \rfloor, 2j+1}$ overlap, so that the first $(\frac{1}{2} - \frac{1}{2}\delta) \cdot Z$ -th input cables exclusively from the first parent, the next $r + (\frac{1}{2} - \frac{1}{2}\delta) \cdot Z$ -th input cables for all $r \in [1, 2\delta Z]$ lies in the overlap region and may be supplied by either parent, and starting from the $1 + (\frac{1}{2} + \frac{3}{2}\delta) \cdot Z$ -th input cable, all the remaining input cables come exclusively from the second parent.
- Children micro nodes. If $d = D$, $\mu\text{Node}_{d,i,j}$ is a leaf, and there is no children. Otherwise, $d < D$, $\mu\text{Node}_{d,i,j}$ is connected to two children, left child $\mu\text{Node}_{d+1,2i,\lfloor j/2 \rfloor}$ and right child $\mu\text{Node}_{d+1,2i+1,\lfloor j/2 \rfloor}$, with $(1 + 3\delta) \cdot Z$ output and input cables, where the first $(\frac{1}{2} + \frac{3}{2}\delta) \cdot Z$ cables go to the left child and last $(\frac{1}{2} + \frac{3}{2}\delta) \cdot Z$ cables go to the right child.

See Figure 3 for an example of tree depth $D = 3$ and the number of leaves $N/Z = 8$. Also, Figure 4 illustrates the connection between a pair of parents and their children micro nodes, where two parents overlap their output cables to the same child.

- In the same (d, i) sequence, each micro node $\mu\text{Node}_{d,i,j}$ is also connected to either its predecessor $\mu\text{Node}_{d,i,j-1}$ if j is odd or its successor $\mu\text{Node}_{d,i,j+1}$ if j is even. It is used to ensure they send requests sequentially to their children's input cables without any gaps, as explained next.

Functionality of a micro node. Recall that the logical address and the position are both at most $\alpha = O(\log N)$ bits. Also, each data element of **Bucket** is $W + 2\alpha \leq w$ bits. For each $d \in [0, D]$, each $i \in [2^d]$, and each $j \in [t_d]$, the micro node $\mu\text{Node}_{d,i,j}$ realizes the following functionality.

1. Internal status: Each micro node $\mu\text{Node}_{d,i,j}$ maintains the following internal state.
 - Two counters $\text{ctr}_l, \text{ctr}_r \in [(\frac{1}{2} + \frac{3}{2}\delta)Z]$ that keep which cable to communicate with the left/right child.
 - A **Bucket** $[w, \alpha, n, t]$, where n is the capacity of **Bucket** and t is the number of supported operations. During the N accesses to the ORAM, in the fetch phase, every μNode is visited $Z \pm \delta Z$ times, which performs $Z \pm \delta Z$ invocations of **ReadAndRemove** on the bucket. Therefore, we set $t = (1 + \delta)Z$. Finally, **Bucket** capacity n is set depending on the node depth d as follows.

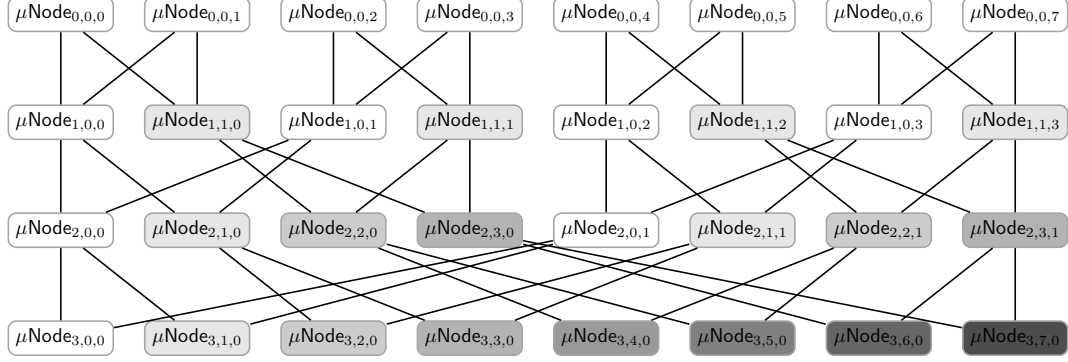


Figure 3: An illustration of the connection between μNode micro nodes, where the tree depth $D = 3$, the number of tree leaves $N/Z = 2^D = 8$. Each edge represents $(\frac{1}{2} + \frac{3}{2}\delta) \cdot Z$ input and output cables, but some cables overlap; see Figure 4 next. Each root micro node $\mu\text{Node}_{0,0,*}$ takes and responds to exactly Z memory accesses from the CPU, which is not shown in this diagram.

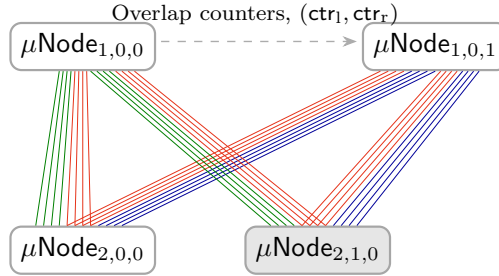


Figure 4: An example of the cables connecting two parents and two children, where each solid edge, no matter colored green, red, or blue, represents a cable that communicates bidirectionally. In the example, each micro node takes $(1 + \delta) \cdot Z = 13$ cables from parents, where $Z = 10$ and $\delta = 0.2$, but each micro node has $(1 + 3\delta) \cdot Z = 16$ cables to children, so that there are $2\delta \cdot Z = 4$ overlapping cables, colored in red. The overlap counter between the parents, shown as the dashed arrow, ensures that each overlapping cable is used exactly once.

- If $d = 0$ or $d = D$, this is the root or a leaf, initialize its internal $\text{Bucket}[w, \alpha, n, t]$ with $n = Z$.
 - If $0 < i < D$, this is a non-root internal node, initialize its internal $\text{Bucket}[w, \alpha, n, t]$ with $n = 5$.
 - Additionally, the micro node is equipped with a $\text{Route}[(1 + 3\delta) \cdot Z, w]$ gadget and a $\text{ReverseRoute}[(1 + 3\delta) \cdot Z, w]$ gadget.
2. Initiated by predecessor:
- (a) Depending on the number j in the sequence, initialize Bucket contents:
 - If $j = 0$, this is the first $\mu\text{Node}_{d,i,0}$ in the sequence (d, i) in this period. Then, if $d = D$, this is a leaf, initialize the Bucket using the sorted contents from the previous period. Else, $d < D$ and this is not a leaf, initialize Bucket to an empty bucket.
 - Otherwise, $j > 0$, the $\mu\text{Node}_{d,i,j}$ copies the Bucket content from its predecessor $\mu\text{Node}_{d,i,j-1}$.
 - (b) Depending on the number j in the sequence, initialize the counters $\text{ctr}_l, \text{ctr}_r$:
 - If j is even, set $\text{ctr}_l = \text{ctr}_r = 0$.
 - If j is odd, set $\text{ctr}_l = \text{ctr}'_l$ and set $\text{ctr}_r = \text{ctr}'_r$, where $(\text{ctr}'_l, \text{ctr}'_r)$ are received from its predecessor $\mu\text{Node}_{d,i,j-1}$.
3. Handle requests from parents: After initialization, this gadget receives at most $(1 + \delta) \cdot Z$ requests from its parents. Let $\text{ctr} \in [(1 + \delta)Z]$ be the number of requests so far, where $\text{ctr} = \text{ctr}_l + \text{ctr}_r$. Each request is part of either a fetch or a eviction of Circuit ORAM, so that the request is associated with a tree path pos such that the tree node (d, i) is on the path pos (when it is part of fetch, the request is additionally associated with an address addr , which is also handled by the per-node computation below). In any case, the request is at most w bits, which consists of a W -bit word and $O(\log N)$ -bit metadata.
- Each request is handled as follows, which passes data down to and receives data from a child, and then replies to the parent.
- (a) Fulfill the per-node computation of Circuit ORAM by performing Bucket operations. The detailed procedure is identical to that of Circuit ORAM and skipped. We focus on passing data to the child below.
 - (b) Perform the following if this node is not a leaf (i.e., $d < D$). After the per-node computation figured out the request to the child, using Route , send the w -bit request to the left child if $\text{pos}[d] = 0$, or to the right child if $\text{pos}[d] = 1$. That is,
 - If $\text{pos}[d] = 0$, invoke Route by setting the ctr -th input cable with the w -bit data and setting the ctr -th control cable to ctr_l .
 - If $\text{pos}[d] = 1$, invoke Route by setting the ctr -th cable with the w -bit data and setting the ctr -th control cable to $\text{ctr}_r + (\frac{1}{2} + \frac{3}{2}\delta) \cdot Z$.

After that, receive the w -bit response from the requested child using `ReverseRoute` in a reversed way; whenever the response is ready, this micro node $\mu\text{Node}_{d,i,j}$ reads the data from the output of `ReverseRoute`.

- If $\text{pos}[d] = 0$, invoke `ReverseRoute` by setting the ctr -th control cable to ctr_l and letting the received data be the ctr -th output of `ReverseRoute`.
- If $\text{pos}[d] = 1$, invoke `ReverseRoute` by setting the ctr -th control cable to $\text{ctr}_r + (\frac{1}{2} + \frac{3}{2}\delta) \cdot Z$ and letting the received data be the ctr -th output of `ReverseRoute`.

If $\text{pos}[d] = 0$, increment ctr_l ; otherwise, $\text{pos}[d] = 1$, increment ctr_r .

- (c) Fulfill the per-node computation of Circuit ORAM, i.e., aggregating the fetched element, and respond to the requesting parent.

4. Initiate the successor: If $j \leq n_d - 1$, send the `Bucket` content to $\mu\text{Node}_{d,i,j+1}$, and send the counters $\text{ctr}'_l, \text{ctr}'_r$ as follows.

- If j is even, let $\text{ctr}'_l = \text{ctr}_l - (\frac{1}{2} - \frac{1}{2}\delta) \cdot Z$ and $\text{ctr}'_r = \text{ctr}_r - (\frac{1}{2} - \frac{1}{2}\delta) \cdot Z$. If $\text{ctr}'_l \notin [2\delta Z]$ or $\text{ctr}'_r \notin [2\delta Z]$, this $\mu\text{Node}_{d,i,j}$ is underflowed or overflowed, output `Abort`.
- If j is odd, there is nothing to set and send.

The above concludes the dynamic circuits that simulate a tree of Circuit ORAM when the circuitry is properly integrated with sorting networks for every period of N accesses. It remains to recursively instantiate the position map of Circuit ORAM, as described below.

To retrieve the position labels for a given data block, additional recursive position map trees are used to store these labels associated with each block. These position map trees are implemented using `Route`, `ReverseRoute`, and `Bucket` gadgets following the above micro nodes. To reduce the overhead of the position map, we follow the “big-block” trick from Zebra [GGS26]. Concretely, instead of storing a constant number of position labels in each position map entry, we pack $\beta' = O(N^\epsilon)$ position labels into a single entry, where ϵ is a small positive constant. As a result, the address space of the position map shrinks by a factor of β' per recursion level, and hence the recursion depth becomes $\log_{\beta'} N = O(1)$. Meanwhile, this packing increases the bit-length of each position map entry to $\beta = \beta' \log N$ bits.

Remark 7.1 (Fullfledged Circuit ORAM with eviction). As mentioned, the above simulates a hypothetical Circuit ORAM in which each logical access traverses exactly one uniformly random tree path, but that performs only the fetching step. The actual Circuit ORAM additionally performs eviction that traverses two root-to-leaf paths deterministically [WCS15, Algorithm 6, EvictDeterministic]. The ingenious eviction [WCS15, Algorithms 2–4 and 6] consists of several detailed steps. At a high level, depending on the times of logical accesses performed so far, the eviction deterministically chooses two paths. Then, each path is repeatedly traversed from the root to the leaf for three times; moreover, for each node in a traversal, the traversal reads and writes $O(\log N) + W$ bits on the node, and then it carries $O(\log N) + W$ bits to the next node, where N is the number of words and W is the word size of the ORAM tree. The eviction is finished after the above traversals. Since the paths are deterministic, the eviction can be simulated with circuits, illustrated below.

The traversals on each path are determined by the number of logical accesses so far, and thus we simply wire the root-to-leaf path as a standard boolean circuit without going through the `Route` nor

ReverseRoute gadgets; i.e., in a given eviction, each parent node is directly connected to its child on the evicting path. Within an eviction, $O(\log N) + W$ bits per node on the paths are maintained between the three traversals, and they are simulated by standard boolean gates. The only steps that may use dynamic gates are to perform operations on the Bucket of each node: by syntactical observation, each traversal performs at most two Bucket operations [WCS15, Algorithms 2–4]; see also Section 6 for all potential operations. We wire the eviction circuits to the Bucket gadgets instantiated in Step 1, and in that step, we instantiate each Bucket with $12Z$ additional operations. That is, the number t of operations is set to $(13 + \delta)Z$. This concludes the modified circuitry.

The eviction increases at most a constant factor in any efficiency metric because each traversal is not more costly than that of the random path. Hence, its cost is omitted in the analysis below.

We summarize the required numbers of Route, ReverseRoute, and Bucket, along with their parameters, for implementing a Boolean RAM program as a dynamic circuit.

Lemma 7.2 (Boolean RAM Program as Dynamic Circuit). *A Boolean RAM of space N on W -bit words with running time T can be implemented as a Boolean dynamic circuit using the following gadgets, where $w_{\text{meta}} = O(\log N)$ is the bit-length of the metadata, $\beta = \beta' \log N$ is the bit-length of each position map entry such that $\log_{\beta'} N = O(1)$, and $Z' = O(Z)$:*

- **Route.**
 - For the main ORAM tree: $O(\frac{T}{Z} \log N)$ number of Route $[Z', W + w_{\text{meta}}]$ and ReverseRoute $[Z', W + w_{\text{meta}}]$.
 - For the position maps: $O(\frac{T}{Z} \log N)$ number of Route $[Z', \beta]$ and ReverseRoute $[Z', \beta]$.
- **Bucket.**
 - For the main ORAM tree: $O(\frac{T}{Z} \log N)$ number of Bucket $[W + w_{\text{meta}}, w_{\text{meta}}, 5, Z']$ and $O(\frac{T}{Z})$ number of Bucket $[W + w_{\text{meta}}, w_{\text{meta}}, Z, Z']$.
 - For the position maps: $O(\frac{T}{Z} \log N)$ number of Bucket $[\beta, w_{\text{meta}}, 5, Z']$ and $O(\frac{T}{Z})$ number of Bucket $[\beta, w_{\text{meta}}, Z, Z']$.
- **Other gates.**
 - $O(TW)$ number of standard boolean gates for CPU computation.
 - $O(T \cdot (W + w_{\text{meta}} + \beta) \cdot \log N)$ number of standard boolean gates for sorting network after every N memory accesses.

Proof. We count the required gadgets by first analyzing the number of μNode gadgets in the dynamic-circuit simulation of Circuit ORAM, and then translating each μNode into the corresponding Route, ReverseRoute, and Bucket gadgets.

For the main ORAM tree, each level contains $O(N/Z)$ micro nodes, and there are $O(\log N)$ levels. Hence, the main ORAM tree requires $O(\frac{N}{Z} \log N)$ micro nodes for every N memory accesses, resulting in $O(\frac{T}{Z} \log N)$ micro nodes over T memory accesses. The same counting applies to each tree in the recursive position maps, so the position maps also contribute $O(\frac{T}{Z} \log N)$ micro nodes in total when the shrinking factor β' satisfies $\log_{\beta'} N = O(1)$.

Each micro node $\mu\text{Node}_{d,i,j}$ is implemented using one `Route`, one `ReverseRoute`, and one `Bucket`, and their parameters are calculated below.

We begin with the first parameter of `Route` and `ReverseRoute`. The expected number of requests received by each μNode is Z . By Chernoff's bound, the probability that this number falls outside the interval $[Z - \delta Z, Z + \delta Z]$ is at most $2 \exp(-\frac{\delta^2 Z}{3})$. Similarly, for any fixed parent-child pair, the expected number of requests sent from the parent to the child is $Z/2$. Again by Chernoff bound, the probability that this number falls outside $[Z/2 - \delta Z/2, Z/2 + \delta Z/2]$ is no more than $2 \exp(-\frac{\delta^2 Z}{6})$. Therefore, for each μNode , the cable allocation neither underflows nor overflows except with probability at most $4 \exp(-\frac{\delta^2 Z}{6})$. Taking a union bound over all μNode , the cable allocation for the entire ORAM succeeds except with probability $O\left(\frac{T}{Z} \cdot \log N \cdot \exp(-\frac{\delta^2 Z}{6})\right)$. To achieve a probability negligible in N or λ , we may set $Z = \omega(\log N)$ and $\delta = \Omega(1)$, e.g., $Z = \log^2 N$ and $\delta = 0.1$; the failure probability is statistical and can differ from the computational security. Hence, setting $(1 + 3\delta) \cdot Z = O(Z)$ suffices. The second parameter of `Route` and `ReverseRoute` is the payload length, which is $W + w_{\text{meta}}$ in the main ORAM tree and $\beta = \beta' \cdot \log N$ in the recursive position maps.

The payload length is $W + w_{\text{meta}}$ in the main ORAM tree and β in the recursive position maps, which explains the first parameter of `Bucket`. The second parameter of `Bucket` is $w_{\text{meta}} = O(\log N)$, since each bucket entry additionally stores an address or position label of length $O(\log N)$.

It remains to justify the capacity and operation parameters of `Bucket`. For internal nodes, it suffices to store only a constant number of real entries in each bucket; following the standard Circuit ORAM analysis [WCS15], capacity 5 is sufficient. For root and leaf nodes, `Bucket` needs to hold as many as $Z = \text{poly}(\log N)$ entries [WCS15]. This explains the third parameter of `Bucket`. For the fourth parameter, namely the number of supported operations, the number of requests received by each μNode is at most $(1 + \delta) \cdot Z$ during fetch by the Chernoff's bound as above. The number of times for a node lies in the chosen path during eviction is $2Z$ deterministically. For each path during eviction, 3 traverses of the path are made and at most 2 bucket operations are made according to [WCS15]. By standard concentration bounds, allocating $Z' = O(Z)$ operations suffices with high probability. $\square$

Remark 7.3. In previous work Zebra [GGS26], garbling each `Route` $[n, \ell]$ or `ReverseRoute` $[n, \ell]$ takes bandwidth $O(n\ell + n^2)$. As a result, when implementing boolean RAM program using the dynamic circuit gadgets, one must choose a relatively small value of n to obtain good efficiency. In contrast, the bandwidth of our `Route` is linear in the number of cables in `Route`. Therefore, our implementation can use fewer, but larger, `Route` gadgets without sacrificing efficiency.

7.2 Cost of Garbled RAMs

To implement the `Bucket` gadgets, we adopt different strategies depending on the bucket capacity. For buckets that store only a constant number n of real entries, a direct scan of the entire bucket suffices, yielding a `Bucket` $[w, \alpha, n, t]$ implementation with $O(wnt)$ number of standard boolean gates. For larger buckets, we instead apply the technique from Section 6, namely, perfect ORAM [CNS18], to realize the bucket functionality more efficiently. Combining Theorem 4.11, Theorem 5.1, Remark 5.2, and Corollary 6.3, we summarize the bandwidth and computation cost of gadgets.

Lemma 7.4 (Bandwidth and Computation Cost of Gadgets). *In the garbling RAM, each gadget incurs the following amortized bandwidth and computation costs:*

- *From garbling the ORAM:*
 - $\text{Route}[n, \ell]$: $O(n(\lambda + \ell))$ bandwidth and $O(n(n\ell + \ell^2) \cdot \text{poly}(\lambda))$ computation.
 - $\text{Bucket}[w, \alpha, 5, t]$: $O(wt)$ bandwidth and $O(wt \cdot \text{poly}(\lambda))$ computation.
 - $\text{Bucket}[w, \alpha, n, t]$: $O(t \cdot (w + \lambda) \cdot \text{poly}(\log n, \log \alpha))$ bandwidth and $O(t \cdot nw^2 \cdot \text{poly}(\log n, \log \alpha, \lambda))$ computation.
- *From CPU computation:*
 - *Standard boolean gates*: $O(1)$ bandwidth and $O(\text{poly}(\lambda))$ computation.

Combining Lemma 7.2 and Lemma 7.4 yields the final efficiency result.

Theorem 7.5 (Garbled Boolean RAMs). *Assume the existence of pseudorandom functions (PRFs) in NC1, and assume one of the following assumptions: (1) Circular Power-DDH in Paillier groups (Definition 4.5), or (2) a variant of Circular Power-DDH in prime-order groups (Definition 4.6), or (3) Circular Power-RLWE (Definition 4.7). There exists a garbled RAM scheme for RAM programs with running time T , memory size N words, and word size W such that the communication is*

$$O(\lambda + W + \beta + \log N) \cdot T \cdot \log N + \text{poly}(\log Z, \log \log N)$$

bits with a $O((W + \log N)^2 + \beta^2) \cdot \text{poly}(\lambda)$ -bit one-time communication for many instances of gadgets, and the computation complexity is

$$O(Z \cdot (W + \beta) \cdot \log N + (Z + W) \cdot \log^2 N + (\log N + Z)(W^2 + \beta^2)) \cdot T \cdot \text{poly}(\log Z, \log \log N, \lambda).$$

Here, λ is the bit-length of either a group or a ring element, depending on the assumption; β, Z are parameters chosen during the underlying implementation of the Circuit ORAM described previously, with $\beta = \beta' \log N$, $\log_{\beta'} N = O(1)$ and $Z = \omega(\log N)$.

Proof. We first analyze the communication complexity.

- **Route.**

$$\begin{aligned} & O\left(\frac{T}{Z} \log N \cdot Z \cdot (\lambda + W + w_{\text{meta}}) + \frac{T}{Z} \log N \cdot Z \cdot (\lambda + \beta)\right) \\ &= O(T \cdot (\lambda + W + \beta + w_{\text{meta}}) \cdot \log N). \end{aligned}$$

- **Bucket.**

$$\begin{aligned} & O\left(\frac{T}{Z} \log N \cdot Z(W + \beta + w_{\text{meta}}) + \frac{T}{Z} \cdot Z(\lambda + W + \beta + w_{\text{meta}}) \text{poly}(\log Z, \log w_{\text{meta}})\right) \\ &= O(T \cdot (W + \beta + w_{\text{meta}}) \cdot \log N + T \cdot (\lambda + W + \beta + w_{\text{meta}}) \cdot \text{poly}(\log Z, \log w_{\text{meta}})). \end{aligned}$$

- **Other gates.** $O(T \cdot (W + w_{\text{meta}} + \beta) \cdot \log N)$.

Summing the above terms yields total communication using $w_{\text{meta}} = O(\log N)$

$$O(\lambda + W + \beta + \log N) \cdot T \cdot (\log N + \text{poly}(\log Z, \log \log N)).$$

We next analyze the computation complexity.

- **Route.**

$$\begin{aligned} & O\left(\frac{T}{Z} \log N \cdot Z \cdot (Z \cdot (W + w_{\text{meta}}) + (W + w_{\text{meta}})^2) \cdot \text{poly}(\lambda) + \frac{T}{Z} \log N \cdot Z \cdot (Z\beta + \beta^2) \cdot \text{poly}(\lambda)\right) \\ &= O(Z \cdot (W + \beta + w_{\text{meta}}) + (W + w_{\text{meta}})^2 + \beta^2) \cdot T \cdot \log N \cdot \text{poly}(\lambda) \end{aligned}$$

- **Bucket.**

$$\begin{aligned} & O\left(\frac{T}{Z} \log N \cdot Z \cdot (W + \beta + w_{\text{meta}}) \cdot \text{poly}(\lambda) + \frac{T}{Z} \cdot (Z^2 \cdot ((W + w_{\text{meta}})^2 + \beta^2)) \cdot \text{poly}(\log Z, \log w_{\text{meta}}, \lambda)\right) \\ &= O((W + \beta + w_{\text{meta}}) \cdot \log N + Z \cdot ((W + w_{\text{meta}})^2 + \beta^2)) \cdot T \cdot \text{poly}(\log Z, \log w_{\text{meta}}, \lambda) \end{aligned}$$

- **Other gates.** $O(T \cdot (W + w_{\text{meta}} + \beta) \cdot \log N \cdot \text{poly}(\lambda))$.

Therefore, using $w_{\text{meta}} = O(\log N)$ the total computation is

$$O(Z \cdot (W + \beta) \cdot \log N + (Z + W) \cdot \log^2 N + (\log N + Z)(W^2 + \beta^2)) \cdot T \cdot \text{poly}(\log Z, \log \log N, \lambda).$$

□

When $N = \text{poly}(\lambda)$ and $\beta = \lambda^c$ for any constant $c \in (0, 1]$, we have $\log_{\beta'} N = \frac{\log N}{\log(\beta'/\log N)} = \frac{\log N}{\Omega(\log \lambda) - \log \log N} = \frac{\log N}{\Omega(\log N) - \log \log N} = O(1)$. Also, we have $\log N = O(\lambda)$, which implies that $\lambda + \beta + \log N = O(\lambda)$. Hence, we have the following corollary.

Corollary 7.6. *Suppose that $N = \text{poly}(\lambda)$, and assume the hardness assumptions stated in Theorem 7.5. There exists a garbled RAM scheme with bandwidth $O(T \cdot \log N \cdot (\lambda + W)) + W^2 \cdot \text{poly}(\lambda)$ bits and $O(TW^2) \cdot \log N \cdot \text{poly}(\lambda)$ computation complexity.*

In the computation complexity, the $\log N$ factor can be absorbed into the $\text{poly}(\cdot)$ term. We keep it explicit to emphasize that the $\log N$ factor does not arise from PRF.

8 Optimizations to Fixed Factor Hashing

In this section, we present our modification to the hash compression and extraction seen in the previous work on succinct low-depth garbling algorithms [LLL26]. There are only two modifications, the first is that our scheme relies on circular security, and the second is that our hash value is shorter, explained below.

The previous work applied fixed factor hashing on each level when garbling a leveled circuit, and they use a freshly new secret in the authenticated shares for each level. Since each secret encrypts the previous secret, their scheme is based on plain (and non-circular) assumptions. However, in our dynamic circuits of garbled RAMs, our circuitry consists of too many levels, so that sending

per-level public data takes too much bandwidth. Therefore, we use a non-leveled variant but require the use of circularly-secure assumptions.

To achieve a lower bandwidth, we also modified the hashing, so that a vector of authenticated shares is compressed into *one* authenticated share $\langle s \cdot h \rangle_P$. In contrast, in the previous work, the compressed authenticated share is a *vector of $O(\lambda)$ shares*, where $O(\lambda)$ is the number of bits of a group/ring element. Thus, this modification reduces our bandwidth by a λ factor. We achieve the lower bandwidth by carefully moving a step from the **Extract** to the **Compress** procedure, i.e., Eq. (16).

We present these modification in Algorithm 5 and 6. Similarly to Li, Lin, and Lu [LLL26], we show only the modification for prime-order groups and lattices, where the construction for Paillier groups can be inferred from the prime-order construction. For simplicity, we borrow the vector notation from the previous work, where the i -th component of vector $\mathbf{x}$ is denoted by $\mathbf{x}[i]$.

8.1 Hashing under DDH

In the fixed factor hashing under DDH seen in Li, Lin, and Lu [LLL26], they use bit decomposition on hash coefficients $b_i \in \mathbb{Z}_p$ in order to ensure correctness of the DDLog algorithm. This is indeed still the case in our modifications, but we want to use less communication. One way to do this is to still bit decompose b_i in the public data of the construction to ensure it is small when each party performs DDLog, but then bit compose each of the shares at the end of **Compress**. This requires additionally changing the definition of **Hash** to be $\text{Hash}(\mathbf{x}) := \sum_i^W b_i \cdot \mathbf{x}[i] \pmod{\mathbb{Z}}$. Now, we are only sending one share of communication while maintaining correctness, which we show after the algorithm for completeness. We make some other minor adjustments to follow circularity as seen in Algorithm 5.

Algorithm 5. Fixed Factor Hashing under DDH

Hash.Setup($1^\lambda, s \in \mathbb{Z}_p$) :

1. For $i \in [W]$, sample

$$b_i \leftarrow \mathbb{Z}_p,$$

$$\mathbf{r}_i \leftarrow \mathbb{Z}_p^\lambda.$$

2. For $j \in [W]$, sample

$$\mathbf{a}_j \leftarrow \mathbb{Z}_p^{\lceil \log p \rceil}.$$

3. For $i \in [W]$ and $j \in [W]$, compute extraction hints

$$\text{ct}_{i,j} \leftarrow (g^{b_i})^{s \cdot \mathbf{a}_j} / g^{\mathbf{u}_j[i] \cdot \text{Bits}(s)},$$

where $\mathbf{u}_j \in \{0, 1\}^W$ denotes the j -th unit vector.

4. For $i \in [W]$, compute compression hints

$$\mathbf{c}_0^i \leftarrow g^{\mathbf{r}_i},$$

$$\mathbf{c}_1^i \leftarrow g^{\mathbf{r}_i s + \text{Bits}(s) \cdot \text{Bits}(b_i)}.$$

5. Output $\text{pd} := (\{\text{ct}_{i,j}\}, \{g^{\mathbf{a}_j}\}, \{\mathbf{c}_0^i\}, \{\mathbf{c}_1^i\}, \{g^{b_i}\})$.

Compress($\text{pd}, \langle s \cdot \mathbf{x} \rangle_{G/E}, \langle \mathbf{x} \rangle_{G,E}$) :

1. For $i \in [W]$, compute bit hashes

$$\mathbf{v}_{G/E} \leftarrow (\mathbf{c}_1^i)^{\langle \mathbf{x}[i] \rangle_{G/E}} / (\mathbf{c}_0^i)^{\langle s \cdot \mathbf{x}[i] \rangle_{G/E}}$$

$$\langle \text{Bits}(s) \cdot \text{Bits}(b_i) \cdot \mathbf{x}[i] \rangle_{G/E} \leftarrow \text{DDLog}(\mathbf{v}_{G/E}) \pmod{p}.$$

2. Compute compression

$$\langle \text{Bits}(s) \cdot \text{Bits}(h) \rangle_{G/E} \leftarrow \sum_{i=1}^W \langle \text{Bits}(s) \cdot \text{Bits}(b_i) \cdot \mathbf{x}[i] \rangle_{G/E} \pmod{p}. \quad (16)$$

3. Bit compose and output $\langle s \cdot h \rangle_{G/E}$.

Extract($\text{pd}, \langle s \cdot h \rangle_{G/E}, j, \langle \mathbf{x} \rangle_{G,E}$) :

1. Compute difference

$$\text{diff}_{G/E,j} \leftarrow \prod_{i=1}^W (\text{ct}_{i,j})^{\langle \mathbf{x}[i] \rangle_{G/E}}.$$

2. Compute share

$$\mathbf{v}_{G/E} \leftarrow (g^{\mathbf{a}_j})^{\langle s \cdot h \rangle_{G/E}} / \text{diff}_{G/E,j},$$

$$\langle \text{Bits}(s) \cdot x_j \rangle_{G/E} \leftarrow \text{DDLog}(\mathbf{v}_{G/E}) \pmod{p}.$$

3. Bit compose and output $\langle s \cdot x_j \rangle_{G/E}$.

Lemma 8.1 (Correctness of Fixed Factor Hashing under DDH). *There exists a negligible function negl such that for every security parameter $\lambda \in \mathbb{N}$, public parameters pp , encoded vectors $\mathbf{x} \in \mathbb{Z}_B^W$, secret $s \in \mathbb{Z}_p$, additive shares $\langle s \cdot \mathbf{x} \rangle, \langle s \cdot \text{Hash}(\mathbf{x}) \rangle$, and index $j \in [W]$, the following holds:*

$$\Pr \left[\langle s \cdot \mathbf{h} \rangle_E = \langle s \cdot \mathbf{h} \rangle_G + s \cdot \text{Hash}(\mathbf{x}) \mid \begin{array}{l} \text{pd} \leftarrow \text{Setup}(1^\lambda, s) \\ \langle s \cdot \mathbf{h} \rangle_G \leftarrow \text{Compress}(\langle s \cdot \mathbf{x} \rangle_G) \\ \langle s \cdot \mathbf{h} \rangle_E \leftarrow \text{Compress}(\langle s \cdot \mathbf{x} \rangle_E) \end{array} \right] > 1 - \text{negl}(\lambda).$$

$$\Pr \left[\langle s \cdot x_j \rangle_E = \langle s \cdot x_j \rangle_G + s \cdot x_j \mid \begin{array}{l} \text{pd} \leftarrow \text{Setup}(1^\lambda, s) \\ \langle s \cdot x_j \rangle_G \leftarrow \text{Extract}(\langle s \cdot \text{Hash}(\mathbf{x}) \rangle_G, j) \\ \langle s \cdot x_j \rangle_E \leftarrow \text{Extract}(\langle s \cdot \text{Hash}(\mathbf{x}) \rangle_E, j, \mathbf{x}) \end{array} \right] > 1 - \text{negl}(\lambda).$$

Proof. The proof is similar to that of Lemma 5.31 in Li, Lin, and Lu [LLL26]. We rewrite the proof here for completeness to show how our new alterations maintain correctness and don't change anything.

The correctness of **Compress** follows from the definition of **Hash**. To verify **Extract**, intermediate

values $\mathbf{v}_G$ and $\mathbf{v}_E$ must satisfy $\mathbf{v}_E = \mathbf{v}_G \cdot g^{\text{Bits}(s) \cdot x_j}$:

$$\begin{aligned}
\mathbf{v}_E &= (g^{\mathbf{a}_j})^{\langle s \cdot h \rangle_E} / \text{diff}_{E,j} \\
&= (g^{\mathbf{a}_j})^{\langle s \cdot h \rangle_{G+s \cdot h}} / \prod_i \text{ct}_{i,j}^{\langle \mathbf{x}[i] \rangle_{G+\mathbf{x}[i]}} \\
&= \mathbf{v}_G \cdot (g^{\mathbf{a}_j})^{s \cdot h} / \prod_i \text{ct}_{i,j}^{\mathbf{x}[i]} \\
&= \mathbf{v}_G \cdot (g^{\mathbf{a}_j})^{s \cdot h} / \left(\prod_i g^{s \cdot \mathbf{a}_j \cdot b_i \cdot \mathbf{x}[i]} \right) \cdot \left(\prod_i g^{\mathbf{u}_j[i] \cdot \text{Bits}(s) \cdot \mathbf{x}[i]} \right) \\
&= \mathbf{v}_G \cdot (g^{s \cdot \mathbf{a}_j})^{h - \sum_i b_i \cdot \mathbf{x}[i]} \cdot g^{\text{Bits}(s) \cdot x_j}.
\end{aligned}$$

From here, we only need to show that $h = \sum_i b_i \cdot \mathbf{x}[i]$. This is true by our new definition of `Hash`, so we have successfully shown that $\mathbf{v}_E = \mathbf{v}_G \cdot g^{\text{Bits}(s) \cdot x_j}$, completing the proof. $\square$

Lemma 8.2 (Security of Fixed Factor Hashing under DDH). *Under a variant of the Circular DDH assumption (Definition 4.5 and Definition 4.6), the distribution of public data `Hash.pd` as computed in Algorithm 5 is computationally indistinguishable from the public data sampled at random `Hash.pd`.*

Proof. The proof follows directly from Circular DDH. $\square$

8.2 Hashing under Lattices

Our alterations to the construction of fixed factor hashing under lattices is slightly more subtle than the group setting. The first attempt is to directly replace the hashing coefficients $\mathbf{b}_i$ with the scalar form rather than the bit decompose variation that Li, Lin, and Lu [LLL26] uses. This, however, would blow up the error to an unmanageable degree and destroy the correctness of the scheme since the hash coefficients contribute to the error. This is clear when looking at the structure of extraction hints $\text{ct}_{i,j,b}$ from their work. A second solution would be to sample scalar b_i from small-coefficient ring elements to reduce the error in the computation. The problem here is the security breaks down if we use scalars as hash coefficients, as the extraction hints currently rely on a non-uniform variation of RLWE that requires the dimension of $\mathbf{b}_i$ be greater than 1.

To solve these issues, our idea is to b_i from $\mathcal{R}_q$ and, rather than fully bit decompose, split each b_i into L chunks of appropriate size. These chunks need to be big enough to satisfy the security of non-uniform RLWE while also being small enough to guarantee a constant number of shares when we compress. Once b_i is in chunks, compressing requires breaking these individual chunks into their bit decomposed form. From there, each party simply multiplies their authenticated shares by each bit of the chunks, bit composes the new authenticated shares back together, then sums the corresponding chunk shares with the computed shares for each other hash coefficient. The number of authenticated shares output from this compression will be L , so we want to make sure this L is asymptotically constant. Algorithm 6 shows the adjustments we make to the algorithm itself.

To determine L , we make adjustments to the parameters of the previous works [ILL25b, LLL26]. In those works, the bound on encoded values B is typically quite large ($< 2^{\text{poly}(\lambda)}$) to accommodate arithmetic garbling and to bound the error of the lifting lemma [BKS19, Lemma 2] that comes

with a large ratio between q and p needed for correctness. However, we can pick a smaller bound for B and still get correctness based on how we decompose b_i . In particular, we will set $B = \lambda^{\omega(1)}$. Following this and keeping every other parameter measured the same, we end up with $p > B^2$, $\Delta > B^4$, and $q > B^6$. This lets us pick $L = \lceil \log q / \log B' \rceil$ where $B' \cdot W \leq B$, satisfying the constraint that L be asymptotically constant. We use B' to ensure that the hash value $\mathbf{h}[k]$ remains B -bounded when computing the hash $\mathbf{h}[k] = \sum_{i=1}^W \mathbf{x}[i] \cdot b_{i,k}$. Comparing B and p is necessary since it determines if our construction succeeds when using the lifting lemma [BKS19, Lemma 2] for correctness. The failure rate of this lemma is $O(n \cdot \|s \cdot \mathbf{h}[k]\|_\infty / p)$. Because of how we chose L , each chunk will hide roughly B information at most, meaning that the probability of failing the lifting lemma is roughly $B/p > \lambda^{-\omega(1)}$.

Algorithm 6. Fixed Factor Hashing under Lattices

Hash.Setup($1^\lambda, s$) :

1. For $i \in [W]$ and $k \in [L]$, sample

$$b_{i,k} \leftarrow \mathcal{R},$$

with integer coefficients bounded by B' .

2. For $j \in [W]$, sample

$$\mathbf{a}_j \leftarrow \mathcal{R}_q^L,$$

$$\mathbf{e}_j \leftarrow \mathcal{D}_{\text{err}}^L.$$

3. For $i \in [W]$ and $j \in [W]$, sample

$$e_{i,j} \leftarrow \mathcal{D}_{\text{err}}.$$

4. Sample

$$\alpha \leftarrow \mathcal{R}_q, \beta \leftarrow \mathcal{R}_p^L.$$

5. For $i \in [W]$ and $j \in [W]$, compute extraction hints

$$\mathbf{ct}_{i,j} \leftarrow (s \cdot \mathbf{a}_j + \mathbf{e}_j) \cdot \mathbf{b}_i + e_{i,j} - s \cdot \mathbf{u}_j[i] \cdot \Delta \pmod{q},$$

where $\mathbf{u}_j \in \{0, 1\}^W$ denotes the j -th unit vector, and $\mathbf{b}_i$ is a vector with L components $b_{i,k}$.

6. Output $\text{pd} := (\{\mathbf{ct}_{i,j}\}, \{\mathbf{a}_j\}, \{\mathbf{b}_i\}, \alpha, \beta)$.

Compress($\text{pd}, \langle s \cdot \mathbf{x} \rangle_{G/E}$) :

1. For $i \in [W]$ and $k \in [L]$, compute

$$\langle s \cdot \mathbf{x}[i] \cdot b_{i,k} \rangle_{G/E} \leftarrow \langle s \cdot \mathbf{x}[i] \rangle_{G/E} \cdot b_{i,k} \pmod{p}.$$

2. For $k \in [L]$, compute the compressions

$$\langle s \cdot \mathbf{h}[k] \rangle_{G/E} \leftarrow \sum_{i=1}^W \langle s \cdot \mathbf{x}[i] \cdot b_{i,k} \rangle_{G/E} \pmod{p}.$$

3. Output $\langle s \cdot \mathbf{h} \rangle_{G/E}$.

Extract $_G(\text{pd}, \langle s \cdot \mathbf{h} \rangle_{G/E}, j)$:

1. Compute share

$$\begin{aligned}\langle s \cdot \mathbf{h} \rangle'_G &\leftarrow \langle s \cdot \mathbf{h} \rangle_G + \beta \pmod{p}. \\ \langle s \cdot x_j \rangle'_G &\leftarrow \left(\sum_{k \in [L]} \mathbf{a}_j[k] \cdot \langle s \cdot \mathbf{h}[k] \rangle'_G \right) + \alpha \pmod{q}. \\ \langle s \cdot x_j \rangle_G &\leftarrow \lfloor \langle s \cdot x_j \rangle'_G / \Delta \rfloor \pmod{p}.\end{aligned}$$

2. Output $\langle s \cdot x_j \rangle_G$.

$\text{Extract}_E(\text{pd}, \langle s \cdot \mathbf{h} \rangle_E, j, \mathbf{x}) :$

1. Compute difference

$$\text{diff}_j \leftarrow \sum_{i=1}^W \text{ct}_{i,j} \cdot \mathbf{x}[i] \pmod{q}.$$

2. Compute the share

$$\begin{aligned}\langle s \cdot \mathbf{h} \rangle'_E &\leftarrow \langle s \cdot \mathbf{h} \rangle_E + \beta \pmod{p}. \\ \langle s \cdot x_j \rangle'_E &\leftarrow \left(\sum_{k \in [L]} \mathbf{a}_j[k] \cdot \langle s \cdot \mathbf{h}[k] \rangle'_E \right) - \text{diff}_j + \alpha \pmod{q}. \\ \langle s \cdot x_j \rangle_E &\leftarrow \lfloor \langle s \cdot x_j \rangle'_E / \Delta \rfloor \pmod{p}.\end{aligned}$$

3. Output $\langle s \cdot x_j \rangle_E$.

Lemma 8.3 (Correctness of Fixed Factor Hashing under Lattices). *There exists a negligible function negl such that for every security parameter $\lambda \in \mathbb{N}$, public parameters pp , encoded vectors $\mathbf{x} \in \{0, 1\}^W$, secret $s \in \mathcal{D}_{sk}$, additive shares $\langle s \cdot \mathbf{x} \rangle, \langle s \cdot \text{Hash}(\mathbf{x}) \rangle$, and index $j \in [W]$, the following holds:*

$$\begin{aligned}\Pr \left[\langle s \cdot \mathbf{h} \rangle_E = \langle s \cdot \mathbf{h} \rangle_G + s \cdot \text{Hash}(\mathbf{x}) \mid \begin{array}{l} \text{pd} \leftarrow \text{Setup}(1^\lambda, s) \\ \langle s \cdot \mathbf{h} \rangle_G \leftarrow \text{Compress}(\langle s \cdot \mathbf{x} \rangle_G) \\ \langle s \cdot \mathbf{h} \rangle_E \leftarrow \text{Compress}(\langle s \cdot \mathbf{x} \rangle_E) \end{array} \right] &> 1 - \text{negl}(\lambda). \\ \Pr \left[\langle s \cdot x_j \rangle_E = \langle s \cdot x_j \rangle_G + s \cdot x_j \mid \begin{array}{l} \text{pd} \leftarrow \text{Setup}(1^\lambda, s) \\ \langle s \cdot x_j \rangle_G \leftarrow \text{Extract}(\langle s \cdot \text{Hash}(\mathbf{x}) \rangle_G, j) \\ \langle s \cdot x_j \rangle_E \leftarrow \text{Extract}(\langle s \cdot \text{Hash}(\mathbf{x}) \rangle_E, j, \mathbf{x}) \end{array} \right] &> 1 - \text{negl}(\lambda).\end{aligned}$$

Proof. The proof is similar to that of Lemma 5.31 in Li, Lin, and Lu [LLL26]. We rewrite the proof here for completeness to show how our new parameter choices maintain correctness.

The correctness of **Compress** follows from the definition of **Hash**. To verify **Extract**, intermediate values $\langle s \cdot \mathbf{h}[k] \rangle'$ must for additive shares over $\mathcal{R}$ except with negligible probability. Recall that $\|\mathbf{h}[k]\|$ is bounded by $W \cdot B' = B$. Thus, the sharing holds with probability $O(n \cdot \|s \cdot \mathbf{h}[k]\|_\infty / p)$ from the randomness of β by the lifting lemma [BKS19, Lemma 2]. By our parameter choice, we have

$$n \cdot \|s \cdot \mathbf{h}[k]\|_\infty \leq \text{poly}(\lambda) \cdot B \ll p.$$

Thus, the failure probability is negligible. Once lifted, the following computation shows that $\langle s \cdot x_j \rangle'_{G/E}$ for additive shares over $\mathcal{R}_q$:

$$\begin{aligned}
\langle s \cdot x_j \rangle'_E &= \sum_k \mathbf{a}_j[k] \cdot \langle s \cdot \mathbf{h}[k] \rangle'_E - \text{diff}_j + \alpha \\
&= \sum_k \mathbf{a}_j[k] \cdot (\langle s \cdot \mathbf{h}[k] \rangle'_G + s \cdot \mathbf{h}[k]) - \text{diff}_j + \alpha \\
&= \langle s \cdot x_j \rangle'_G + s \cdot \mathbf{a}_j^\top \cdot \mathbf{h} - \text{diff}_j \\
&= \langle s \cdot x_j \rangle'_G + s \cdot \mathbf{a}_j^\top \cdot \mathbf{h} - \sum_i s \cdot (\mathbf{a}_j^\top \cdot \mathbf{b}_i) \cdot \mathbf{x}[i] \\
&\quad + \sum_i (\mathbf{e}_j^\top \cdot \mathbf{b}_i - e_{i,j}) \cdot \mathbf{x}[i] + \sum_i s \cdot \mathbf{u}_j[i] \cdot \Delta \cdot \mathbf{x}[i] \\
&= \langle s \cdot x_j \rangle'_G + s \cdot \mathbf{x}[j] \cdot \Delta + \text{err}.
\end{aligned}$$

We have that $\|\text{err}\|_\infty < \text{poly}(\lambda) \cdot B$, so we apply the rounding lemma [BKS19, Lemma 1], which states that the values $\langle s \cdot x_j \rangle_{G/E}$ are additive shares except with probability $O(n \cdot \|\text{err}\|_\infty / \Delta)$. Given our parameters, we get that

$$n \cdot \|\text{err}\|_\infty < \text{poly}(\lambda) \cdot B \ll \Delta,$$

which concludes that $\langle s \cdot x_j \rangle_{G/E}$ form additive shares of $s \cdot x_j$ except with negligible probability. $\square$

Lemma 8.4 (Security of Fixed Factor Hashing under Lattices). *Under a variant of the Circular RLWE assumption (Definition 4.7), the distribution of public data Hash.pd as computed in Algorithm 6 is computationally indistinguishable from the public data sampled at random Hash.pd.*

Proof. The proof is similar to that of Lemma 5.32 in Li, Lin, and Lu [LLL26], except we must invoke a variant of the Circular RLWE assumption. Additionally, the dimension of vectors will be $L = \lceil \log q / \log B' \rceil$ rather than $\lceil \log q \rceil$, which still fall under the security of the non-uniform RLWE as in [BLMR13]. $\square$

References

- [ACC⁺16] Prabhanjan Ananth, Yu-Chi Chen, Kai-Min Chung, Huijia Lin, and Wei-Kai Lin. Delegating RAM computations with adaptive soundness and privacy. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 3–30. Springer, Berlin, Heidelberg, October / November 2016. 1
- [ADOS22] Damiano Abram, Ivan Damgård, Claudio Orlandi, and Peter Scholl. An algebraic framework for silent preprocessing with trustless setup and active security. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 421–452. Springer, Cham, August 2022. 18, 20

- [AIK04] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0 . In *45th Annual Symposium on Foundations of Computer Science*, pages 166–175. IEEE Computer Society Press, October 2004. [5](#)
- [AKL⁺20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 403–432. Springer, Cham, May 2020. [2](#)
- [AKLS21] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. Oblivious RAM with worst-case logarithmic overhead. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part IV*, volume 12828 of *Lecture Notes in Computer Science*, pages 610–640, Virtual Event, August 2021. Springer, Cham. [2](#)
- [AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *15th Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM Press, April 1983. [40](#), [42](#), [44](#)
- [AL18] Prabhanjan Ananth and Alex Lombardi. Succinct garbling schemes from functional encryption through a local simulation paradigm. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018: 16th Theory of Cryptography Conference, Part II*, volume 11240 of *Lecture Notes in Computer Science*, pages 455–472. Springer, Cham, November 2018. [1](#)
- [AMR25] Damiano Abram, Giulio Malavolta, and Lawrence Roy. Key-homomorphic computations for RAM: Fully succinct randomised encodings and more. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025, Part III*, volume 16002 of *Lecture Notes in Computer Science*, pages 236–268. Springer, Cham, August 2025. [5](#)
- [App17] Benny Applebaum. Garbled circuits as randomized encodings of functions: a primer. Cryptology ePrint Archive, Report 2017/385, 2017. [5](#)
- [Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS ’68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery. [40](#), [42](#)
- [BGI⁺01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer, Berlin, Heidelberg, August 2001. [1](#)
- [BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under DDH. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 509–539. Springer, Berlin, Heidelberg, August 2016. [20](#)
- [BGI⁺18] Elette Boyle, Niv Gilboa, Yuval Ishai, Huijia Lin, and Stefano Tessaro. Foundations of homomorphic secret sharing. In Anna R. Karlin, editor, *ITCS 2018: 9th Innovations*

- in *Theoretical Computer Science Conference*, volume 94, pages 21:1–21:21. Leibniz International Proceedings in Informatics (LIPIcs), January 2018. [2](#)
- [BGL⁺15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM Symposium on Theory of Computing*, pages 439–448. ACM Press, June 2015. [5](#)
- [BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325. Association for Computing Machinery, January 2012. [5](#)
- [BHHO08] Dan Boneh, Shai Halevi, Michael Hamburg, and Rafail Ostrovsky. Circular-secure encryption from decision Diffie-Hellman. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 108–125. Springer, Berlin, Heidelberg, August 2008. [19](#)
- [BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society Press, May 2013. [1](#)
- [BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 784–796. ACM Press, October 2012. [1](#), [5](#)
- [BKS19] Elette Boyle, Lisa Kohl, and Peter Scholl. Homomorphic secret sharing from lattices without FHE. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 3–33. Springer, Cham, May 2019. [20](#), [55](#), [56](#), [57](#), [58](#)
- [BLMR13] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 410–428. Springer, Berlin, Heidelberg, August 2013. [58](#)
- [BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd Annual ACM Symposium on Theory of Computing*, pages 503–513. ACM Press, May 1990. [1](#)
- [BV11a] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd Annual Symposium on Foundations of Computer Science*, pages 97–106. IEEE Computer Society Press, October 2011. [5](#)
- [BV11b] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, Berlin, Heidelberg, August 2011. [19](#)

- [CCC⁺16] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Cryptography for parallel RAM from indistinguishability obfuscation. In Madhu Sudan, editor, *ITCS 2016: 7th Conference on Innovations in Theoretical Computer Science*, pages 179–190. Association for Computing Machinery, January 2016. [1](#)
- [CCHR16] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled RAM or: How to delegate your database. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 61–90. Springer, Berlin, Heidelberg, October / November 2016. [1](#)
- [CH16] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. In Madhu Sudan, editor, *ITCS 2016: 7th Conference on Innovations in Theoretical Computer Science*, pages 169–178. Association for Computing Machinery, January 2016. [1](#)
- [CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the “free-XOR” technique. In Ronald Cramer, editor, *TCC 2012: 9th Theory of Cryptography Conference*, volume 7194 of *Lecture Notes in Computer Science*, pages 39–53. Springer, Berlin, Heidelberg, March 2012. [1](#)
- [CNS18] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018: 16th Theory of Cryptography Conference, Part II*, volume 11240 of *Lecture Notes in Computer Science*, pages 636–668. Springer, Cham, November 2018. [3](#), [17](#), [39](#), [40](#), [41](#), [50](#)
- [CSLN21] T.-H. Hubert Chan, Elaine Shi, Wei-Kai Lin, and Kartik Nayak. Perfectly oblivious (parallel) RAM revisited, and improved constructions. In Stefano Tessaro, editor, *ITC 2021: 2nd Conference on Information-Theoretic Cryptography*, volume 199 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:23. Schloss Dagstuhl, July 2021. [3](#), [40](#)
- [DJ01] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier’s probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001: 4th International Workshop on Theory and Practice in Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, Berlin, Heidelberg, February 2001. [18](#)
- [DJN10] Ivan Damgård, Mads Jurik, and Jesper Buus Nielsen. A generalization of Paillier’s public-key system with applications to electronic voting. *International Journal of Information Security*, 9(6):371–385, 2010. [2](#)
- [DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In Yuval Ishai, editor, *TCC 2011: 8th Theory of Cryptography Conference*, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163. Springer, Berlin, Heidelberg, March 2011. [40](#)
- [Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178. ACM Press, May / June 2009. [5](#)

- [GGS26] Tianyao Gu, Ashrujit Ghoshal, and Elaine Shi. Zebra: Arithmetic garbled RAM for large words from DCR. In *Eurocrypt 2026*, 2026. [1](#), [2](#), [3](#), [4](#), [5](#), [6](#), [7](#), [15](#), [24](#), [40](#), [43](#), [44](#), [48](#), [50](#)
- [GHL⁺14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422. Springer, Berlin, Heidelberg, May 2014. [1](#)
- [GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In Venkatesan Guruswami, editor, *56th Annual Symposium on Foundations of Computer Science*, pages 210–229. IEEE Computer Society Press, October 2015. [1](#)
- [GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM Symposium on Theory of Computing*, pages 449–458. ACM Press, June 2015. [1](#), [11](#)
- [GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. [2](#), [40](#)
- [Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 182–194. ACM Press, May 1987. [2](#)
- [Goo14] Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In David B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 684–693. ACM Press, May / June 2014. [44](#)
- [GTRS25] Tianyao Gu, Afonso Tinoco, Sri Harish G. Rajan, and Elaine Shi. PicoGRAM: Practical garbled RAM from decisional Diffie-Hellman. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025, Part VIII*, volume 16007 of *Lecture Notes in Computer Science*, pages 247–281. Springer, Cham, August 2025. [1](#), [3](#), [6](#), [43](#)
- [Hea24] David Heath. Efficient arithmetic in garbled circuits. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part V*, volume 14655 of *Lecture Notes in Computer Science*, pages 3–31. Springer, Cham, May 2024. [7](#)
- [HHWW19] Ariel Hamlin, Justin Holmgren, Mor Weiss, and Daniel Wichs. On the plausibility of fully homomorphic encryption for RAMs. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part I*, volume 11692 of *Lecture Notes in Computer Science*, pages 589–619. Springer, Cham, August 2019. [5](#)
- [HKO22] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. EpiGRAM: Practical garbled RAM. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 3–33. Springer, Cham, May / June 2022. [1](#), [6](#), [11](#)

- [HKO23] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. Tri-state circuits - A circuit model that captures RAM. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 128–160. Springer, Cham, August 2023. [1](#), [3](#), [6](#), [7](#), [15](#), [29](#), [43](#)
- [IK00] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *41st Annual Symposium on Foundations of Computer Science*, pages 294–304. IEEE Computer Society Press, November 2000. [5](#)
- [ILL25a] Yuval Ishai, Hanjun Li, and Huijia Lin. Succinct homomorphic MACs from groups and applications. In *66th Annual Symposium on Foundations of Computer Science*, pages 2505–2545. IEEE Computer Society Press, December 2025. [1](#), [19](#), [20](#)
- [ILL25b] Yuval Ishai, Hanjun Li, and Huijia Lin. A unified framework for succinct garbling from homomorphic secret sharing. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025, Part IV*, volume 16003 of *Lecture Notes in Computer Science*, pages 390–425. Springer, Cham, August 2025. [1](#), [2](#), [8](#), [9](#), [10](#), [11](#), [13](#), [14](#), [15](#), [18](#), [19](#), [20](#), [21](#), [22](#), [23](#), [25](#), [29](#), [32](#), [35](#), [37](#), [38](#), [39](#), [55](#)
- [JLS21] Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. In Samir Khuller and Virginia Vassilevska Williams, editors, *53rd Annual ACM Symposium on Theory of Computing*, pages 60–73. ACM Press, June 2021. [1](#)
- [JLS22] Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from LPN over $\mathbb{F}_p$, DLIN, and PRGs in NC^0 . In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 670–699. Springer, Cham, May / June 2022. [1](#)
- [KLLL26] Jaehyung Kim, Hanjun Li, Huijia Lin, and Zeyu Liu. SIMD HSS and aHMAC from interval encoding with application to one-bit-per-gate garbling. In *CRYPTO 2026*, 2026. [3](#)
- [KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfssdóttir, and Igor Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, Berlin, Heidelberg, July 2008. [1](#)
- [LLL26] Hanjun Li, Huijia Lin, and George Lu. Succinct garbled circuits with low-depth garbling algorithms. In *Eurocrypt 2026*, 2026. [3](#), [4](#), [9](#), [11](#), [12](#), [14](#), [15](#), [20](#), [21](#), [22](#), [25](#), [52](#), [53](#), [54](#), [55](#), [57](#), [58](#)
- [LLH26] Can Liu, Lenny Liu, Ning Luo, and David Heath. Improved garbled RAM via garbled merge. In *ACM CCS 2026*, 2026. [1](#), [3](#), [4](#), [6](#), [7](#), [8](#), [24](#)
- [LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic RAM computation from ring LWE. In Barna Saha and

- Rocco A. Servedio, editors, *55th Annual ACM Symposium on Theory of Computing*, pages 595–608. ACM Press, June 2023. [5](#)
- [LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 523–542. Springer, Cham, August 2018. [2](#)
- [LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer, Berlin, Heidelberg, May 2013. [1](#)
- [LO17] Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 66–92. Springer, Cham, August 2017. [1](#)
- [LWYY25a] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. Authenticated BitGC for actively secure rate-one 2PC. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025, Part IV*, volume 16003 of *Lecture Notes in Computer Science*, pages 652–687. Springer, Cham, August 2025. [1](#)
- [LWYY25b] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. BitGC: Garbled circuits with 1 bit per gate. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology – EUROCRYPT 2025, Part VII*, volume 15607 of *Lecture Notes in Computer Science*, pages 437–466. Springer, Cham, May 2025. [1](#)
- [MORS24] Pierre Meyer, Claudio Orlandi, Lawrence Roy, and Peter Scholl. Rate-1 arithmetic garbling from homomorphic secret sharing. In Elette Boyle and Mohammad Mahmoody, editors, *TCC 2024: 22nd Theory of Cryptography Conference, Part IV*, volume 15367 of *Lecture Notes in Computer Science*, pages 71–97. Springer, Cham, December 2024. [1](#)
- [MORS25] Pierre Meyer, Claudio Orlandi, Lawrence Roy, and Peter Scholl. Silent circuit re-linearisation: Sublinear-size (boolean and arithmetic) garbled circuits from DCR. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025, Part IV*, volume 16003 of *Lecture Notes in Computer Science*, pages 426–458. Springer, Cham, August 2025. [1](#)
- [NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC ’99, page 129–139, New York, NY, USA, 1999. Association for Computing Machinery. [1](#)
- [Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd Annual ACM Symposium on Theory of Computing*, pages 514–523. ACM Press, May 1990. [2](#)
- [Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT’99*, vol-

- ume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, Berlin, Heidelberg, May 1999. [2](#), [18](#)
- [PLS23] Andrew Park, Wei-Kai Lin, and Elaine Shi. NanoGRAM: Garbled RAM with $\tilde{O}(\log N)$ overhead. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part I*, volume 14004 of *Lecture Notes in Computer Science*, pages 456–486. Springer, Cham, April 2023. [1](#), [6](#), [40](#)
- [RR21] Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part I*, volume 12825 of *Lecture Notes in Computer Science*, pages 94–124, Virtual Event, August 2021. Springer, Cham. [1](#)
- [RVV24] Seyoon Ragavan, Neekon Vafa, and Vinod Vaikuntanathan. Indistinguishability obfuscation from bilinear maps and LPN variants. In Elette Boyle and Mohammad Mahmoody, editors, *TCC 2024: 22nd Theory of Cryptography Conference, Part IV*, volume 15367 of *Lecture Notes in Computer Science*, pages 3–36. Springer, Cham, December 2024. [1](#)
- [Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Computer Society Press, November 1994. [2](#)
- [SvDS⁺13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 299–310. ACM Press, November 2013. [40](#)
- [Wak68] Abraham Waksman. A permutation network. *Journal of the ACM (JACM)*, 15(1):159–163, 1968. [40](#), [42](#)
- [WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015: 22nd Conference on Computer and Communications Security*, pages 850–861. ACM Press, October 2015. [2](#), [6](#), [15](#), [39](#), [40](#), [43](#), [44](#), [48](#), [49](#), [50](#)
- [Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE Computer Society Press, October 1986. [1](#)
- [YPHK23] Yibin Yang, Stanislav Peceny, David Heath, and Vladimir Kolesnikov. Towards generic MPC compilers via variable instruction set architectures (VISAs). In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 2516–2530. ACM Press, November 2023. [1](#)
- [ZLY⁺25] Wenhao Zhang, Hanlin Liu, Kang Yang, Wen jie Lu, Yu Yu, Xiao Wang, and Chenkai Weng. BitGC made (more) efficient. In *Eurocrypt 2026*, 2025. [3](#)

- [ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 220–250. Springer, Berlin, Heidelberg, April 2015. 1

A Appendix

A.1 Perfect ORAM Algorithms

For completeness, we provide below a detailed algorithmic description of perfect ORAM, split into the following three algorithm boxes.

Algorithm 7. Levels of ORAM

Level.Build($n, S = \{(\text{addr}_i, \text{data}_i)\}$):

1. If $n < |S|$, find the first $|S| - n$ number of dummy entries and remove them. That is, find first $|S| - n$ number of $j \in [n]$ such that $\text{data}_j = \perp$, and remove $(\text{addr}_j, \text{data}_j)$.
2. Let A be an array of length $2n$. Copy the input entries S into the first n locations of A , and append n special dummies $(\perp_1, \perp), (\perp_2, \perp), \dots, (\perp_n, \perp)$ to the last n locations.
3. Permute A with an oblivious random permutation π and append each entry with two attributes: **next** and **pos** = $(\ell_{id}, \text{offset})$. Set $A[i].\text{pos} \leftarrow (\perp, i)$ for every $i \in [2n]$.
4. Obviously sort A by **addr** and construct the address-position map $U = \{(\text{addr}, \text{pos})\}$ from the entries that are not dummy.
5. For each $i \in [n - 1]$, set the dummy entry $\perp_i$'s **next** to be the **offset** of $\perp_{i+1}$. With a slight abuse of notation, we use the entry's address to denote the corresponding entry. Thus, we have $\perp_i.\text{next} = \perp_{i+1}.\text{pos}.\text{offset}$. Let **dpos** be $\perp_1.\text{pos}.\text{offset}$.
6. Obviously sort A by **pos** and build an access-revealing onetime memory $\text{otm} \leftarrow \text{OTM.Build}_{2n}(A)$.
7. Return $\text{level} = (\text{otm}, \text{dpos}, 2n), U$.

Level.Read(q, offset):

1. If $q \neq \perp$, set $\text{entry} \leftarrow \text{OTM.Read}(\text{otm}, \text{offset})$. If $\text{entry}.\text{addr} \neq \perp$, set $\text{val} \leftarrow \text{entry}.\text{data}$ and then set the corresponding entry's **addr** and **data** both to be $\perp$ in the **otm**. Otherwise, set $\text{val} \leftarrow \perp$.
2. Otherwise, perform a dummy read $\text{entry} \leftarrow \text{OTM.Read}(\text{otm}, \text{dpos})$. Set $\text{val} \leftarrow \perp$, and the corresponding entry's **addr** as $\perp$ in the **otm**. Update $\text{level}.\text{dpos} \leftarrow \text{entry}.\text{next}$.
3. Return val .

Level.Getall():

1. Let A be the array of entries in the level. Obviously sort A by **addr** so that all remaining non-dummy entries are packed in front.
2. Return the first n entries after removing the metadata, namely **next** and **pos**.

Algorithm 8. Recursive Sub-ORAM

Sub – ORAM.Build(d):

1. Initialize $d + 1$ hierarchical levels: $\text{level}_j \leftarrow \perp$ for each $j \in \{0, 1, \dots, d\}$.
2. Initialize an extra space $\text{fresh} \leftarrow \perp$ containing the block that is freshly fetched.
3. Let the depth of the sub-oram depth be d .
4. Return $\text{suboram}_d \leftarrow (\text{depth}, \text{fresh}, \text{level}_0, \text{level}_1, \dots, \text{level}_d)$.

Sub – ORAM.Lookup(addr, pos):

1. If $\text{suboram}.\text{depth} = 0$, this sub-oram only contains one entry. Thus, just let ans be the data component of this entry.
2. Parse the input position label as $\text{pos} = (\ell_{id}, \text{offset})$, where ℓ_{id} is the level index and offset is the position inside level ℓ_{id} .
3. For each level $j \in \{0, 1, \dots, d\}$, perform:

$$\text{ans}_j \leftarrow \begin{cases} \text{level}_j.\text{read}(\text{addr}, \text{offset}), & \text{if } j = \ell_{id}, \\ \text{level}_j.\text{read}(\perp, \perp), & \text{otherwise.} \end{cases}$$

4. Let ans be the $\text{ans}_{\ell_{id}}$. Update $\text{suboram}.\text{fresh} \leftarrow (\text{addr}, \text{ans})$.
5. Return ans .

Sub – ORAM.Shuffle(U, ℓ):

1. Let $A \leftarrow \text{fresh} \cup \text{level}_0.\text{Getall} \cup \dots \cup \text{level}_{\ell-1}.\text{Getall}$. Set all $\text{level}_0, \text{level}_1, \dots, \text{level}_{\ell-1}$ to be $\perp$.
2. Obviously merge A and U into a single array sorted by logical address addr , placing entries from A before entries from U for equal addresses, and in one linear scan set A 's entry to $\perp$ whenever it is immediately followed by a U -entry with the same address.
3. Obviously sort $A' \leftarrow A \cup U$ by logical address addr , placing entries from U before entries from A whenever the addresses are equal.
4. Scan the sorted array A' once. For every adjacent pair with the same address, preserve the first entry, and remove the second entry.
5. Build a new level: $(\text{level}, U') \leftarrow \text{Level.Build}(2^\ell, A')$.
6. Set $\text{level}_\ell \leftarrow \text{level}$. For each $i \in [2^\ell]$, set $U'[i].\text{pos}.\ell_{id} = \ell$.
7. Return U' .

Algorithm 9. Perfect ORAM

Perfect – ORAM.Build(n):

1. Assume that n is a power of 2. Let $D = \log n$.
2. For each depth $d \in \{0, 1, \dots, D\}$, initialize a position-based ORAM $\text{suboram}_d \leftarrow \text{Sub-ORAM.Build}(d)$.
3. Return $\text{oram} \leftarrow (\text{suboram}_0, \text{suboram}_1, \dots, \text{suboram}_D)$.

Perfect – ORAM.Fetch($\text{op}, \text{addr}, \text{data}^*$):

1. For each depth $d \in \{1, \dots, D\}$, let $\text{addr}^{(d)}$ denote the depth- d prefix of addr .
2. For the trivial case sub-oram suboram_0 , $\text{ans}_0 \leftarrow \text{suboram}_0.\text{Lookup}(\perp, \perp)$. This returned ans_0 contains two positions for $\text{addr}^{(1)} = 0$ and $\text{addr}^{(1)} = 1$ in suboram_1 respectively.

3. For each depth $d = 1, \dots, D - 1$, do the following:
 - (a) Parse ans_{d-1} and extract the position label pos_d corresponding to the address $\text{addr}^{(d)}$.
 - (b) Read $\text{ans}_d \leftarrow \text{suboram}_d.\text{Lookup}(\text{addr}^{(d)}, \text{pos}_d)$. Similarly, ans_d contains two positions for $\text{addr}^{(d)}\|0$ and $\text{addr}^{(d)}\|1$ in suboram_{d+1} respectively
4. Parse ans_{D-1} and extract the position label pos_D corresponding to the address $\text{addr}^{(D)} = \text{addr}$. Let $\text{val} \leftarrow \text{suboram}_D.\text{Lookup}(\text{addr}, \text{pos}_D)$.
5. If $\text{op} = \text{write}$, update $\text{suboram}_D.\text{fresh} \leftarrow (\text{addr}, \text{data}^*)$.
6. Call $\text{Perfect-ORAM.Maintain}()$.
7. Return val .

Perfect – ORAM.Maintain():

1. Let ℓ_D be the smallest level index of suboram_D such that $\text{suboram}.\text{level}_{\ell_D} = \perp$. If no such level exists, set $\ell_D \leftarrow D$.
2. Set the deepest update array as $U_D \leftarrow \emptyset$.
3. For $d = D, D - 1, \dots, 0$, do the following:
 - (a) Let $\ell_d \leftarrow \min(d, \ell_D)$.
 - (b) Rebuild the current recursion depth: $U'_d \leftarrow \text{suboram}_d.\text{Shuffle}(U_d, \ell_d)$.
 - (c) If $d > 0$, convert the returned address-position map into the update array of the next smaller depth: $U_{d-1} \leftarrow \text{Convert}(U'_d)$.
4. Return.

Perfect – ORAM.Convert($U = \{(\text{addr}_i, \text{pos}_i)\}$):

1. Obviously sort U by addr .
2. Let d be the bit length of addr . Scan the sorted array once. For every adjacent pair $(\text{addr}^{(d-1)}\|0, \text{pos}^0), (\text{addr}^{(d-1)}\|1, \text{pos}^1)$ with the same prefix $\text{addr}^{(d-1)}$, replace the first entry by $(\text{addr}^{(d-1)}, (\text{pos}^0, \text{pos}^1))$ and remove the second one.
3. Return the resulting array.

B Disclosure of AI Usage

We use generative AI tools (ChatGPT and/or Gemini) to polish our existing draft, including the Abstract and Preliminaries. All drafts are written by the authors prior to the AI usage, and all AI-assisted edits are limited to rephrasing or reordering. We verified the correctness and originality of all content, including references.