Laconic Oblivious Transfer and its Applications

Chongwon Cho
HRL Laboratories

Nico Döttling*
UC Berkeley

Sanjam Garg*
UC Berkeley

Divya Gupta*,†
Microsoft Research India

Peihan Miao*
UC Berkeley

Antigoni Polychroniadou‡
Cornell University

Abstract

In this work, we introduce a novel technique for secure computation over large inputs. Specifically, we provide a new oblivious transfer (OT) protocol with a laconic receiver. Laconic OT allows a receiver to commit to a large input D (of length M) via a short message. Subsequently, a single short message by a sender allows the receiver to learn $m_{D[L]}$, where the messages m_0, m_1 and the location $L \in [M]$ are dynamically chosen by the sender. All prior constructions of OT required the receiver's outgoing message to grow with D.

Our key contribution is an instantiation of this primitive based on the Decisional Diffie-Hellman (DDH) assumption in the common reference string (CRS) model. The technical core of this construction is a novel use of somewhere statistically binding (SSB) hashing in conjunction with hash proof systems. Next, we show applications of laconic OT to non-interactive secure computation on large inputs and multi-hop homomorphic encryption for RAM programs.

*Research supported in part from 2017 AFOSR YIP Award, DARPA/ARL SAFEWARE Award W911NF15C0210, AFOSR Award FA9550-15-1-0274, NSF CRII Award 1464397, and research grants by the Okawa Foundation, Visa Inc., and Center for Long-Term Cybersecurity (CLTC, UC Berkeley). The views expressed are those of the author and do not reflect the official policy or position of the funding agencies.

†Work done while at University of California, Berkeley.

‡Part of the work done while visiting University of California, Berkeley. Research supported in part the National Science Foundation under Grant No. 1617676, IBM under Agreement 4915013672, and the Packard Foundation under Grant 2015-63124.

Contents

1 Introduction

Big data poses serious challenges for the current cryptographic technology. In particular, cryptographic protocols for secure computation are typically based on Boolean circuits, where both the computational complexity and communication complexity scale with the size of the input dataset, which makes it generally unsuitable for even moderate dataset sizes. Over the past few decades, substantial effort has been devoted towards realizing cryptographic primitives that overcome these challenges. This includes works on fully-homomorphic encryption (FHE) [Gen09, BV11b, BV11a, GSW13] and on the RAM setting of oblivious RAM [Gol87, Ost90] and secure RAM computation [OS97, GKK+12, LO13, GHL+14, GGMP16]. Protocols based on FHE generally have a favorable communication complexity and are basically non-interactive, yet incur a prohibitively large computational overhead (dependent on the dataset size). On the other hand, protocols for the RAM model generally have a favorable computational overhead, but lack in terms of communication efficiency (that grows with the program running time), especially in the multi-party setting. Can we achieve the best of both worlds? In this work we make positive progress on this question. Specifically, we introduce a new tool called laconic oblivious transfer that helps to strike a balance between the two seemingly opposing goals.

Oblivious transfer (or OT for short) is a fundamental and powerful primitive in cryptography [Kil88, IPS08]. Since its first introduction by Rabin [Rab81], OT has been a foundational building block for realizing secure computation protocols [Yao82, GMW87, IPS08]. However, typical secure computation protocols involve executions of multiple instances of an oblivious transfer protocol. In fact, the number of needed oblivious transfers grows with the input size of one of the parties, which is the receiver of the oblivious transfer.[1] In this work, we observe that a two-message OT protocol, with a short message from the receiver, can be a key tool towards the goal of obtaining *simultaneous* improvements in computational and communication cost for secure computation.

1.1 Laconic OT

In this paper, we introduce the notion of laconic oblivious transfer (or laconic OT for short). Laconic OT allows an OT receiver to commit to a large input $D \in \{0,1\}^M$ via a short message. Subsequently, the sender responds with a single short message to the receiver depending on dynamically chosen two messages m_0, m_1 and a location $L \in [M]$. The sender's response message allows the receiver to recover $m_{D[L]}$ (while $m_{1-D[L]}$ remains computationally hidden). Furthermore, without any additional communication with the receiver, the sender could repeat this process for multiple choices of L. The construction we give is secure against semi-honest adversaries, but it can be upgraded to the malicious setting in a similar way as we will discuss in Section 1.2 for the first application.

Our construction of laconic OT is obtained by first realizing a "mildly compressing" laconic OT protocol for which the receiver's message is factor-2 compressing, i.e., half the size of its input. We base this construction on the Decisional Diffie-Hellman (DDH) assumption. We note that, subsequent to our work, the factor-2 compression construction has been simplified by Döttling and Garg [DG17] (another alternative simplification can be obtained using [AIKW13]). Next we show that such a "mildly compressing" laconic OT can be bootstrapped, via the usage of a Merkle Hash

[1] We remark that related prior works on OT extension [Bea96, IKNP03, KK13, ALSZ13] makes the number of public key operations performed during protocol executions independent of the receiver's input size. However, the communication complexity of receivers in these protocols still grows with the input size of the receiver.

Tree and Yao's Garbled Circuits [Yao82], to obtain a "fully compressing" laconic OT, where the size of the receiver's message is independent of its input size. The laconic OT scheme with a Merkle Tree structure allows for good properties like local verification and local updates, which makes it a powerful tool in secure computation with large inputs.

We will show new applications of laconic OT to non-interactive secure computation and homomorphic encryption for RAM programs, as briefly described below in Sections 1.2 and 1.3.

1.2 Warm-Up Application: Non-Interactive Secure Computation on Large Inputs

Can a receiver publish a (small) encoding of her large confidential database D so that any sender, who holds a secret input x, can reveal the output $f(x, D)$ (where f is a circuit) to the receiver by sending her a single message? For security, we want the receiver's encoding to hide D and the sender's message to hide x. Using laconic OT, we present the first solution to this problem. In our construction, the receiver's published encoding is independent of the size of her database, but we do not restrict the size of the sender's message.[2]

RAM Setting. Consider the scenario where f can be computed using a RAM program P of running time t. We use the notation $P^D(x)$ to denote the execution of the program P on input x with random access to the database D. We provide a construction where as before the size of the receiver's published message is independent of the size of the database D. Moreover, the size of the sender's message (and computational cost of the sender and the receiver) grows only with t and the receiver learns nothing more than the output $P^D(x)$ and the locations in D touched during the computation. Note that in all prior works on general secure RAM computation [OS97, GKK+12, LO13, WHC+14, GHL+14, GLOS15, GLO15] the size of the receiver's message grew at least with its input size.[3]

Against Malicious Adversaries. The results above are obtained in the semi-honest setting. We can upgrade to security against a malicious sender by use of (i) non-interactive zero knowledge proofs (NIZKs) [FLS90] at the cost of additionally assuming doubly enhanced trapdoor permutations or bilinear maps [CHK04, GOS06], (ii) the techniques of Ishai et al. [IKO+11] while obtaining slightly weaker security,[4] or (iii) interactive zero-knowledge proofs but at the cost of additional interaction.

Upgrading to security against a malicious receiver is tricky. This is because the receiver's public encoding is short and hence, it is not possible to recover the receiver's entire database just given the encoding. Standard simulation-based security can be obtained by using (i) universal arguments

[2] We remark that solutions for this problem based on fully-homomorphic encryption (FHE) [Gen09, LNO13], unlike our result, reduce the communication cost of both the sender's and the receiver's messages to be independent of the size of D, but require additional rounds of interaction.

[3] The communication cost of the receiver's message can be reduced to depend only on the running time of the program by allowing round complexity to grow with the running time of the program (using Merkle Hashing). Analogous to the circuit case, we remark that FHE-based solutions can make the communication of both the sender and the receiver small, but at the cost of extra rounds. Moreover, in the setting of RAM programs FHE-based solutions additionally incur an increased computational cost for the receiver. In particular, the receiver's computational cost grows with the size of its database.

[4] The receiver is required to keep the output of the computation private.

as done by [CV12, COV15] at the cost of additional interaction, or (ii) using SNARKs at the cost of making extractability assumptions [BCCT12, BSCG+13].[5]

Other Related Work. Prior works consider secure computation which hides the input size of one [MRK03,IP07,ADT11,LNO13] or both parties [LNO13]. Our notion only requires the receiver's communication cost to be independent of the its input size, and is therefore weaker. However, these results are largely restricted to special functionalities, such as zero-knowledge sets and computing certain branching programs (which imply input-size hiding private set intersection). The general result of [LNO13] uses FHE and as mentioned earlier needs more rounds of interaction.[6]

1.3 Main Application: Muti-Hop Homomorphic Encryption for RAM Programs

Consider a scenario where S (a server), holding an input x, publishes an encryption ct_0 of her private input x under her public key. Now this ciphertext is passed on to a client Q_1 that homomorphically computes a (possibly private) program P_1 accessing (private) memory D_1 on the value encrypted in ct_0, obtaining another ciphertext ct_1. More generally, the computation could be performed by multiple clients. In other words, clients $Q_2, Q_3, \cdots$ could sequentially compute private programs $P_2, P_3, \cdots$ accessing their own private databases $D_2, D_3, \cdots$. Finally, we want S to be able to use her secret key to decrypt the final ciphertext and recover the output of the computation. For security, we require simulation based security for a client Q_i against a collusion of the server and any subset of the clients, and IND-CPA security for the server's ciphertext.

Though we described the simple case above, we are interested in the general case when computation is performed in different sequences of the clients. Examples of two such computation paths are shown in Figure 1. Furthermore, we consider the setting of persistent databases, where each client is able to execute dynamically chosen programs on the encrypted ciphertexts while using the same database that gets updated as these programs are executed.

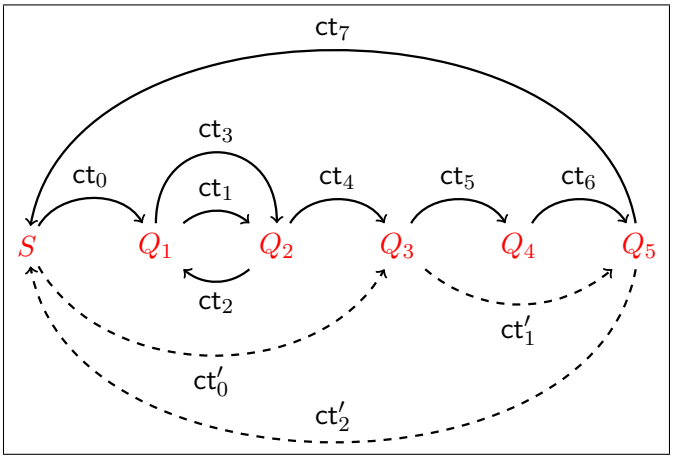


Figure 1: Two example paths of computation on server S's ciphertexts.

FHE-Based Solution. Gentry's [Gen09] fully homomorphic encryption (FHE) scheme offers a solution to the above problem when circuit representations of the desired programs $P_1, P_2, \ldots$ are considered. Specifically, S could encrypt her input x using an FHE

[5] We finally note that relaxing to the weaker notion of indistinguishability-based security we can expect to obtain the best of both worlds, i.e. a non-interactive solution while making only a black-box use of the adversary (a.k.a. avoiding the use of extractability assumptions). We leave this open for future work.

[6] We remark that in an orthogonal work of Hubacek and Wichs [HW15] obtain constructions where the communication cost is independent of the length of the output of the computation using indistinguishability obfuscation [GGH+13b].

scheme. Now, the clients can publicly compute arbitrary programs on the encrypted value using a public evaluation procedure. This procedure can be adapted to preserve the privacy of the computed circuit [OPP14, DS16, BPMW16] as well. However, this construction only works for circuits. Realizing the scheme for RAM programs involves first converting the RAM program into a circuit of size at least linear in the size of the database. This linear effort can be exponential in the running time of the program for several applications of interest such as binary search.

Our Relaxation. In obtaining homomorphic encryption for RAM programs, we start by relaxing the compactness requirement in FHE.[7] Compactness in FHE requires that the size of the ciphertexts does not grow with computation. In particular, in our scheme, we allow the evaluated ciphertexts to be bigger than the original ciphertext. Gentry, Halevi and Vaikuntanathan [GHV10] considered an analogous setting for the case of circuits. As in Gentry et al. [GHV10], in our setting computation itself will happen at the time of decryption. Therefore, we additionally require that clients $Q_1, Q_2, \cdots$ first ship pre-processed versions of their databases to S for the decryption, and security will additionally require that S does not learn the access pattern of the programs on client databases. This brings us to the following question:

Can we realize multi-hop encryption schemes for RAM programs where the ciphertext grows linearly only in the running time of the computation performed on it?

We show that laconic OT can be used to realize such a multi-hop homomorphic encryption scheme for RAM programs. Our result bridges the gap between growth in ciphertext size and computational complexity of homomorphic encryption for RAM programs.

Our work also leaves open the problem of realizing (fully or somewhat) homomorphic encryption for RAM programs with (somewhat) compact ciphertexts and for which computational cost grows with the running time of the computation, based on traditional computational assumptions. Our solution for multi-hop RAM homomorphic encryption is for the semi-honest (or, semi-malicious) setting only. We leave open the problem of obtaining a solution in the malicious setting.[8]

1.4 Roadmap

We now lay out a roadmap for the remainder of the paper. In Section 2 we give a technical overview of this work. We introduce the notion of laconic OT formally in Section 3, and give a construction with factor-2 compression in Section 4, which can be bootstrapped to a fully compressing updatable laconic OT in Section 5. Finally we present our two applications in Sections 6 and 7.

[7] One method for realizing homomorphic encryption for RAM programs [GKP+13, GHRW14, CHJV15, BGL+15, KLW15] would be to use obfuscation [GGH+13b] based on multilinear maps [GGH13a]. However, in this paper we focus on basing homomorphic RAM computation on DDH and defer the work on obfuscation to future work.

[8] Using NIZKs alone does not solve the problem, because locations accessed during computation are dynamically decided.

2 Technical Overview

2.1 Laconic OT

We will now provide an overview of laconic OT and our constructions of this new primitive. Laconic OT consists of two major components: a hash function and an encryption scheme. We will call the hash function Hash and the encryption scheme $(\mathsf{Send}, \mathsf{Receive})$. In a nutshell, laconic OT allows a receiver R to compute a *succinct* digest digest of a large database D and a private state $\hat{\mathsf{D}}$ using the hash function Hash. After digest is made public, anyone can non-interactively send OT messages to R w.r.t. a location L of the database such that the receiver's choice bit is $D[L]$. Here, $D[L]$ is the database-entry at location L. In more detail, given digest, a database location L, and two messages m_0 and m_1, the algorithm Send computes a ciphertext e such that R, who owns $\hat{\mathsf{D}}$, can use the decryption algorithm $\mathsf{Receive}$ to decrypt e to obtain the message $m_{D[L]}$.

For security, we require sender privacy against semi-honest receiver. In particular, given an honest receiver's view, which includes the database D, the message $m_{1-D[L]}$ is computationally hidden. We formalize this using a simulation based definition. On the other hand, we do not require receiver privacy as opposed to standard oblivious transfer, namely, no security guarantee is provided against a cheating (semi-honest) sender. This is mostly for ease of exposition. Nevertheless, adding receiver privacy to laconic OT can be done in a straightforward manner via the usage of garbled circuits and two-message OT (see Section 3.1 for a detailed discussion).

For efficiency, we have the following requirement: First, the size of digest only depends on the security parameter and is independent of the size of the database D. Moreover, after digest and $\hat{\mathsf{D}}$ are computed by Hash, the workload of *both* the sender and receiver (that is, the runtime of both Send and $\mathsf{Receive}$) becomes essentially independent of the size of the database (i.e., depending at most polynomially on $\log(|D|)$).

Notice that our security definition and efficiency requirement immediately imply that the Hash algorithm used to compute the succinct digest must be collision resistant. Thus, it is clear that the hash function must be keyed and in our case it is keyed by a common reference string.

Construction at a high level. We first construct a laconic OT scheme with factor-2 compression, which compresses a 2λ-bit database to a λ-bit digest. Next, to get laconic OT for databases of arbitrary size, we bootstrap this construction using an interesting combination of Merkle hashing and garbled circuits. Below, we give an overview of each of these steps.

2.1.1 Laconic OT with Factor-2 Compression

We start with a construction of a laconic OT scheme with factor-2 compression, i.e., a scheme that hashes a 2λ-bit database to a λ-bit digest. This construction is inspired by the notion of witness encryption [GGSW13]. We will first explain the scheme based on witness encryption. Then, we show how this specific witness encryption scheme can be realized with the more standard notion of hash proof systems (HPS) [CS02]. Our overall scheme will be based on the security of Decisional Diffie-Hellman (DDH) assumption.

Construction Using Witness Encryption. Recall that a witness encryption scheme is defined for an NP-language $\mathcal{L}$ (with corresponding witness relation $\mathcal{R}$). It consists of two algorithms Enc and Dec. The algorithm Enc takes as input a problem instance x and a message m, and

produces a ciphertext. A recipient of the ciphertext can use Dec to decrypt the message if $x \in \mathcal{L}$ and the recipient knows a witness w such that $\mathcal{R}(x, w)$ holds. There are two requirements for a witness encryption scheme, correctness and security. Correctness requires that if $\mathcal{R}(x, w)$ holds, then $\mathsf{Dec}(x, w, \mathsf{Enc}(x, m)) = m$. Security requires that if $x \notin \mathcal{L}$, then $\mathsf{Enc}(x, m)$ computationally hides m.

We will now discuss how to construct a laconic OT with factor-2 compression using a two-to-one hash function and witness encryption. Let $\mathsf{H} : \mathcal{K} \times \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$ be a keyed hash function, where $\mathcal{K}$ is the key space. Consider the language $\mathcal{L} = \{(K, L, y, b) \in \mathcal{K} \times [2\lambda] \times \{0,1\}^{\lambda} \times \{0,1\} \mid \exists D \in \{0,1\}^{2\lambda}$ such that $\mathsf{H}(K, D) = y$ and $D[L] = b\}$. Let $(\mathsf{Enc}, \mathsf{Dec})$ be a witness encryption scheme for the language $\mathcal{L}$.

The laconic OT scheme is as follows: The Hash algorithm computes $y = \mathsf{H}(K, D)$ where K is the common reference string and $D \in \{0,1\}^{2\lambda}$ is the database. Then y is published as the digest of the database. The Send algorithm takes as input K, y, a location L, and two messages (m_0, m_1) and proceeds as follows. It computes two ciphertexts $\mathsf{e}_0 \leftarrow \mathsf{Enc}((K, L, y, 0), m_0)$ and $\mathsf{e}_1 \leftarrow \mathsf{Enc}((K, L, y, 1), m_1)$ and outputs $\mathsf{e} = (\mathsf{e}_0, \mathsf{e}_1)$. The $\mathsf{Receive}$ algorithm takes as input K, L, y, D, and the ciphertext $\mathsf{e} = (\mathsf{e}_0, \mathsf{e}_1)$ and proceeds as follows. It sets $b = D[L]$, computes $m \leftarrow \mathsf{Dec}((K, L, y, b), D, \mathsf{e}_b)$ and outputs m.

It is easy to check that the above scheme satisfies correctness. However, we run into trouble when trying to prove sender privacy. Since H compresses 2λ bits to λ bits, most hash values have exponentially many pre-images. This implies that for most values of (K, L, y), it holds that both $(K, L, y, 0) \in \mathcal{L}$ and $(K, L, y, 1) \in \mathcal{L}$, that is, most problem instances are yes-instances. However, to reduce sender privacy of our scheme to the security of witness encryption, we ideally want that if $y = \mathsf{H}(K, D)$, then $(K, L, y, D[L]) \in \mathcal{L}$ while $(K, L, y, 1 - D[L]) \notin \mathcal{L}$. To overcome this problem, we will use a somewhere statistically binding hash function that allows us to artificially introduce no-instances as described below.

Somewhere Statistically Binding Hash to the Rescue. Somewhere statistically binding (SSB) hash functions [HW15, KLW15, OPWW15] support a special key generation procedure such that the hash value information theoretically fixes certain bit(s) of the pre-image. In particular, the special key generation procedure takes as input a location L and generates a key $K^{(L)}$. Then the hash function keyed by $K^{(L)}$ will bind the L-th bit of the pre-image. That is, $K^{(L)}$ and $y = \mathsf{H}(K^{(L)}, D)$ uniquely determines $D[L]$. The security requirement for SSB hashing is the *index-hiding* property, i.e., keys $K^{(L)}$ and $K^{(L')}$ should be computationally indistinguishable for any $L \neq L'$.

We can now establish security of the above laconic OT scheme when instantiated with SSB hash functions. To prove security, we will first replace the key K by a key $K^{(L)}$ that statistically binds the L-th bit of the pre-image. The index hiding property guarantees that this change goes unnoticed. Now for every hash value $y = \mathsf{H}(K^{(L)}, D)$, it holds that $(K, L, y, D[L]) \in \mathcal{L}$ while $(K, L, y, 1 - D[L]) \notin \mathcal{L}$. We can now rely on the security of witness encryption to argue that $\mathsf{Enc}((K^{(L)}, L, y, 1 - D[L]), m_{1-D[L]})$ computationally hides the message $m_{1-D[L]}$.

Working with DDH. The above described scheme relies on a witness encryption scheme for the language $\mathcal{L}$. We note that witness encryption for general NP languages is only known under strong assumptions such as graded encodings [GGSW13] or indistinguishability obfuscation [GGH+13b]. Nevertheless, the aforementioned laconic OT scheme does not need full power of general witness

encryption. In particular, we will leverage the fact that hash proof systems [CS02] can be used to construct statistical witness encryption schemes for specific languages [GGSW13]. Towards this end, we will carefully craft an SSB hash function that is hash proof system friendly, that is, allows for a hash proof system (or statistical witness encryption) for the language $\mathcal{L}$ required above. Our construction of the HPS-friendly SSB hash is based on the Decisional Diffie-Hellman assumption and is inspired from a construction by Okamoto et al. [OPWW15].

We will briefly outline our HPS-friendly SSB hash below. We strongly encourage the reader to see Section 4.2 for the full construction or see [DG17] for a simplified construction.

Let $\mathbb{G}$ be a (multiplicative) cyclic group of order p generated by a generator g. A hashing key is of the form $\hat{\mathbf{H}} = g^{\mathbf{H}}$ (the exponentiation is done component-wisely), where the matrix $\mathbf{H} \in \mathbb{Z}_p^{2\times 2\lambda}$ is chosen uniformly at random. The hash function of $\mathbf{x} \in \mathbb{Z}_p^{2\lambda}$ is computed as $\mathsf{H}(\hat{\mathbf{H}}, \mathbf{x}) = \hat{\mathbf{H}}^{\mathbf{x}} \in \mathbb{G}^2$ (where $(\hat{\mathbf{H}}^{\mathbf{x}})_i = \prod_{k=1}^{2\lambda} \hat{\mathbf{H}}_{i,k}^{x_k}$, hence $\hat{\mathbf{H}}^{\mathbf{x}} = g^{\mathbf{H}\mathbf{x}}$). The binding key $\hat{\mathbf{H}}^{(i)}$ is of the form $\hat{\mathbf{H}}^{(i)} = g^{\mathbf{A}+\mathbf{T}}$, where $\mathbf{A} \in \mathbb{Z}_p^{2\times 2\lambda}$ is a random rank 1 matrix, and $\mathbf{T} \in \mathbb{Z}_p^{2\times 2\lambda}$ is a matrix with zero entries everywhere, except that $\mathbf{T}_{2,i} = 1$.

Now we describe a witness encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ for the language $\mathcal{L} = \{(\hat{\mathbf{H}}, i, \hat{\mathbf{y}}, b) \mid \exists \mathbf{x} \in \mathbb{Z}_p^{2\lambda} \text{ s.t. } \hat{\mathbf{H}}^{\mathbf{x}} = \hat{\mathbf{y}} \text{ and } x_i = b\}$. $\mathsf{Enc}((\hat{\mathbf{H}}, i, \hat{\mathbf{y}}, b), m)$ first sets

$$\hat{\mathbf{H}}' = \begin{pmatrix} \hat{\mathbf{H}} \\ g^{\mathbf{e}_i^\top} \end{pmatrix} \in \mathbb{G}^{3\times 2\lambda}, \hat{\mathbf{y}}' = \begin{pmatrix} \hat{\mathbf{y}} \\ g^b \end{pmatrix} \in \mathbb{G}^3,$$

where $\mathbf{e}_i \in \mathbb{Z}_p^{2\lambda}$ is the i-th unit vector. It then picks a random $\mathbf{r} \in \mathbb{Z}_p^3$ and computes a ciphertext $c = \left(\left((\hat{\mathbf{H}}')^\top\right)^{\mathbf{r}}, \left((\hat{\mathbf{y}}')^\top\right)^{\mathbf{r}} \oplus m\right)$. To decrypt a ciphertext $c = (\hat{\mathbf{h}}, z)$ given a witness $\mathbf{x} \in \mathbb{Z}_p^{2\lambda}$, we compute $m = z \oplus \hat{\mathbf{h}}^{\mathbf{x}}$. It is easy to check correctness. For the security proof, see Section 4.3.

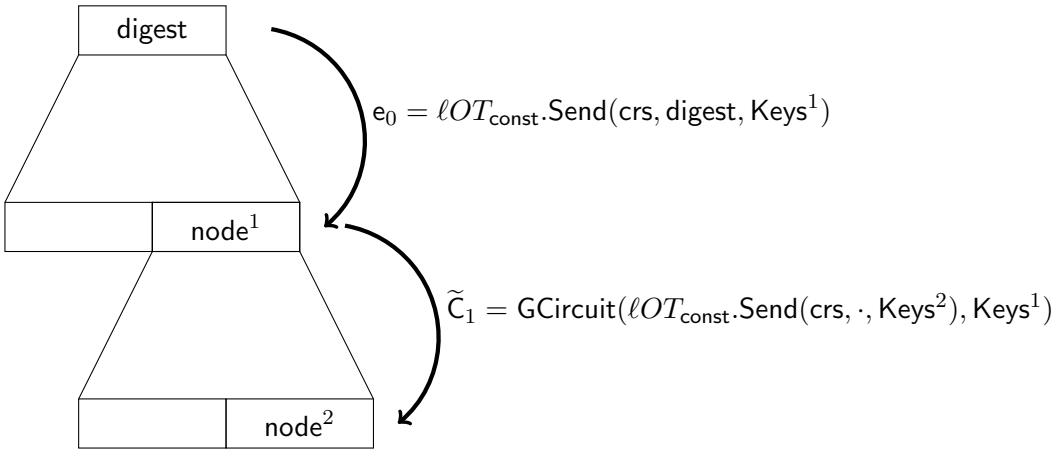
2.1.2 Bootstrapping Laconic OT

We will now provide a bootstrapping technique that constructs a laconic OT scheme with arbitrary compression factor from one with factor-2 compression. Let ℓOT_{const} denote a laconic OT scheme with factor-2 compression.

Bootstrapping the Hash Function via a Merkle Tree. A binary Merkle tree is a natural way to construct hash functions with an arbitrary compression factor from two-to-one hash functions, and this is exactly the route we pursue. A binary Merkle tree is constructed as follows: The database is split into blocks of λ bits, each of which forms the leaf of the tree. An interior node is computed as the hash value of its two children via a two-to-one hash function. This structure is defined recursively from the leaves to the root. When we reach the root node (of λ bits), its value is defined to be the (succinct) hash value or digest of the entire database. This procedure defines the hash function.

The next step is to define the laconic OT algorithms Send and $\mathsf{Receive}$ for the above hash function. Our first observation is that given the digest, the sender can transfer specific messages corresponding to the values of the left and right children of the root (via 2λ executions of $\ell OT_{\mathsf{const}}.\mathsf{Send}$). Hence, a naive approach for the sender is to output ℓOT_{const} encryptions for the path of nodes from the root to the leaf of interest. This approach runs into an immediate issue because to compute ℓOT_{const} encryptions at any layer other than the root, the sender needs to know the value at that internal node. However, in the scheme a sender only knows the value of the root and nothing else.

Traversing the Merkle Tree via Garbled Circuits. Our main idea to make the above naive idea work is via an interesting usage of garbled circuits. At a high level, the sender will output a sequence of garbled circuits (one per layer of the tree) to transfer messages corresponding to the path from the root to the leaf containing the L-th bit, so that the receiver can traverse the Merkle tree from the root to the leaf as illustrated in Figure 2.



Above GCircuit is a circuit garbling procedure, which garbles the circuit $\ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \cdot, \mathsf{Keys}^2)$ using input keys Keys^1 (see Section 5.1.1for the definition of garbled circuits).

Figure 2: The Bootstrapping Step

In more detail, the construction works as follows: The Send algorithm outputs ℓOT_{const} encryptions using the root digest and a collection of garbled circuits, one per layer of the Merkle tree. The i-th circuit has a bit b hardwired in it, which specifies whether the path should go to the left or right child at the i-th layer. It takes as input a pair of sibling nodes $(\mathsf{node}_0, \mathsf{node}_1)$ along the path at layer i and outputs ℓOT_{const} encryptions corresponding to nodes on the path at layer $i+1$ w.r.t. node_b as the hash value. Conceptually, the circuit computes ℓOT_{const} encryptions for the next layer.

The ℓOT_{const} encryptions at the root encrypt the input keys of the first garbled circuit. In the garbled circuit at layer i, the messages being encrypted/sent correspond to the input keys of the garbled circuit at layer $i+1$. The last circuit takes two sibling leaves as input which contains $D[L]$, and outputs ℓOT_{const} encryptions of m_0 and m_1 corresponding to location L (among the 2λ locations).

Given a laconic OT ciphertext, which consists of ℓOT_{const} ciphertexts w.r.t. the root digest and a sequence of garbled circuits, the receiver can traverse the Merkle tree as follows. First he runs $\ell OT_{\mathsf{const}}.\mathsf{Receive}$ for the ℓOT_{const} ciphertexts using as witness the children of the root, obtaining the input labels corresponding to these to be fed into the first garbled circuit. Next, he uses the input labels to evaluate the first garbled circuit, obtaining ℓOT_{const} ciphertexts for the second layer. He then runs $\ell OT_{\mathsf{const}}.\mathsf{Receive}$ again for these ciphertexts using as witness the children of the second node on the path. This procedure continues till the last layer.

Security of the construction can be established using the sender security of $\ell OT_{\mathsf{const}}.\mathsf{Receive}$ and simulation based security of the circuit garbling scheme.

Extension. Finally, for our RAM applications we need a slightly stronger primitive which we call *updatable laconic OT* that additionally allows for modifications/writes to the database while ensuring that the digest is updated in a consistent manner. The construction sketched in this paragraph can be modified to support this stronger notion. For a detailed description of this notion refer to Section 3.2.

2.2 Non-interactive Secure Computation on Large Inputs

The Circuit Setting. This is the most straightforward application of laconic OT. We will provide a non-interactive secure computation protocol where the receiver R, holding a large database D, publishes a short encoding of it such that any sender S, with private input x, can send a single message to reveal $C(x, D)$ to R. Here, C is the circuit being evaluated.

Recall the garbled circuit based approach to non-interactive secure computation, where R can publish the first message of a two-message oblivious transfer (OT) for his input D, and the sender responds with a garbled circuit for $C[x, \cdot]$ (with hardcoded input x) and sends the input labels corresponding to D via the second OT message. The downside of this protocol is that R's public message grows with the size of D, which could be substantially large.

We resolve this issue via our new primitive laconic OT. In our protocol, R's first message is the digest digest of his large database D. Next, the sender generates the garbled circuit for $C[x, \cdot]$ as before. It also transfers the labels for each location of D via laconic OT Send messages. Hence, by efficiency requirements of laconic OT, the length of R's public message is independent of the size of D. Moreover, sender privacy against a semi-honest receiver follows directly from the sender privacy of laconic OT and security of garbled circuits. To achieve receiver privacy, we can enhance the laconic OT with receiver privacy (discussed in Section 3.1).

The RAM Setting. This is the RAM version of the above application where S holds a RAM program P and R holds a large database D. As before, we want that (1) the length of R's first message is independent of $|D|$, (2) R's first message can be published and used by multiple senders, (3) the database is persistent for a sequence of programs for every sender, and (4) the computational complexity of both S and R per program execution grows only with running time of the corresponding program. For this application, we only achieve unprotected memory access (UMA) security against a corrupt receiver, i.e., the memory access pattern in the execution of $P^D(x)$ is leaked to the receiver. We achieve full security against a corrupt sender.

For simplicity, consider a read-only program such that each CPU step outputs the next location to be read based on the value read from last location. At a high level, since we want the sender's complexity to grow only with the running time t of the program, we cannot create a garbled circuit that takes D as input. Instead, we would go via the garbled RAM based approaches where we have a sequence of t garbled circuits where each circuit executes one CPU step. A CPU step circuit takes the current CPU state and the last bit read from the database D as input and outputs an updated state and a new location to be read. The new location would be read from the database and fed into the next CPU step. The most non-trivial part in all garbled RAM constructions is being able to compute the correct labels for the next circuit based on the value of $D[L]$, where L is the location being read. Since we are working with garbled circuits, it is crucial for security that the receiver does not learn two labels for any input wire. We solve this issue via laconic OT as follows.

For the simpler case of sender security, R publishes the short digest of D, which is fed into the first garbled circuit and this digest is passed along the sequence of garbled circuits. When a circuit wants to read a location L, it outputs the laconic OT ciphertexts which encrypt the input keys for the next circuit and use digest of D as the hash value.[9] Security against a corrupt receiver follows from the sender security of laconic OT and security of garbled circuits. To achieve security against a corrupt sender, R does not publishes digest in the clear. Instead, the labels for digest for the first circuit are transferred to R via regular OT.

Note that the garbling time of the sender as well as execution time of the receiver will grow only with the running time of the program. This follows from the efficiency requirements of laconic OT.

Above, we did not describe how we deal with general programs that also write to the database or memory. We achieve this via updatable laconic OT (for definition see Section 3.2), This allows for transferring the labels for updated digest (corresponding to the updated database) to the next circuit. For a formal description of our scheme for general RAM programs, see Section 6.

2.3 Multi-Hop Homomorphic Encryption for RAM Programs

Our model and problem — a bit more formally. We consider a scenario where a server S, holding an input x, publishes a public key pk and an encryption ct of x under pk. Now this ciphertext is passed on to a client Q that will compute a (possibly private) program P accessing memory D on the value encrypted in ct, obtaining another ciphertext ct'. Finally, we want that the server can use its secret key to recover $P^D(x)$ from the ciphertext ct' and $\widetilde{D}$, where $\widetilde{D}$ is an encrypted form of D that has been previously provided to S in a one-time setup phase. More generally, the computation could be performed by multiple clients $Q_1, \ldots, Q_n$. In this case, each client is required to place a pre-processed version of its database $\widetilde{D}_i$ with the server during setup. The computation itself could be performed in different sequences of the clients (for different extensions of the model, see Section 7.1). Examples of two such computation paths are shown in Figure 1.

For security, we want IND-CPA security for server's input x. For honest clients, we want *program-privacy* as well as *data-privacy*, i.e., the evaluation does not leak anything beyond the output of the computation even when the adversary corrupts the server and any subset of the clients. We note that data-privacy is rather easy to achieve via encryption and ORAM. Hence we focus on the challenges of achieving UMA security for honest clients, i.e., the adversary is allowed to learn the database D as well as memory access pattern of P on D.

UMA secure multi-hop scheme. We first build on the ideas from non-interactive secure computation for RAM programs. Every client first passes its database to the server. Then in every round, the server sends an OT message for input x. We assume for simplicity that every client has an up-to-date digest of its own database. Next, the first client Q_1 generates a garbled program for P_1, say ct_1 and sends it to Q_2. Here, the garbled program consists of t_1 (t_1 is the running time of P_1) garbled circuits accessing D_1 via laconic OT as described in the previous application. Now, Q_2 appends its garbled program for P_2 to the end of ct_1 and generates ct_2 consisting of ct_1 and new garbled program. Note that P_2 takes the output of P_1 as input and hence, the output keys of the

[9] We note that the above idea of using laconic OT also gives a conceptually very simple solution for UMA secure garbled RAM scheme [LO13]. Moreover, there is a general transformation [GHL+14] that converts any UMA secure garbled RAM into one with full security via the usage of symmetric key encryption and oblivious RAM. This would give a simplified construction of fully secure garbled RAM under DDH assumption.

last garbled circuit of P_1 have to be compatible with the input keys of the first garbled circuit of P_2 and so on. If we continue this procedure, after the last client Q_n, we get a sequence of garbled circuits where the first t_1 circuits access D_1, the next set accesses from D_2 and so on. Finally, the server S can evaluate the sequence of garbled circuits given $D_1, \ldots, D_n$. It is easy to see that correctness holds. But we have no security for clients.

The issue is similar to the issue pointed out by [GHV10] for the case of multi-hop garbled circuits. If the client Q_{i-1} colludes with the server, then they can learn both input labels for the garbled program of Q_i. To resolve this issue it is crucial that Q_i re-randomizes the garbled circuits provided by Q_{i-1}. For this we rely on re-randomizable garbled circuits provided by [GHV10], where given a garbled circuit anyone can re-garble it such that functionality of the original circuit is preserved while the re-randomized garbled circuit is unrecognizable even to the party who generated it. In our protocol we use re-randomizable garbled circuits but we stumble upon the following issue.

Recall that in the RAM application above, a garbled circuit outputs the laconic OT ciphertexts corresponding to the input keys of the next circuit. Hence, the input keys of the $(\tau+1)$-th circuit have to be hardwired inside the τ-th circuit. Since all of these circuits will be re-randomized for security, for correctness we require that we transform the hardwired keys in a manner consistent with the future re-randomization. But for security, Q_{i-1} does not know the randomness that will be used by Q_i.

Our first idea to resolve this issue is as follows: The circuits generated by Q_{i-1} will take additional inputs $s_i, \ldots s_n$ which are the randomness used by future parties for their re-randomization procedure. Since we are in the non-interactive setting, we cannot run an OT protocol between clients Q_{i-1} and later clients. We resolve this issue by putting the first message of OT for s_j in the public key of client Q_j and client Q_{i-1} will send the OT second messages along with ct_{i-1}. We do not want the clients' public keys to grow with the running time of the programs, hence, we think of s_j as PRF keys and each circuit re-randomization will invoke the PRF on a unique input.

The above approach causes a subtle issue in the security proof. Suppose, for simplicity, that client Q_i is the only honest client. When arguing security, we want to simulate all the garbled circuits in ct_i. To rely on the security of re-randomization, we need to replace the output of the PRF with key s_i with uniform random values but this key is fed as input to the circuits of the previous clients. We note that this is not a circularity issue but makes arguing security hard. We solve this issue as follows: Instead of feeding in PRF keys directly to the garbled circuits, we feed in corresponding outputs of the PRF. We generate the PRF output via a bunch of PRF circuits that take the PRF keys as input (see Figure 3). Now during simulation, we will first simulate these PRF circuits, followed by the simulation of the main circuits. We describe the scheme formally in Section 7.3.

3 Laconic Oblivious Transfer

In this section, we will introduce a primitive we call *Laconic OT* (or, ℓOT for short). We will start by describing laconic OT and then provide an extension of it to the notion of updatable laconic OT.

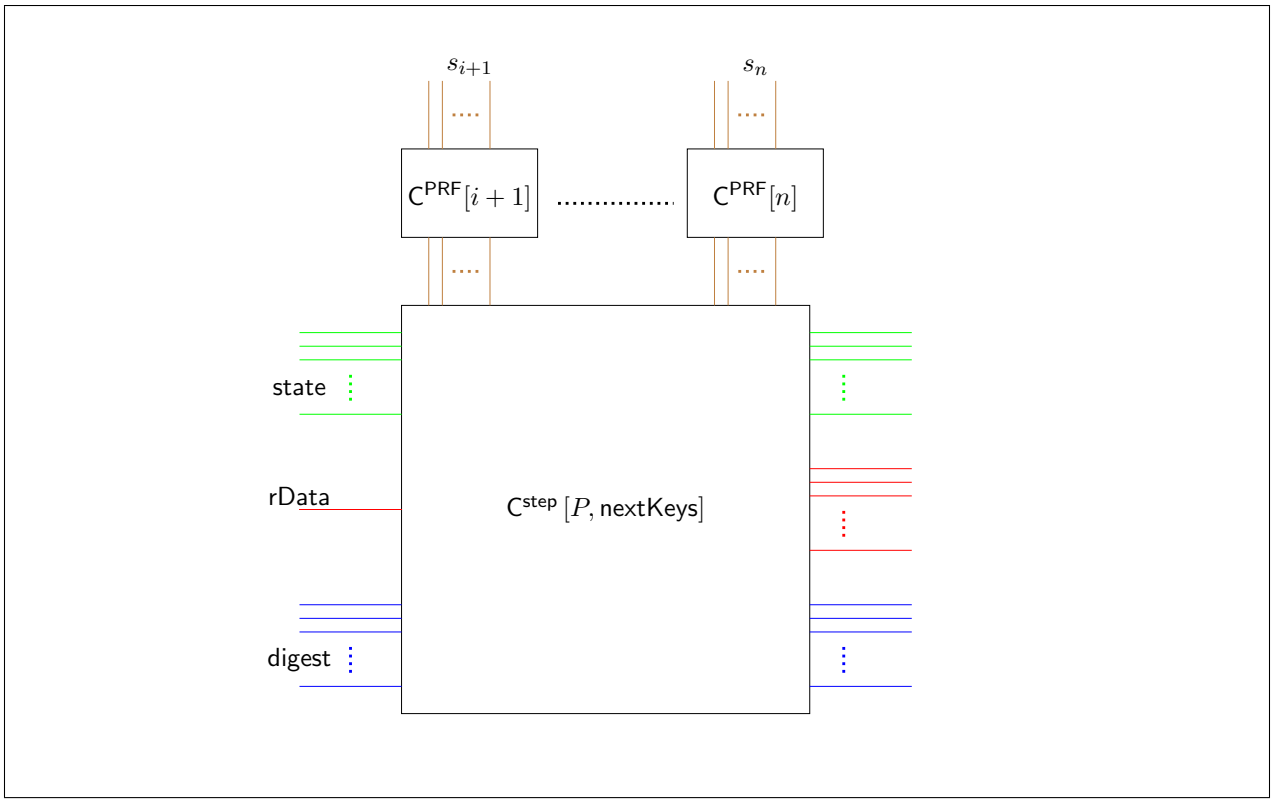


Figure 3: One step circuit for P_i along with the attached PRF circuits generated by Q_i.

3.1 Laconic OT

Definition 3.1 (Laconic OT)**.** *A laconic OT (ℓOT) scheme syntactically consists of four algorithms* crsGen*,* Hash*,* Send *and* $\mathsf{Receive}$*.*

- $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$*. It takes as input the security parameter* 1^λ *and outputs a common reference string* crs*.*
- $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$*. It takes as input a common reference string* crs *and a database* $D \in \{0,1\}^*$ *and outputs a digest* digest *of the database and a state* $\hat{\mathsf{D}}$*.*
- $\mathsf{e} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$*. It takes as input a common reference string* crs*, a digest* digest*, a database location* $L \in \mathbb{N}$ *and two messages* m_0 *and* m_1 *of length* λ*, and outputs a ciphertext* e*.*
- $m \leftarrow \mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L)$*. This is a RAM algorithm with random read access to* $\hat{\mathsf{D}}$*. It takes as input a common reference string* crs*, a ciphertext* e*, and a database location* $L \in \mathbb{N}$*. It outputs a message* m*.*

We require the following properties of an ℓOT scheme $(\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive})$*.*

- ***Correctness:*** *We require that it holds for any database* D *of size at most* $M = \mathsf{poly}(\lambda)$ *for any polynomial function* $\mathsf{poly}(\cdot)$*, any memory location* $L \in [M]$*, and any pair of messages*

$(m_0, m_1) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$ *that*

$$\Pr\left[m = m_{D[L]} \;\middle|\; \begin{array}{ccc} \mathsf{crs} & \leftarrow & \mathsf{crsGen}(1^\lambda) \\ (\mathsf{digest}, \hat{\mathsf{D}}) & \leftarrow & \mathsf{Hash}(\mathsf{crs}, D) \\ \mathsf{e} & \leftarrow & \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1) \\ m & \leftarrow & \mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L) \end{array} \right] = 1,$$

where the probability is taken over the random choices made by crsGen *and* Send*.*

- ***Sender Privacy Against Semi-Honest Receivers:*** *There exists a PPT simulator* $\ell\mathsf{OTSim}$ *such that the following holds. For any database* D *of size at most* $M = \mathsf{poly}(\lambda)$ *for any polynomial function* $\mathsf{poly}(\cdot)$*, any memory location* $L \in [M]$*, and any pair of messages* $(m_0, m_1) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$*, let* $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$ *and* $\mathsf{digest} \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$*. Then it holds that*

$$(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)) \stackrel{\mathrm{c}}{\approx} \left(\mathsf{crs}, \ell\mathsf{OTSim}(D, L, m_{D[L]})\right).$$

- ***Efficiency Requirement:*** *The length of* digest *is a fixed polynomial in* λ *independent of the size of the database; we will assume for simplicity that* $|\mathsf{digest}| = \lambda$*. Moreover, the algorithm* Hash *runs in time* $|D| \cdot \mathsf{poly}(\log |D|, \lambda)$*,* Send *and* Receive *run in time* $\mathsf{poly}(\log |D|, \lambda)$*.*

Receiver Privacy. In the above definition, we do not require receiver privacy as opposed to standard oblivious transfer, namely, no security guarantee is provided against a cheating (semi-honest) sender. This is mostly for ease of exposition. We would like to point out that adding receiver privacy (i.e., standard simulation based security against a semi-honest sender) to laconic OT can be done in a straightforward way. Instead of sending digest directly from the receiver to the sender and sending e back to the receiver, the two parties compute Send together via a two-round secure 2PC protocol, where the input of the receiver is digest and the input of the sender is (L, m_0, m_1), and only the receiver obtains the output e. This can be done using standard two-message OT and garbled circuits.

Multiple executions of Send **that share the same** digest**.** Notice that since the common reference string is public (i.e., not chosen by the simulator), the sender can involve Send function multiple times while still ensuring that security can be argued from the above definition (for the case of single execution) via a standard hybrid argument.

It will be convenient to use the following shorthand notations (generalizing the above notions) to run laconic OT for every single element in a database. Let $\mathsf{Keys} = ((\mathsf{Key}_{1,0}, \mathsf{Key}_{1,1}), \ldots, (\mathsf{Key}_{M,0}, \mathsf{Key}_{M,1}))$ be a list of $M = |D|$ key-pairs, where each key is of length λ. Then we will define

$$\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \mathsf{Keys}) = \left(\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, 1, \mathsf{Key}_{1,0}, \mathsf{Key}_{1,1}), \ldots, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, M, \mathsf{Key}_{M,0}, \mathsf{Key}_{M,1})\right).$$

Likewise, for a vector $\mathsf{e} = (\mathsf{e}_1, \ldots, \mathsf{e}_M)$ of ciphertexts define

$$\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}) = \left(\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}_1, 1), \ldots, \mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}_M, M)\right).$$

Similarly, let $\mathsf{Labels} = \mathsf{Keys}_D = (\mathsf{Key}_{1,D[1]}, \ldots, \mathsf{Key}_{M,D[M]})$, and define

$$\ell\mathsf{OTSim}(\mathsf{crs}, D, \mathsf{Labels}) = \left(\ell\mathsf{OTSim}(\mathsf{crs}, D, 1, \mathsf{Key}_{1,D[1]}), \ldots, \ell\mathsf{OTSim}(\mathsf{crs}, D, M, \mathsf{Key}_{M,D[M]}\right).$$

By the sender security for multiple executions, we have that

$$(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \mathsf{Keys})) \stackrel{\mathrm{c}}{\approx} (\mathsf{crs}, \ell\mathsf{OTSim}(\mathsf{crs}, D, \mathsf{Labels})).$$

3.2 Updatable Laconic OT

For our applications, we will need a version of laconic OT for which the receiver's short commitment digest to his database can be updated quickly (in time much smaller than the size of the database) when a bit of the database changes. We call this primitive supporting this functionality updatable laconic OT and define more formally below. At a high level, updatable laconic OT comes with an additional pair of algorithms $\mathsf{SendWrite}$ and $\mathsf{ReceiveWrite}$ which transfer the keys for an updated digest digest^* to the receiver. For convenience, we will define $\mathsf{ReceiveWrite}$ such that it also performs the write in $\hat{\mathsf{D}}$.

Definition 3.2 (Updatable Laconic OT)**.** *An updatable laconic OT (updatable ℓOT) scheme consists of algorithms* $\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive}$ *as per Definition 3.1 and additionally two algorithms* $\mathsf{SendWrite}$ *and* $\mathsf{ReceiveWrite}$ *with the following syntax.*

- $\mathsf{e_w} \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, \mathsf{digest}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}\right)$. *It takes as input the common reference string* crs*, a digest* digest*, a location $L \in \mathbb{N}$, a bit $b \in \{0,1\}$ to be written, and* $|\mathsf{digest}|$ *pairs of messages $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}$, where each $m_{j,c}$ is of length λ. And it outputs a ciphertext* $\mathsf{e_w}$*.*
- $\{m_j\}_{j=1}^{|\mathsf{digest}|} \leftarrow \mathsf{ReceiveWrite}^{\hat{\mathsf{D}}}(\mathsf{crs}, L, b, \mathsf{e_w})$. *This is a RAM algorithm with random read/write access to $\hat{\mathsf{D}}$. It takes as input the common reference string* crs*, a location L, a bit $b \in \{0,1\}$ and a ciphertext* $\mathsf{e_w}$*. It updates the state $\hat{\mathsf{D}}$ (such that $D[L] = b$) and outputs messages $\{m_j\}_{j=1}^{|\mathsf{digest}|}$.*

We require the following properties on top of properties of a laconic OT scheme.

- ***Correctness With Regard To Writes:*** *For any database D of size at most $M = \mathsf{poly}(\lambda)$ for any polynomial function $\mathsf{poly}(\cdot)$, any memory location $L \in [M]$, any bit $b \in \{0,1\}$, and any messages $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}$ of length λ, the following holds. Let D^* be identical to D, except that $D^*[L] = b$,*

$$\Pr\left[\begin{array}{l} m'_j = m_{j,\mathsf{digest}^*_j} \\ \forall j \in [|\mathsf{digest}|] \end{array} \;\middle|\; \begin{array}{rl} \mathsf{crs} & \leftarrow \mathsf{crsGen}(1^\lambda) \\ (\mathsf{digest}, \hat{\mathsf{D}}) & \leftarrow \mathsf{Hash}(\mathsf{crs}, D) \\ (\mathsf{digest}^*, \hat{\mathsf{D}}^*) & \leftarrow \mathsf{Hash}(\mathsf{crs}, D^*) \\ \mathsf{e_w} & \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, \mathsf{digest}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}\right) \\ \{m'_j\}_{j=1}^{|\mathsf{digest}|} & \leftarrow \mathsf{ReceiveWrite}^{\hat{\mathsf{D}}}(\mathsf{crs}, L, b, \mathsf{e_w}) \end{array}\right] = 1,$$

  where the probability is taken over the random choices made by crsGen *and* $\mathsf{SendWrite}$*. Furthermore, we require that the execution of* $\mathsf{ReceiveWrite}^{\hat{\mathsf{D}}}$ *above updates $\hat{\mathsf{D}}$ to $\hat{\mathsf{D}}^*$. (Note that* digest *is included in $\hat{\mathsf{D}}$, hence* digest *is also updated to* digest^**.)*

- ***Sender Privacy Against Semi-Honest Receivers With Regard To Writes:*** *There exists a PPT simulator* $\ell\mathsf{OTSimWrite}$ *such that the following holds. For any database D of size at most $M = \mathsf{poly}(\lambda)$ for any polynomial function $\mathsf{poly}(\cdot)$, any memory location $L \in [M]$, any bit $b \in \{0,1\}$, and any messages $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}$ of length λ, let* $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$*,*

$(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$, *and* $(\mathsf{digest}^*, \hat{\mathsf{D}}^*) \leftarrow \mathsf{Hash}(\mathsf{crs}, D^*)$, *where* D^* *is identical to* D *except that* $D^*[L] = b$. *Then it holds that*

$$\begin{pmatrix} \mathsf{crs}, \mathsf{SendWrite}(\mathsf{crs}, \mathsf{digest}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{digest}|}) \end{pmatrix} \\ \overset{c}{\approx} \quad \begin{pmatrix} \mathsf{crs}, \ell\mathsf{OTSimWrite}\left(\mathsf{crs}, D, L, b, \{m_{j,\mathsf{digest}_j^*}\}_{j\in[|\mathsf{digest}|]}\right) \end{pmatrix}.$$

- ***Efficiency Requirements:*** *We require that both* SendWrite *and* ReceiveWrite *run in time* $\mathsf{poly}(\log |D|, \lambda)$.

4 Laconic Oblivious Transfer with Factor-2 Compression

In this section, based on the DDH assumption we will construct a laconic OT scheme for which the hash function Hash compresses a database of length 2λ into a digest of length λ. We would refer to this primitive as laconic OT with factor-2 compression. We note that, subsequent to our work, the factor-2 compression construction has been simplified by Döttling and Garg [DG17] (another alternative simplification can be obtained using [AIKW13]). We refer the reader to [DG17] for the simpler construction and preserve the older construction here.

We will first construct the following two primitives as building blocks: (1) a somewhere statistically binding (SSB) hash function, and (2) a hash proof system that allows for proving knowledge of preimage bits for this SSB hash function. We will then present the ℓOT scheme with factor-2 compression in Section 4.4.

4.1 Somewhere Statistically Binding Hash Functions and Hash Proof Systems

In this section, we give definitions of somewhere statistically binding (SSB) hash functions [HW15] and hash proof systems [CS98]. For simplicity, we will only define SSB hash functions that compress 2λ values in the domain into λ bits. The more general definition works analogously.

Definition 4.1 (Somewhere Statistically Binding Hashing)**.** *An SSB hash function* SSBH *consists of three algorithms* crsGen, bindingCrsGen *and* Hash *with the following syntax.*

- $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$. *It takes the security parameter* λ *as input and outputs a common reference string* crs.
- $\mathsf{crs} \leftarrow \mathsf{bindingCrsGen}(1^\lambda, i)$. *It takes as input the security parameter* λ *and an index* $i \in [2\lambda]$, *and outputs a common reference string* crs.
- $y \leftarrow \mathsf{Hash}(\mathsf{crs}, x)$. *For some domain* $\mathfrak{D}$, *it takes as input a common reference string* crs *and a string* $x \in \mathfrak{D}^{2\lambda}$, *and outputs a string* $y \in \{0,1\}^\lambda$.

We require the following properties of an SSB hash function.

- ***Statistically Binding at Position*** i*: For every* $i \in [2\lambda]$ *and an overwhelming fraction of* crs *in the support of* $\mathsf{bindingCrsGen}(1^\lambda, i)$ *and every* $x \in \mathfrak{D}^{2\lambda}$, *we have that* $(\mathsf{crs}, \mathsf{Hash}(\mathsf{crs}, x))$ *uniquely determines* x_i. *More formally, for all* $x' \in \mathfrak{D}^{2\lambda}$ *such that* $x_i \neq x'_i$ *we have that* $\mathsf{Hash}(\mathsf{crs}, x') \neq \mathsf{Hash}(\mathsf{crs}, x)$.

- ***Index Hiding***: *It holds for all* $i \in [2\lambda]$ *that* $\mathsf{crsGen}(1^\lambda) \overset{c}{\approx} \mathsf{bindingCrsGen}(1^\lambda, i)$*, i.e., common reference strings generated by* crsGen *and* bindingCrsGen *are computationally indistinguishable.*

Next, we define hash proof systems [CS98] that are designated verifier proof systems that allow for proving that the given problem instance in some language. We give the formal definition as follows.

Definition 4.2 (Hash Proof System)**.** *Let* $\mathcal{L}_z \subseteq \mathcal{M}_z$ *be an NP-language residing in a universe* $\mathcal{M}_z$*, both parametrized by some parameter* z*. Moreover, let* $\mathcal{L}_z$ *be characterized by an efficiently computable witness-relation* $\mathcal{R}$*, namely, for all* $x \in M_z$ *it holds that* $x \in \mathcal{L}_z \Leftrightarrow \exists w : \mathcal{R}(x, w) = 1$*. A hash proof system* HPS *for* $\mathcal{L}_z$ *consists of three algorithms* KeyGen*,* $\mathsf{H}_{\mathsf{public}}$ *and* $\mathsf{H}_{\mathsf{secret}}$ *with the following syntax.*

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda, z)$*: Takes as input the security parameter* λ *and a parameter* z*, and outputs a public-key and secret key pair* $(\mathsf{pk}, \mathsf{sk})$*.*
- $y \leftarrow \mathsf{H}_{\mathsf{public}}(\mathsf{pk}, x, w)$*: Takes as input a public key* pk*, an instance* $x \in \mathcal{L}_z$*, and a witness* w*, and outputs a value* y*.*
- $y \leftarrow \mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, x)$*: Takes as input a secret key* sk *and an instance* $x \in M_z$*, and outputs a value* y*.*

We require the following properties of a hash proof system.

- ***Perfect Completeness***: *For every* z*, every* $(\mathsf{pk}, \mathsf{sk})$ *in the support of* $\mathsf{KeyGen}(1^\lambda, z)$*, and every* $x \in \mathcal{L}_z$ *with witness* w *(i.e.,* $\mathcal{R}(x, w) = 1$*), it holds that*

$$\mathsf{H}_{\mathsf{public}}(\mathsf{pk}, x, w) = \mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, x).$$

- ***Perfect Soundness***: *For every* z *and every* $x \in M_z \setminus \mathcal{L}_z$*, let* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda, z)$*, then it holds that*

$$(z, \mathsf{pk}, \mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, x)) \equiv (z, \mathsf{pk}, u),$$

  where u *is distributed uniformly random in the range of* $\mathsf{H}_{\mathsf{secret}}$*. Here,* $\equiv$ *denotes distributional equivalence.*

4.2 HPS-friendly SSB Hashing

In this section, we will construct an HPS-friendly SSB hash function that supports a hash proof system. In particular, there is a hash proof system that enables proving that a certain bit of the pre-image of a hash-value has a certain fixed value (in our case, either 0 or 1).

We start with some notations. Let $(\mathbb{G}, \cdot)$ be a cyclic group of order p with generator g. Let $\mathbf{M} \in \mathbb{Z}_p^{m \times n}$ be a matrix. We will denote by $\hat{\mathbf{M}} = g^{\mathbf{M}} \in \mathbb{G}^{m \times n}$ the element-wise exponentiation of g with the elements of $\mathbf{M}$. We also define $\hat{\mathbf{L}} = \hat{\mathbf{H}}^{\mathbf{M}} \in \mathbb{G}^{m \times k}$, where $\hat{\mathbf{H}} \in \mathbb{G}^{m \times n}$ and $\mathbf{M} \in \mathbb{Z}_p^{n \times k}$ as follows: Each element $\hat{\mathbf{L}}_{i,j} = \prod_{k=1}^{n} \hat{\mathbf{H}}_{i,k}^{\mathbf{M}_{k,j}}$ (intuitively this operation corresponds to matrix multiplication in the exponent). This is well-defined and efficiently computable.

Computational Assumptions. In the following, we first define the computational problems on which we will base the security of our HPS-friendly SSB hash function.

Definition 4.3 (The Decisional Diffie-Hellman (DDH) Problem)**.** *Let $(\mathbb{G}, \cdot)$ be a cyclic group of prime order p and with generator g. Let a, b, c be sampled uniformly at random from $\mathbb{Z}_p$ (i.e., $a, b, c \xleftarrow{\$} \mathbb{Z}_p$). The DDH problem asks to distinguish the distributions (g, g^a, g^b, g^{ab}) and (g, g^a, g^b, g^c).*

Definition 4.4 (Matrix Rank Problem)**.** *Let m, n be integers and let $\mathbb{Z}_p^{m\times n;r}$ be the set of all $m \times n$ matrices over $\mathbb{Z}_p$ with rank r. Further, let $1 \leq r_1 < r_2 \leq \min(m, n)$. The goal of the matrix rank problem, denoted as $\mathsf{MatrixRank}(\mathbb{G}, m, n, r_1, r_2)$, is to distinguish the distributions $g^{\mathbf{M}_1}$ and $g^{\mathbf{M}_2}$, where $\mathbf{M}_1 \xleftarrow{\$} \mathbb{Z}_p^{m\times n;r_1}$ and $\mathbf{M}_2 \xleftarrow{\$} \mathbb{Z}_p^{m\times n;r_2}$.*

In a recent result by Villar [Vil12] it was shown that the matrix rank problem can be reduced almost tightly to the DDH problem.

Theorem 4.5 ([Vil12] Theorem 1, simplified)**.** *Assume there exists a PPT distinguisher $\mathcal{D}$ that solves $\mathsf{MatrixRank}(\mathbb{G}, m, n, r_1, r_2)$ problem with advantage ϵ. Then, there exists a PPT distinguisher $\mathcal{D}'$ (running in almost time as $\mathcal{D}$) that solves DDH problem over $\mathbb{G}$ with advantage at least $\frac{\epsilon}{\lceil \log_2(r_2/r_1) \rceil}$.*

We next give the construction of an HPS-friendly SSB hash function.

Construction. Our construction builds on the scheme of Okamoto et al. [OPWW15]. We will not delve into the details of their scheme and directly jump into our construction.

Let n be an integer such that $n = 2\lambda$, and let $(\mathbb{G}, \cdot)$ be a cyclic group of order p and with generator g. Let $\mathbf{T}_i \in \mathbb{Z}_p^{2\times n}$ be a matrix which is zero everywhere except the i-th column, and the i-th column is equal to $\mathbf{t} = (0, 1)^\top$. The three algorithms of the SSB hash function are defined as follows.

- $\mathsf{crsGen}(1^\lambda)$: Pick a uniformly random matrix $\mathbf{H} \xleftarrow{\$} \mathbb{Z}_p^{2\times n}$ and output $\hat{\mathbf{H}} = g^{\mathbf{H}}$.
- $\mathsf{bindingCrsGen}(1^\lambda, i)$: Pick a uniformly random vector $(w_1, w_2)^\top = \mathbf{w} \xleftarrow{\$} \mathbb{Z}_p^2$ with the restriction that $w_1 = 1$, pick a uniformly random vector $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_p^n$ and set $\mathbf{A} \leftarrow \mathbf{w} \cdot \mathbf{a}^\top$. Set $\mathbf{H} \leftarrow \mathbf{T}_i + \mathbf{A}$ and output $\hat{\mathbf{H}} = g^{\mathbf{H}}$.
- $\mathsf{Hash}(\mathsf{crs}, \mathbf{x})$: Parse $\mathbf{x}$ as a vector in $\mathfrak{D}^n$ ($\mathfrak{D} = \mathbb{Z}_p$) and parse $\mathsf{crs} = \hat{\mathbf{H}}$. Compute $\mathbf{y} \in \mathbb{G}^2$ as $\mathbf{y} = \hat{\mathbf{H}}^{\mathbf{x}}$. Parse $\mathbf{y}$ as a binary string and output the result.

Compression. Notice that we can get factor two compression for an input space $\{0, 1\}^{2\lambda}$ by restricting the domain to $\mathfrak{D}' = \{0, 1\} \subset \mathfrak{D}$. The input length $n = 2\lambda$, where λ is set to be twice the number of bits in the bit representation of a group element in $\mathbb{G}$. In the following we will assume that $n = 2\lambda$ and that the bit-representation size of a group element in $\mathbb{G}$ is $\frac{\lambda}{2}$.

We will first show that the distributions $\mathsf{crsGen}(1^\lambda)$ and $\mathsf{bindingCrsGen}(1^\lambda, i)$ are computationally indistinguishable for every index $i \in [n]$, given that the DDH problem is computationally hard in the group $\mathbb{G}$.

Lemma 4.6 (Index Hiding)**.** *Assume that the* $\mathsf{MatrixRank}(\mathbb{G}, 2, n, 1, 2)$ *problem is hard. Then the distributions* $\mathsf{crsGen}(1^\lambda)$ *and* $\mathsf{bindingCrsGen}(1^\lambda, i)$ *are computationally indistinguishable, for every* $i \in [n]$.

Proof. Assume there exists a PPT distinguisher $\mathcal{D}$ that distinguishes the distributions $\mathsf{crsGen}(1^\lambda)$ and $\mathsf{bindingCrsGen}(1^\lambda, i)$ with non-negligible advantage ϵ. We will construct a PPT distinguisher $\mathcal{D}'$ that distinguishes $\mathsf{MatrixRank}(\mathbb{G}, 2, n, 1, 2)$ with non-negligible advantage.

The distinguisher $\mathcal{D}'$ does the following on input $\hat{\mathbf{M}} \in \mathbb{G}^{2\times n}$. It computes $\hat{\mathbf{H}} \in \mathbb{G}^{2\times n}$ as element-wise multiplication of $\hat{\mathbf{M}}$ and $g^{\mathbf{T}_i}$ and runs $\mathcal{D}$ on $\hat{\mathbf{H}}$. If $\mathcal{D}$ outputs crsGen, then $\mathcal{D}'$ outputs rank 2, otherwise $\mathcal{D}'$ outputs rank 1.

We will now show that $\mathcal{D}'$ also has non-negligible advantage. Write $\mathcal{D}'$'s input as $\hat{\mathbf{M}} = g^{\mathbf{M}}$. If $\mathbf{M}$ is chosen uniformly random with rank 2, then $\mathbf{M}$ is uniform in $\mathbb{Z}_p^{2\times n}$ with overwhelming probability. Hence with overwhelming probability, $\mathbf{M} + \mathbf{T}_i$ is also distributed uniformly random and it follows that $\hat{\mathbf{H}} = g^{\mathbf{M}+\mathbf{T}_i}$ is uniformly random in $\mathbb{G}^{2\times n}$ which is identical to the distribution generated by $\mathsf{crsGen}(1^\lambda)$. On the other hand, if $\mathbf{M}$ is chosen uniformly random with rank 1, then there exists a vector $\mathbf{w} \in \mathbb{Z}_p^2$ such that each column of $\mathbf{M}$ can be written as $a_i \cdot \mathbf{w}$. We can assume that the first element w_1 of $\mathbf{w}$ is 1, since the case $w_1 = 0$ happens only with probability $1/p = \mathsf{negl}(\lambda)$ and if $w_1 \neq 0$ we can replace all a_i by $a_i' = a_i \cdot w_1$ and replace w_i by $w_i' = \frac{w_i}{w_1}$. Thus, we can write $\mathbf{M}$ as $\mathbf{M} = \mathbf{w} \cdot \mathbf{a}^\top$ and consequently $\hat{\mathbf{H}}$ as $\hat{\mathbf{H}} = g^{\mathbf{w}\cdot\mathbf{a}^\top+\mathbf{T}_i}$. Notice that $\mathbf{a}$ is uniformly distributed, hence $\hat{\mathbf{H}}$ is identical to the distribution generated by $\mathsf{bindingCrsGen}(1^\lambda, i)$. Since $\mathcal{D}$ can distinguish the distributions $\mathsf{crsGen}(1^\lambda)$ and $\mathsf{bindingCrsGen}(1^\lambda, i)$ with non-negligible advantage ϵ, $\mathcal{D}'$ can distinguish $\mathsf{MatrixRank}(\mathbb{G}, 2, n, 1, 2)$ with advantage $\epsilon - \mathsf{negl}(\lambda)$, which contradicts the hardness of $\mathsf{MatrixRank}(\mathbb{G}, 2, n, 1, 2)$. □

A corollary of Lemma 4.6 is that for all $i, j \in [n]$ the distributions $\mathsf{bindingCrsGen}(1^\lambda, i)$ and $\mathsf{bindingCrsGen}(1^\lambda, j)$ are indistinguishable, stated as follows.

Corollary 4.7. *Assume the* $\mathsf{MatrixRank}(\mathbb{G}, 2, n, 1, 2)$ *problem is computationally hard. Then it holds for all* $i, j \in [n]$ *that* $\mathsf{bindingCrsGen}(1^\lambda, i)$ *and* $\mathsf{bindingCrsGen}(1^\lambda, j)$ *are computationally indistinguishable.*

We next show that if the common reference string $\mathsf{crs} = \hat{\mathbf{H}}$ is generated by $\mathsf{bindingCrsGen}(1^\lambda, i)$, then the hash value $\mathsf{Hash}(\mathsf{crs}, \mathbf{x})$ is statistically binded to x_i.

Lemma 4.8 (Statistically Binding at Position i)**.** *For every* $i \in [n]$*, every* $\mathbf{x} \in \mathbb{Z}_p^n$*, and all choices of* crs *in the support of* $\mathsf{bindingCrsGen}(1^\lambda, i)$ *we have that for every* $\mathbf{x}' \in \mathbb{Z}_p^n$ *such that* $x_i' \neq x_i$*,* $\mathsf{Hash}(\mathsf{crs}, \mathbf{x}) \neq \mathsf{Hash}(\mathsf{crs}, \mathbf{x}')$.

Proof. We first write crs as $\hat{\mathbf{H}} = g^{\mathbf{H}} = g^{\mathbf{w}\cdot\mathbf{a}^\top+\mathbf{T}_i}$ and $\mathsf{Hash}(\mathsf{crs}, \mathbf{x})$ as $\mathsf{Hash}(\hat{\mathbf{H}}, \mathbf{x}) = g^{\mathbf{y}} = g^{\mathbf{H}\cdot\mathbf{x}}$. Thus, by taking the discrete logarithm with basis g our task is to demonstrate that there exists a unique x_i from $\mathbf{H} = \mathbf{w} \cdot \mathbf{a}^\top + \mathbf{T}_i$ and $\mathbf{y} = \mathbf{H} \cdot \mathbf{x}$. Observe that

$$\begin{aligned}\mathbf{y} = \mathbf{H} \cdot \mathbf{x} &= (\mathbf{w} \cdot \mathbf{a}^\top + \mathbf{T}_i) \cdot \mathbf{x} = \mathbf{w} \cdot \langle \mathbf{a}, \mathbf{x} \rangle + \mathbf{T}_i \cdot \mathbf{x} \\ &= \begin{pmatrix} 1 \\ w_2 \end{pmatrix} \cdot \langle \mathbf{a}, \mathbf{x} \rangle + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \cdot x_i,\end{aligned}$$

where $\langle \mathbf{a}, \mathbf{x} \rangle$ is the inner product of $\mathbf{a}$ and $\mathbf{x}$. If $\mathbf{a} \neq \mathbf{0}$, then we can use any non-zero element of $\mathbf{a}$ to compute w_2 from $\mathbf{H}$, and recover x_i by computing $x_i = y_2 - w_2 \cdot y_1$; otherwise $\mathbf{a} = \mathbf{0}$, so $x_i = y_2$. □

4.3 A Hash Proof System for Knowledge of Preimage Bits

In this section, we give our desired hash proof systems. In particular, we need a hash proof system for membership in a subspace of a vector space. In our proof we need the following technical lemma.

Lemma 4.9. *Let* $\mathbf{M} \in \mathbb{Z}_p^{m \times n}$ *be a matrix. Let* $\mathsf{colsp}(\mathbf{M}) = \{\mathbf{M} \cdot \mathbf{x} \mid \mathbf{x} \in \mathbb{Z}_p^n\}$ *be its column space, and* $\mathsf{rowsp}(\mathbf{M}) = \{\mathbf{x}^\top \cdot \mathbf{M} \mid \mathbf{x} \in \mathbb{Z}_p^m\}$ *be its row space. Assume that* $\mathbf{y} \in \mathbb{Z}_p^m$ *and* $\mathbf{y} \notin \mathsf{colsp}(\mathbf{M})$*. Let* $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_p^m$ *be chosen uniformly at random. Then it holds that*

$$(\mathbf{M}, \mathbf{y}, \mathbf{r}^\top \mathbf{M}, \mathbf{r}^\top \mathbf{y}) \equiv (\mathbf{M}, \mathbf{y}, \mathbf{r}^\top \mathbf{M}, u),$$

where $u \xleftarrow{\$} \mathbb{Z}_p$ *is distributed uniformly and independently of* $\mathbf{r}$*. Here,* $\equiv$ *denotes distributional equivalence.*

Proof. For any $\mathbf{t} \in \mathsf{rowsp}(\mathbf{M})$ and $s \in \mathbb{Z}_p$, consider following linear equation system

$$\begin{cases} \mathbf{r}^\top \mathbf{M} = \mathbf{t} \\ \mathbf{r}^\top \mathbf{y} = s \end{cases}.$$

Let $\mathcal{N}$ be the left null space of $\mathbf{M}$. We know that $\mathbf{y} \notin \mathsf{colsp}(\mathbf{M})$, hence $\mathbf{M}$ has rank $\leq m-1$, therefore $\mathcal{N}$ has dimension ≥ 1. Let $\mathbf{r}_0$ be an arbitrary solution for $\mathbf{r}^\top \mathbf{M} = \mathbf{t}$, and let $\mathbf{n}$ be a vector in $\mathcal{N}$ such that $\mathbf{n}^\top \mathbf{y} \neq \mathbf{0}$ (there must be such a vector since $\mathbf{y} \notin \mathsf{colsp}(\mathbf{M})$). Then there exists a solution $\mathbf{r}$ for the above linear equation system, that is,

$$\mathbf{r} = \mathbf{r}_0 + (\mathbf{n}^\top \mathbf{y})^{-1} \cdot (s - \mathbf{r}_0^\top \mathbf{y}) \cdot \mathbf{n},$$

where $(\mathbf{n}^\top \mathbf{y})^{-1}$ is the multiplicative inverse of $\mathbf{n}^\top \mathbf{y}$ in $\mathbb{Z}_p$. Then two cases arise: (i) column vectors of $(\mathbf{M}\ \mathbf{y})$ are full-rank, or (ii) not. In this first case, there is a unique solution for $\mathbf{r}$. In the second case the solution space has the same size as the left null space of $(\mathbf{M}\ \mathbf{y})$. Therefore, in both cases, the number of solutions for $\mathbf{r}$ is the same for every $(\mathbf{t}, s)$ pair.

As $\mathbf{r}$ is chosen uniformly at random, all pairs $(\mathbf{t}, s) \in \mathsf{rowsp}(\mathbf{M}) \times \mathbb{Z}_p$ have the same probability of occurrence and the claim follows. □

Construction. Fix a matrix $\hat{\mathbf{H}} \in \mathbb{G}^{2 \times n}$ and an index $i \in [n]$. We will construct a hash proof system $\mathsf{HPS} = (\mathsf{KeyGen}, \mathsf{H}_{\mathsf{public}}, \mathsf{H}_{\mathsf{secret}})$ for the following language $\mathcal{L}_{\hat{\mathbf{H}},i}$:

$$\mathcal{L}_{\hat{\mathbf{H}},i} = \{(\hat{\mathbf{y}}, b) \in \mathbb{G}^2 \times \{0,1\} \mid \exists \mathbf{x} \in \mathbb{Z}_p^n \text{ s.t. } \hat{\mathbf{y}} = \hat{\mathbf{H}}^{\mathbf{x}} \text{ and } x_i = b\}.$$

Note that in our hash proof system we only enforce that a single specified bit is b, where $b \in \{0,1\}$. However, our hash proof system does not place any requirement on the value used at any of the other locations. In fact the values used at the other locations may actually be from the full domain $\mathfrak{D}$ (i.e., $\mathbb{Z}_p$). Observe that the formal definition of the language $\mathcal{L}_{\hat{\mathbf{H}},i}$ above incorporates this difference in how the honest computation of the hash function is performed and what the hash proof system is supposed to prove.

For ease of exposition, it will be convenient to work with a matrix $\hat{\mathbf{H}}' \in \mathbb{G}_p^{3 \times n}$:

$$\hat{\mathbf{H}}' = \begin{pmatrix} \hat{\mathbf{H}} \\ g^{\mathbf{e}_i^\top} \end{pmatrix},$$

where $\mathbf{e}_i \in \mathbb{Z}_p^n$ is the i-th unit vector, with all elements equal to zero except the i^{th} one which is equal to one.

- $\mathsf{KeyGen}(1^\lambda, (\hat{\mathbf{H}}, i))$: Choose $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_p^3$ uniformly at random. Compute $\hat{\mathbf{h}} = \left((\hat{\mathbf{H}}')^\top\right)^{\mathbf{r}}$. Set $\mathsf{pk} = \hat{\mathbf{h}}$ and $\mathsf{sk} = \mathbf{r}$. Output $(\mathsf{pk}, \mathsf{sk})$.
- $\mathsf{H}_{\mathsf{public}}(\mathsf{pk}, (\hat{\mathbf{y}}, b), \mathbf{x})$: Parse pk as $\hat{\mathbf{h}}$. Compute $\hat{z} = (\hat{\mathbf{h}}^\top)^{\mathbf{x}}$ and output $\hat{z}$.
- $\mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, (\hat{\mathbf{y}}, b))$: Parse sk as $\mathbf{r}$ and set $\hat{\mathbf{y}}' = \begin{pmatrix} \hat{\mathbf{y}} \\ g^b \end{pmatrix}$. Compute $\hat{z} = ((\hat{\mathbf{y}}')^\top)^{\mathbf{r}}$ and output $\hat{z}$.

Lemma 4.10. *For every matrix $\hat{\mathbf{H}} \in \mathbb{G}^{2\times n}$ and every $i \in [n]$, HPS is a hash proof system for the language $\mathcal{L}_{\hat{\mathbf{H}},i}$.*

Proof. Let $\hat{\mathbf{H}} = g^{\mathbf{H}}$, $\mathbf{r} = (\mathbf{r}^*, r_3)$ where $\mathbf{r}^* \in \mathbb{Z}_p^2$. Let $\mathbf{y}' := \log_g \hat{\mathbf{y}}'$, $\mathbf{y} := \log_g \hat{\mathbf{y}}$, $\mathbf{H}' := \log_g \hat{\mathbf{H}}'$, $\mathbf{h} := \log_g \hat{\mathbf{h}}$.

For perfect correctness, we need to show that for every $i \in [n]$, every $\hat{\mathbf{H}} \in \mathbb{G}^{2\times n}$, and every $(\mathsf{pk}, \mathsf{sk})$ in the support of $\mathsf{KeyGen}(1^\lambda, (\hat{\mathbf{H}}', i))$, if $(\hat{\mathbf{y}}, b) \in \mathcal{L}_{\hat{\mathbf{H}},i}$ and $\mathbf{x}$ is a witness for membership (i.e., $\hat{\mathbf{y}} = \hat{\mathbf{H}}^{\mathbf{x}}$ and $x_i = b$), then it holds that $\mathsf{H}_{\mathsf{public}}(\mathsf{pk}, (\hat{\mathbf{y}}, b), \mathbf{x}) = \mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, (\hat{\mathbf{y}}, b))$.

To simplify the argument, we again consider the statement under the discrete logarithm with basis g. Then it holds that

$$\begin{aligned}
&\log_g \left(\mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, (\hat{\mathbf{y}}, b))\right) \\
=&\log_g \left(\left((\hat{\mathbf{y}}')^\top\right)^{\mathbf{r}}\right) = \langle \mathbf{y}', \mathbf{r}\rangle = \langle \mathbf{y}, \mathbf{r}^*\rangle + b \cdot r_3 \\
=&\langle \mathbf{H}\cdot\mathbf{x}, \mathbf{r}^*\rangle + x_i \cdot r_3 = \langle \mathbf{H}'\mathbf{x}, \mathbf{r}\rangle = \langle (\mathbf{H}')^\top \mathbf{r}, \mathbf{x}\rangle \\
=&\langle \mathbf{h}, \mathbf{x}\rangle = \log_g \left((\hat{\mathbf{h}}^\top)^{\mathbf{x}}\right) \\
=&\log_g \left(\mathsf{H}_{\mathsf{public}}(\mathsf{pk}, (\hat{\mathbf{y}}, b), \mathbf{x})\right).
\end{aligned}$$

For perfect soundness, let $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda, (\hat{\mathbf{H}}', i))$. We will show that if $(\hat{\mathbf{y}}, b) \notin \mathcal{L}_{\hat{\mathbf{H}},i}$, then $\mathsf{H}_{\mathsf{secret}}(\mathsf{sk}, (\hat{\mathbf{y}}, b))$ is distributed uniformly random in the range of $\mathsf{H}_{\mathsf{secret}}$, even given $\hat{\mathbf{H}}$, i, and pk. Again under the discrete logarithm, this is equivalent to showing that $\langle \mathbf{y}', \mathbf{r}\rangle$ is distributed uniformly random given $\mathbf{H}'$ and $\mathbf{h} = (\mathbf{H}')^\top \mathbf{r}$.

Note that we can re-write the language $\mathcal{L}_{\hat{\mathbf{H}},i} = \{(\hat{\mathbf{y}}, b) \in \mathbb{G}^2 \times \mathbb{Z}_p \mid \exists \mathbf{x} \in \mathbb{Z}_p^n \text{ s.t. } \mathbf{H}'\mathbf{x} = \mathbf{y}'\}$. It follows that if $(\hat{\mathbf{y}}, b) \notin \mathcal{L}_{\hat{\mathbf{H}},i}$, then $\mathbf{y}' \notin \mathsf{span}(\mathbf{H}')$. Now it follows directly from Lemma 4.9 that

$$\mathbf{r}^\top \mathbf{y}' \equiv u$$

given $\mathbf{H}'$ and $\mathbf{r}^\top \mathbf{H}'$, where u is distributed uniformly random. This concludes the proof. □

Remark 4.11. *While proving the security of our applications based on the above hash-proof system, we would generate $\hat{\mathbf{H}}$ to be the output of $\mathsf{bindingCrsGen}(1^\lambda, i)$ and use the property that if $(\hat{\mathbf{y}}, b) \in \mathcal{L}_{\hat{\mathbf{H}},i}$, then $(\hat{\mathbf{y}}, (1-b)) \notin \mathcal{L}_{\hat{\mathbf{H}},i}$. This follows directly from Lemma 4.8 (that is, $\hat{\mathbf{H}}$ and $\hat{\mathbf{y}}$ uniquely fixes x_i).*

4.4 The Laconic OT Scheme

We are now ready to put the pieces together and provide our ℓOT scheme with factor-2 compression.

Construction. Let $\mathsf{SSBH} = (\mathsf{SSBH.crsGen}, \mathsf{SSBH.bindingCrsGen}, \mathsf{SSBH.Hash})$ be the HPS-friendly SSB hash function constructed in Section 4.2 with domain $\mathfrak{D} = \mathbb{Z}_p$. Notice that we achieve factor-2 compression (namely, compressing 2λ bits into λ bits) by restricting the domain from $\mathfrak{D}^n$ to $\{0,1\}^n$ in our laconic OT scheme. Also, abstractly let the associated hash proof system be $\mathsf{HPS} = (\mathsf{HPS.KeyGen}, \mathsf{HPS.H_{public}}, \mathsf{HPS.H_{secret}})$ for the language

$$\mathcal{L}_{\mathsf{crs},i} = \{(\mathsf{digest}, b) \in \{0,1\}^\lambda \times \{0,1\} \mid \exists D \in \mathfrak{D}^{2\lambda} : \mathsf{SSBH.Hash}(\mathsf{crs}, D) = \mathsf{digest} \text{ and } D[i] = b\}.$$

Recall that the bit-representation size of a group element of $\mathbb{G}$ is $\frac{\lambda}{2}$, hence the language defined above is the same as the one defined in Section 4.3.

Now we construct the laconic OT scheme $\ell OT = (\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive})$ as follows.

- $\mathsf{crsGen}(1^\lambda)$: Compute $\mathsf{crs} \leftarrow \mathsf{SSBH.crsGen}(1^\lambda)$ and output crs.
- $\mathsf{Hash}(\mathsf{crs}, D \in \{0,1\}^{2\lambda})$:
  $\mathsf{digest} \leftarrow \mathsf{SSBH.Hash}(\mathsf{crs}, D)$
  $\hat{\mathsf{D}} \leftarrow (D, \mathsf{digest})$
  Output $(\mathsf{digest}, \hat{\mathsf{D}})$
- $\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$:
  Let HPS be the hash-proof system for the language $\mathcal{L}_{\mathsf{crs},L}$
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{HPS.KeyGen}(1^\lambda, (\mathsf{crs}, L))$
  $c_0 \leftarrow m_0 \oplus \mathsf{HPS.H_{secret}}(\mathsf{sk}, (\mathsf{digest}, 0))$
  $c_1 \leftarrow m_1 \oplus \mathsf{HPS.H_{secret}}(\mathsf{sk}, (\mathsf{digest}, 1))$
  Output $\mathsf{e} = (\mathsf{pk}, c_0, c_1)$
- $\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L)$:
  Parse $\mathsf{e} = (\mathsf{pk}, c_0, c_1)$
  Parse $\hat{\mathsf{D}} = (D, \mathsf{digest})$, and set $b \leftarrow D[L]$.
  $m \leftarrow c_b \oplus \mathsf{HPS.H_{public}}(\mathsf{pk}, (\mathsf{digest}, b), D)$
  Output m

We will now show that ℓOT is a laconic OT protocol with factor-2 compression, i.e., it has compression factor 2, and satisfies the correctness and sender privacy requirements. First notice that $\mathsf{SSBH.Hash}$ is factor-2 compressing, so Hash also has compression factor 2. We next argue correctness and sender privacy in Lemmas 4.12 and 4.13, respectively.

Lemma 4.12. *Given that* HPS *satisfies the correctness property, the* ℓOT *scheme also satisfies the correctness property.*

Proof. Fix a common reference string crs in the support of $\mathsf{crsGen}(1^\lambda)$, a database string $D \in \{0,1\}^{2\lambda}$ and an index $L \in [2\lambda]$. For any crs, D, L such that $D[L] = b$, let $\mathsf{digest} = \mathsf{Hash}(\mathsf{crs}, D)$. Then it clearly holds that $(\mathsf{digest}, b) \in \mathcal{L}_{\mathsf{crs},L}$. Thus, by the correctness property of the hash proof system HPS it holds that

$$\mathsf{HPS.H_{secret}}(\mathsf{sk}, (\mathsf{digest}, b)) = \mathsf{HPS.H_{public}}(\mathsf{pk}, (\mathsf{digest}, b), D).$$

By the construction of $\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$, $c_b = m_b \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, b))$. Hence the output m of $\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}, L)$ is

$$\begin{aligned} m =& c_b \oplus \mathsf{HPS.H}_{\mathsf{public}}(\mathsf{pk}, (\mathsf{digest}, b), D) \\ =& m_b \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, b)) \oplus \mathsf{HPS.H}_{\mathsf{public}}(\mathsf{pk}, (\mathsf{digest}, b), D) \\ =& m_b. \end{aligned}$$

□

Lemma 4.13. *Given that* SSBH *is index-hiding and has the statistically binding property and that* HPS *is sound, then the ℓOT scheme satisfies sender privacy against semi-honest receiver.*

Proof. We first construct the simulator $\ell\mathsf{OTSim}$.

$\ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]})$:
  $\mathsf{digest} \leftarrow \mathsf{SSBH.Hash}(\mathsf{crs}, D)$
  Let HPS be the hash-proof system for the language $\mathcal{L}_{\mathsf{crs},L}$
  $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{HPS.KeyGen}(1^\lambda, (\mathsf{crs}, L))$
  $c_0 \leftarrow m_{D[L]} \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, 0))$
  $c_1 \leftarrow m_{D[L]} \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, 1))$
  Output (pk, c_0, c_1)

For any database D of size at most $M = \mathsf{poly}(\lambda)$ for any polynomial function $\mathsf{poly}(\cdot)$, any memory location $L \in [M]$, and any pair of messages $(m_0, m_1) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$, let $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$ and $\mathsf{digest} \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$. Then we will prove that the two distributions $(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1))$ and $(\mathsf{crs}, \ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]}))$ are computationally indistinguishable. Consider the following hybrids.

- Hybrid 0: This is the real experiment, namely $(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1))$.
- Hybrid 1: Same as hybrid 0, except that crs is computed by $\mathsf{crs} \leftarrow \mathsf{SSBH.bindingCrsGen}(1^\lambda, L)$.
- Hybrid 2: Same as hybrid 1, except that $c_{1-D[L]}$ is computed by $c_{1-D[L]} \leftarrow m_{D[L]} \oplus \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, 1 - D[L]))$. That is, both c_0 and c_1 encrypt the same message $m_{D[L]}$.
- Hybrid 3: Same as hybrid 2, except that crs is computed by $\mathsf{crs} \leftarrow \mathsf{SSBH.crsGen}(1^\lambda)$. This is the simulated experiment, namely $(\mathsf{crs}, \ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]}))$.

Indistinguishability of hybrid 0 and hybrid 1 follows directly from Lemma 4.6, as we replace the distribution of crs from $\mathsf{SSBH.crsGen}(1^\lambda)$ to $\mathsf{SSBH.bindingCrsGen}(1^\lambda, L)$. Indistinguishability of hybrids 2 and 3 also follows from Lemma 4.6, as we replace the distribution of crs from $\mathsf{SSBH.bindingCrsGen}(1^\lambda, L)$ back to $\mathsf{SSBH.crsGen}(1^\lambda)$.

We will now show that hybrids 1 and 2 are identically distributed. Since crs is in the support of $\mathsf{SSBH.bindingCrsGen}(1^\lambda, i)$ and $\mathsf{digest} = \mathsf{SSBH.Hash}(\mathsf{crs}, D)$, by Lemma 4.8 it holds that $(\mathsf{digest}, 1 - D[L]) \notin \mathcal{L}_{\mathsf{crs},L}$. By the soundness property of the hash-proof system HPS, it holds that

$$(\mathsf{crs}, L, \mathsf{pk}, \mathsf{HPS.H}_{\mathsf{secret}}(\mathsf{sk}, (\mathsf{digest}, 1 - D[L]))) \equiv (\mathsf{crs}, L, \mathsf{pk}, u),$$

for a uniformly random u. Furthermore, $c_{D[L]}$ can be computed by $m_{D[L]} \oplus \mathsf{HPS.H_{public}}(\mathsf{pk}, (\mathsf{digest}, D[L]), D)$. Hence

$$\begin{aligned}
&(\mathsf{crs}, L, \mathsf{pk}, m_{D[L]} \oplus \mathsf{HPS.H_{secret}}(\mathsf{sk}, (\mathsf{digest}, 1 - D[L])), c_{D[L]}) \\
\equiv &(\mathsf{crs}, L, \mathsf{pk}, u, c_{D[L]}) \\
\equiv &(\mathsf{crs}, L, \mathsf{pk}, m_{1-D[L]} \oplus \mathsf{HPS.H_{secret}}(\mathsf{sk}, (\mathsf{digest}, 1 - D[L])), c_{D[L]}).
\end{aligned}$$

This concludes the proof. □

5 Construction of Updatable Laconic OT

In this subsection, we will construct an updatable laconic OT that supports a hash function that allows for compression from an input (database) of size an arbitrary polynomial in λ to λ bits. As every updatable laconic OT protocol is also a (standard) laconic OT protocol, we will only construct the former. Our main technique in this construction, is the use of garbled circuits to bootstrap a laconic OT with factor-2 compression into one with an arbitrary compression factor.

Below in Section 5.1 we describe some background on the primitives needed for realizing our laconic OT construction. Then we will give the construction of laconic OT along with its correctness and security proofs in Sections 5.2 and 5.3, respectively.

5.1 Background

In this section we recall the needed background of garbled circuits and Merkle trees.

5.1.1 Garbled Circuits

Garbled circuits were first introduced by Yao [Yao82] (see Lindell and Pinkas [LP09] and Bellare et al. [BHR12] for a detailed proof and further discussion). A circuit garbling scheme GC is a tuple of PPT algorithms $(\mathsf{GCircuit}, \mathsf{Eval})$. Very roughly $\mathsf{GCircuit}$ is the circuit garbling procedure and Eval the corresponding evaluation procedure. Looking ahead, each individual wire w of the circuit being garbled will be associated with two labels, namely $\mathsf{key}_{w,0}, \mathsf{key}_{w,1}$.

- $\widetilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}, \{\mathsf{key}_{w,b}\}_{w \in \mathsf{inp}(\mathsf{C}), b \in \{0,1\}}\right)$: $\mathsf{GCircuit}$ takes as input a security parameter λ, a circuit C, and a set of labels $\mathsf{key}_{w,b}$ for all the input wires $w \in \mathsf{inp}(\mathsf{C})$ and $b \in \{0,1\}$. This procedure outputs a *garbled circuit* $\widetilde{\mathsf{C}}$.
- $y \leftarrow \mathsf{Eval}\left(\widetilde{\mathsf{C}}, \{\mathsf{key}_{w,x_w}\}_{w \in \mathsf{inp}(\mathsf{C})}\right)$: Given a garbled circuit $\widetilde{\mathsf{C}}$ and a garbled input represented as a sequence of input labels $\{\mathsf{key}_{w,x_w}\}_{w \in \mathsf{inp}(\mathsf{C})}$, Eval outputs y.

Terminology of Keys and Labels. We note that, in the rest of the paper, we use the notation Keys to refer to both the secret values sampled for wires and the notation Labels to refer to exactly one of them. In other words, generation of garbled circuit involves Keys while computation itself depends just on Labels. Let $\mathsf{Keys} = ((\mathsf{key}_{1,0}, \mathsf{key}_{1,1}), \ldots, (\mathsf{key}_{n,0}, \mathsf{key}_{n,1}))$ be a list of n key-pairs, we denote Keys_x for a string $x \in \{0,1\}^n$ to be a list of labels $(\mathsf{key}_{1,x_1}, \ldots, \mathsf{key}_{n,x_n})$.

Correctness. For correctness, we require that for any circuit C and input $x \in \{0,1\}^m$ (here m is the input length to C) we have that:

$$\Pr\left[\mathsf{C}(x) = \mathsf{Eval}\left(\widetilde{\mathsf{C}}, \{\mathsf{key}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}\right)\right] = 1$$

where $\widetilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}, \{\mathsf{key}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}\right)$.

Security. For security, we require that there is a PPT simulator $\mathsf{CircSim}$ such that for any C, x, and uniformly random keys $\{\mathsf{key}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}$, we have that

$$\left(\widetilde{\mathsf{C}}, \{\mathsf{key}_{w,x_w}\}_{w\in\mathsf{inp}(\mathsf{C})}\right) \overset{c}{\approx} \mathsf{CircSim}\left(1^\lambda, \mathsf{C}, y\right)$$

where $\widetilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}, \{\mathsf{key}_{w,b}\}_{w\in\mathsf{inp}(\mathsf{C}),b\in\{0,1\}}\right)$ and $y = \mathsf{C}(x)$.

5.1.2 Merkle Tree

In this section we briefly review Merkle trees. A Merkle tree is a hash based data structure that generically extend the domain of a hash function. The following description will be tailored to the hash function of the laconic OT scheme that we will present in Section 5.2. Given a two-to-one hash function $\mathsf{Hash} : \{0,1\}^{2\lambda} \to \{0,1\}^\lambda$, we can use a Merkle tree to construct a hash function that compresses a database of an arbitrary (a priori unbounded polynomial in λ) size to a λ-bit string. Now we briefly illustrate how to compress a database $D \in \{0,1\}^M$ (assume for ease of exposition that $M = 2^d \cdot \lambda$). First, we partition D into strings of length 2λ; we call each string a *leaf*. Then we use Hash to compress each leaf into a new string of length λ; we call each string a *node*. Next, we bundle the new nodes in pairs of two and call these pairs *siblings*, i.e., each pair of siblings is a string of length 2λ. We then use Hash again to compress each pair of siblings into a new node of size λ. We continue the process till a single node of size λ is obtained. This process forms a binary tree structure, which we refer to as a Merkle tree. Looking ahead, the hash function of the laconic OT scheme has output $(\hat{\mathsf{D}}, \mathsf{digest})$, where $\hat{\mathsf{D}}$ is the entire Merkle tree, and digest is the root of the tree.

A Merkle tree has the following property. In order to verify that a database D with hash root digest has a certain value b at a location L (namely, $D[L] = b$), there is no need to provide the entire Merkle tree. Instead, it is sufficient to provide a path of siblings from the Merkle tree root to the leaf that contains location L. It can then be easily verified if the hash values from the leaf to the root are correct.

Moreover, a Merkle tree can be updated in the same fashion when the value at a certain location of the database is updated. Instead of recomputing the entire tree, we only need to recompute the nodes on the path from the updated leaf to the root. This can be done given the path of siblings from the root to the leaf.

5.2 Construction

We will now provide our construction to bootstrap an ℓOT scheme with factor-2 compression into an updatable ℓOT scheme with an arbitrary compression factor, which can compress a database of an arbitrary (a priori unbounded polynomial in λ) size.

Overview. We first give an overview of the construction. Consider a database $D \in \{0,1\}^M$ such that $M = 2^d \cdot \lambda$. Given a laconic OT scheme with factor-2 compression (denoted as ℓOT_{const}), we will first use a Merkle tree to obtain a hash function with arbitrary (polynomial) compression factor. As described in Section 5.1.2, the Hash function of the updatable ℓOT scheme will have an output $(\hat{\mathsf{D}}, \mathsf{digest})$, where $\hat{\mathsf{D}}$ is the entire Merkle tree, and digest is the root of the tree.

In the Send algorithm, suppose we want to send a message depending on a bit $D[L]$, we will follow the natural approach of traversing the Merkle tree layer by layer until reaching the leaf containing L. In particular, L can be represented as $L = (b_1, \ldots, b_{d-1}, t)$, where $b_1, \ldots, b_{d-1}$ are bits representing the path from the root to the leaf containing location L, and $t \in [2\lambda]$ is the position within the leaf. The Send algorithm first takes as input the root digest of the Merkle tree, and it will generate a chain of garbled circuits, which would enable the receiver to traverse the Merkle tree from the root to the leaf. And upon reaching the leaf, the receiver will be able to evaluate the last garbled circuit and retrieve the message corresponding to the t-th bit of the leaf.

We briefly explain the chain of garbled circuits as follows. The chain consists of $d-1$ traversing circuits along with a reading circuit. Every traversing circuit takes as input a pair of siblings $\mathsf{sbl} = (\mathsf{sbl}_0, \mathsf{sbl}_1)$ at a certain layer of the Merkle tree, chooses sbl_b which is the node in the path from root to leaf, and generates a laconic OT ciphertext (using $\ell OT_{\mathsf{const}}.\mathsf{Send}$) which encrypts the input keys of the next traversing garbled circuit and uses sbl_b as the hash value. Looking ahead, when the receiver evaluates the traversing circuit and obtains the laconic OT ciphertext, he can then use the siblings at the next layer to decrypt the ciphertext (by $\ell OT_{\mathsf{const}}.\mathsf{Receive}$) and obtain the corresponding input labels for the next traversing garbled circuit. Using the chain of traversing garbled circuits the receiver can therefore traverse from the first layer to the leaf of the Merkle tree. Furthermore, the correct keys for the first traversing circuit are sent via the ℓOT_{const} with digest (i.e., root of the tree) as the hash value.

Finally, the last traversing circuit will transfer keys for the last reading circuit to the receiver in a similar fashion as above. The reading circuit takes the leaf as input and outputs $m_{\mathsf{leaf}[t]}$, i.e., the message corresponding to the t-th bit of the leaf. Hence, when evaluating the reading circuit, the receiver can obtain the message $m_{\mathsf{leaf}[t]}$.

$\mathsf{SendWrite}$ and $\mathsf{ReceiveWrite}$ are similar as Send and $\mathsf{Receive}$, except that (a) $\mathsf{ReceiveWrite}$ updates the Merkle tree from the leaf to the root, and (b) the last writing circuit recomputes the root of the Merkle tree and outputs messages corresponding to the new root. To enable (b), the writing circuit will take as input the whole path of siblings from the root to the leaf. The input keys for the writing circuit corresponding to the siblings at the $(i+1)$-th layer are transferred via the i-th traversing circuit. That is, the i-th traversing circuit transfers the keys for the $(i+1)$-th transferring circuit as well as partial keys for the writing circuit. In the actual construction, both the reading circuit and writing circuit take as input the entire path of siblings (for the purpose of symmetry).

The Construction. Let $\ell OT_{\mathsf{const}} = (\ell OT_{\mathsf{const}}.\mathsf{crsGen}, \ell OT_{\mathsf{const}}.\mathsf{Hash}, \ell OT_{\mathsf{const}}.\mathsf{Send}, \ell OT_{\mathsf{const}}.\mathsf{Receive})$ be a laconic OT protocol with factor-2 compression. Let $\mathsf{GC} = (\mathsf{GCircuit}, \mathsf{Eval})$ be a circuit garbling scheme. Without loss of generality, let $D \in \{0,1\}^M$ be a database such that $|M| = 2^d \cdot \lambda$. A location $L \in [M]$ can be represented as $(b_1, b_2, \ldots, b_{d-1}, t) \in \{0,1\}^{d-1} \times [2\lambda]$, where the bits b_i's define the path from the root to a leaf in the Merkle tree, and $t \in [2\lambda]$ defines a position in that leaf.

Before delving into the construction, we first describe three gadget circuits: the traversing circuit $\mathsf{C}^{\mathsf{trav}}$, the reading circuit $\mathsf{C}^{\mathsf{read}}$, and the writing circuit $\mathsf{C}^{\mathsf{write}}$. These circuits are defined

formally in Figures 4, 5, and 6, respectively.

The traversing circuit has hardwired inside it a common reference string crs, a bit b and two vectors of input keys $\mathsf{Keys}, \widetilde{\mathsf{Keys}}$, each containing 2λ key-pairs (a key-pair is a pair of λ-bit strings). It takes as input a pair of siblings $\mathsf{sbl} = (\mathsf{sbl}_0, \mathsf{sbl}_1)$, each of length λ, and generates two laconic OT Send messages with sbl_b as the digest and $\mathsf{Keys}, \widetilde{\mathsf{Keys}}$ as message vectors respectively. Further, it also has the randomness needed for $\ell OT_{\mathsf{const}}.\mathsf{Send}$ hardwired inside it.

The reading circuit $\mathsf{C}^{\mathsf{read}}$ has a location $t \in [2\lambda]$, and messages $m_0, m_1 \in \{0,1\}^\lambda$ hardwired inside it. It takes as input a path of siblings from the root to a leaf, reads the t-th bit of the leaf, and outputs either m_0 or m_1 depending on that bit.

The writing circuit $\mathsf{C}^{\mathsf{write}}$ has hardwired inside it a common reference string crs, a location $L \in [M]$, a bit b and a vector of messages Keys consisting of λ key-pairs. It takes as input a path of siblings from the root to a leaf, changes the t-th bit of the leaf to b (where L corresponds to t-th location in the leaf), recomputes the Merkle tree root along the path, and outputs the corresponding labels for the new root/digest.

Circuit $\mathsf{C}^{\mathsf{trav}}$

Hardwired Values: crs, b, Keys, $\widetilde{\mathsf{Keys}}$, r, $\widetilde{r}$

Input: sbl

Parse sbl as $(\mathsf{sbl}_0, \mathsf{sbl}_1)$

$\mathsf{e} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}_b, \mathsf{Keys}; r)$

$\widetilde{\mathsf{e}} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}_b, \widetilde{\mathsf{Keys}}; \widetilde{r})$

Output $(\mathsf{e}, \widetilde{\mathsf{e}})$

Figure 4: The Traversing Circuit $\mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b, \mathsf{Keys}, \widetilde{\mathsf{Keys}}, r, \widetilde{r}]$

Circuit $\mathsf{C}^{\mathsf{read}}$

Hardwired Values: t, m_0, m_1

Input: path

Parse $\mathsf{path} = (\mathsf{sbl}^1, \ldots, \mathsf{sbl}^{d-1}, \mathsf{leaf})$

Output $m_{\mathsf{leaf}[t]}$

Figure 5: The Reading Circuit $\mathsf{C}^{\mathsf{read}}[t, m_0, m_1]$

Now we construct the updatable ℓOT, namely $(\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive}, \mathsf{SendWrite}, \mathsf{ReceiveWrite})$ as follows.

- $\mathsf{crsGen}(1^\lambda)$: Sample $\mathsf{crs} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{crsGen}(1^\lambda)$ and output crs.
- $\mathsf{Hash}(\mathsf{crs}, D \in \{0,1\}^M)$:
  Build a Merkle tree $\hat{\mathsf{D}}$ of D using $\ell OT_{\mathsf{const}}.\mathsf{Hash}(\mathsf{crs}, \cdot)$, as in Section 5.1.2.
  Let digest be the root of $\hat{\mathsf{D}}$.
  Output $(\mathsf{digest}, \hat{\mathsf{D}})$.
- $\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1)$:
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$.
  Pick $\left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)$ as input keys for $\mathsf{C}^{\mathsf{read}}$,

Circuit $\mathsf{C}^{\mathsf{write}}$

Hardwired Values: crs, L, b, Keys

Input: path

Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$

Parse $\mathsf{path} = (\mathsf{sbl}^1, \ldots, \mathsf{sbl}^{d-1}, \mathsf{leaf})$, and parse $\mathsf{sbl}^i = (\mathsf{sbl}^i_0, \mathsf{sbl}^i_1)$ for $i \in [d-1]$

$\mathsf{leaf}[t] \leftarrow b$

$\mathsf{sbl}^d \leftarrow \mathsf{leaf}$

For $i = d-1$ downto 1:

$\quad \mathsf{sbl}^i_{b_i} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Hash}(\mathsf{crs}, \mathsf{sbl}^{i+1})$

$\mathsf{digest}^* \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Hash}(\mathsf{crs}, \mathsf{sbl}^1)$.

Output $\mathsf{Keys}_{\mathsf{digest}^*}$

Figure 6: The Writing Circuit $\mathsf{C}^{\mathsf{write}}[\mathsf{crs}, L, b, \mathsf{Keys}]$

where $\widetilde{\mathsf{Keys}}^i$ corresponds to the input keys of sbl^i for $i \in [d-1]$,
and $\widetilde{\mathsf{Keys}}^d$ corresponds to the input keys of leaf.

$\widetilde{\mathsf{C}}^{\mathsf{read}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{read}}[t, m_0, m_1], \left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)\right)$

Let Keys^d be 0^*

For $i = d-1$ downto 1:

Pick Keys^i as input keys for $\mathsf{C}^{\mathsf{trav}}$

Pick $r_i, \widetilde{r}_i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$

$\widetilde{\mathsf{C}}_i \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_i, \mathsf{Keys}^{i+1}, \widetilde{\mathsf{Keys}}^{i+1}, r_i, \widetilde{r}_i], \mathsf{Keys}^i\right)$

$\mathsf{e}_0 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \mathsf{Keys}^1)$

$\widetilde{\mathsf{e}}_0 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \widetilde{\mathsf{Keys}}^1)$

Output $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$

- $\mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, L, \mathsf{e})$:

Parse $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$

Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$

Parse $\hat{\mathsf{D}}$ as a Merkle tree.

Denote the end node of path $b_1 b_2 \ldots b_i$ by $\hat{\mathsf{D}}_{b_1 b_2 \ldots b_i}$.

For $i = 1$ to $d-1$:

$\quad \mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 1})$

$\quad \mathsf{Labels}^i \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \mathsf{e}_{i-1}, \mathsf{sbl}^i)$

$\quad \widetilde{\mathsf{Labels}}^i \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_{i-1}, \mathsf{sbl}^i)$

$\quad (\mathsf{e}_i, \widetilde{\mathsf{e}}_i) \leftarrow \mathsf{Eval}(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i)$

$\mathsf{leaf} \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 1})$

$\widetilde{\mathsf{Labels}}^d \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_{d-1}, \mathsf{leaf})$

$m \leftarrow \mathsf{Eval}\left(\widetilde{\mathsf{C}}^{\mathsf{read}}, \left(\widetilde{\mathsf{Labels}}^1, \ldots, \widetilde{\mathsf{Labels}}^d\right)\right)$

Output m

- $\mathsf{SendWrite}(\mathsf{crs}, \mathsf{digest}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^\lambda)$:

Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$.
Pick $\left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)$ as input keys for $\mathsf{C}^{\mathsf{write}}$,
where $\widetilde{\mathsf{Keys}}^i$ corresponds to the input keys of sbl^i for $i \in [d-1]$,
and $\widetilde{\mathsf{Keys}}^d$ corresponds to the input keys of leaf.
$\widetilde{\mathsf{C}}^{\mathsf{write}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{write}}[\mathsf{crs}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^\lambda], \left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)\right)$
Let Keys^d be 0^*
For $i = d-1$ downto 1:
Pick Keys^i as input keys for $\mathsf{C}^{\mathsf{trav}}$
Pick $r_i, \widetilde{r}_i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$
$\widetilde{\mathsf{C}}_i \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_i, \mathsf{Keys}^{i+1}, \widetilde{\mathsf{Keys}}^{i+1}, r_i, \widetilde{r}_i], \mathsf{Keys}^i\right)$
$\mathsf{e}_0 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \mathsf{Keys}^1)$
$\widetilde{\mathsf{e}}_0 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{digest}, \widetilde{\mathsf{Keys}}^1)$
Output $\mathsf{e}_{\mathsf{w}} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{write}})$

- $\mathsf{ReceiveWrite}^{\hat{\mathsf{D}}}(\mathsf{crs}, L, b, \mathsf{e}_{\mathsf{w}})$:
Parse $\mathsf{e}_{\mathsf{w}} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{write}})$
Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$
Parse $\hat{\mathsf{D}}$ as a Merkle tree.
Denote the end node of path $b_1 b_2 \ldots b_i$ by $\hat{\mathsf{D}}_{b_1 b_2 \ldots b_i}$.
Computing messages corresponding to the new digest:
For $i = 1$ to $d-1$:
$\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 1})$
$\mathsf{Labels}^i \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \mathsf{e}_{i-1}, \mathsf{sbl}^i)$
$\widetilde{\mathsf{Labels}}^i \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_{i-1}, \mathsf{sbl}^i)$
$(\mathsf{e}_i, \widetilde{\mathsf{e}}_i) \leftarrow \mathsf{Eval}(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i)$
$\mathsf{leaf} \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 1})$
$\widetilde{\mathsf{Labels}}^d \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_{d-1}, \mathsf{leaf})$
$\{m_j\}_{j=1}^\lambda \leftarrow \mathsf{Eval}\left(\widetilde{\mathsf{C}}^{\mathsf{write}}, \left(\widetilde{\mathsf{Labels}}^1, \ldots, \widetilde{\mathsf{Labels}}^d\right)\right)$
Updating the Merkle tree:
$\left(\hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 0} || \hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 1}\right)[t] \leftarrow b$
For $i = d-1$ downto 0:
$\hat{\mathsf{D}}_{b_1 \ldots b_i} \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Hash}(\mathsf{crs}, \hat{\mathsf{D}}_{b_1 \ldots b_i 0} || \hat{\mathsf{D}}_{b_1 \ldots b_i 1})$
Update digest with the new root of $\hat{\mathsf{D}}$

Output $\{m_j\}_{j=1}^\lambda$

Efficiency. It can be seen from the scheme that the length of digest is λ. The algorithm Hash runs in time $|D| \cdot \mathsf{poly}(\log |D|, \lambda)$. Furthermore, $\mathsf{Send}, \mathsf{Receive}, \mathsf{SendWrite}, \mathsf{SendWrite}$ all run in time $\mathsf{poly}(\log |D|, \lambda)$.

Correctness. We briefly argue (perfect) correctness of the updatable laconic OT scheme. Given a ciphertext $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$ computed by Send, correctness of ℓOT_{const} ensures that $\mathsf{Labels}^1 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \mathsf{e}_0, \mathsf{sbl}^1)$ outputs the correct labels for $\widetilde{\mathsf{C}}_1$ and that $\widetilde{\mathsf{Labels}}^1 \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_0, \mathsf{sbl}^1)$ outputs the correct labels for $\widetilde{\mathsf{C}}^{\mathsf{read}}$, namely $\mathsf{Labels}^1 = \mathsf{Keys}^1_{\mathsf{sbl}^1}$ and $\widetilde{\mathsf{Labels}}^1 = \widetilde{\mathsf{Keys}}^1_{\mathsf{sbl}^1}$. In turn, correctness of the garbling scheme guarantees that $\widetilde{\mathsf{C}}_1$ outputs the correct $(\mathsf{e}_1, \widetilde{\mathsf{e}}_1)$, namely $\mathsf{e}_1 = \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^1_{b_1}, \mathsf{Keys}^2; r_1)$ and $\widetilde{\mathsf{e}}_1 = \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^1_{b_1}, \widetilde{\mathsf{Keys}}^2; \widetilde{r}_1)$. It follows inductively that for every $i = 1, 2, \ldots, d-1$, $\mathsf{Labels}^i = \mathsf{Keys}^i_{\mathsf{sbl}^i}$, $\widetilde{\mathsf{Labels}}^i = \widetilde{\mathsf{Keys}}^i_{\mathsf{sbl}^i}$, $\mathsf{e}_i = \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^i_{b_i}, \mathsf{Keys}^{i+1}; r_i)$, $\widetilde{\mathsf{e}}_i = \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^i_{b_i}, \widetilde{\mathsf{Keys}}^{i+1}; \widetilde{r}_i)$. Again by using the correctness of ℓOT_{const}, $\widetilde{\mathsf{Labels}}^d \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Receive}(\mathsf{crs}, \widetilde{\mathsf{e}}_{d-1}, \mathsf{leaf})$ gives $\widetilde{\mathsf{Labels}}^d = \widetilde{\mathsf{Keys}}^d_{\mathsf{leaf}}$. Then by using correctness of the garbling scheme it follows that evaluating $\widetilde{\mathsf{C}}^{\mathsf{read}}$ gives the correct output $m_{D[L]}$. Correctness with regard to writes can be argued analogously.

5.3 Security

In this section, we will prove the security of the above updatable laconic OT scheme.

Theorem 5.1 (Sender Privacy against Semi-honest Receivers)**.** *Given that ℓOT_{const} has sender privacy and that the garbled circuit scheme* $\mathsf{GCircuit}$ *is secure, the updatable laconic OT scheme ℓOT has sender privacy.*

Proof. Let $\ell\mathsf{OTSim}_{\mathsf{const}}$ be the simulator for ℓOT_{const} and $\mathsf{CircSim}$ be the simulator for the garbling scheme $\mathsf{GCircuit}$. Below, we provide the two simulators $\ell\mathsf{OTSim}$ for a read and $\ell\mathsf{OTSimWrite}$ for the write.

- $\ell\mathsf{OTSim}(\mathsf{crs}, D, L, m)$:
  $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$
  $\left(\widetilde{\mathsf{C}}^{\mathsf{read}}, \left(\widetilde{\mathsf{Labels}}^1, \ldots, \widetilde{\mathsf{Labels}}^d\right)\right) \leftarrow \mathsf{CircSim}(1^\lambda, \mathsf{C}^{\mathsf{read}}, m)$
  $\mathsf{leaf} \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{d-1} 1})$
  $\mathsf{e}_{d-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{leaf}, 0^*)$
  $\widetilde{\mathsf{e}}_{d-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}\left(\mathsf{crs}, \mathsf{leaf}, \widetilde{\mathsf{Labels}}^d\right)$
  For $i = d-1$ downto 1:
    $(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i) \leftarrow \mathsf{CircSim}(1^\lambda, \mathsf{C}^{\mathsf{trav}}, (\mathsf{e}_i, \widetilde{\mathsf{e}}_i))$
    $\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 1})$
    $\mathsf{e}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \mathsf{Labels}^i)$
    $\widetilde{\mathsf{e}}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \widetilde{\mathsf{Labels}}^i)$
  Output $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$

- $\ell\mathsf{OTSimWrite}(\mathsf{crs}, D, L, b, \{m_j\}_{j=1}^\lambda)$:
  $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$
  $\left(\widetilde{\mathsf{C}}^{\mathsf{write}}, \left(\widetilde{\mathsf{Labels}}^1, \ldots, \widetilde{\mathsf{Labels}}^d\right)\right) \leftarrow \mathsf{CircSim}(1^\lambda, \mathsf{C}^{\mathsf{write}}, \{m_j\}_{j=1}^\lambda)$

$\mathsf{leaf} \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{d-1}0}, \hat{\mathsf{D}}_{b_1 \ldots b_{d-1}1})$
$\mathsf{e}_{d-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{leaf}, 0^*)$
$\widetilde{\mathsf{e}}_{d-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}\left(\mathsf{crs}, \mathsf{leaf}, \widetilde{\mathsf{Labels}}^d\right)$
For $i = d-1$ downto 1:
  $(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i) \leftarrow \mathsf{CircSim}(1^\lambda, \mathsf{C}^{\mathsf{trav}}, (\mathsf{e}_i, \widetilde{\mathsf{e}}_i))$
  $\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1}0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1}1})$
  $\mathsf{e}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \mathsf{Labels}^i)$
  $\widetilde{\mathsf{e}}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \widetilde{\mathsf{Labels}}^i)$
Output $\mathsf{e}_{\mathsf{w}} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{write}})$

In the following we will only prove sender security with regard to reads. Since $(\mathsf{Send}, \ell\mathsf{OTSim})$ and $(\mathsf{SendWrite}, \ell\mathsf{OTSimWrite})$ are very similar, sender security with regard to writes can be argued analogously.

We prove security via a hybrid argument. In the first hybrid, we replace the ciphertexts e_0 and $\widetilde{\mathsf{e}}_0$ computed by $\ell OT_{\mathsf{const}}.\mathsf{Send}$ with ciphertexts computed by $\ell\mathsf{OTSim}_{\mathsf{const}}$.

Afterwards, we can use security of the garbling scheme to replace the honestly generated $\widetilde{\mathsf{C}}_1$ with a simulated one, and run $\ell\mathsf{OTSim}_{\mathsf{const}}$ using the simulated input labels of $\widetilde{\mathsf{C}}_1$. As the output of $\widetilde{\mathsf{C}}_1$ is again a pair of ciphertexts $(\mathsf{e}_1, \widetilde{\mathsf{e}}_1)$, we will simulate it using $\ell\mathsf{OTSim}_{\mathsf{const}}$ in the next hybrid. We continue alternating between simulating the garbled circuits and simulating the ciphertexts, until reaching the reading circuit. Once we reach the reading circuit, it holds that all $\widetilde{\mathsf{Labels}}^i$ are information theoretically fixed to the path from the root to the leaf containing L. We will then invoke the garbled circuit security of the reading circuit, and conclude the hybrid argument.

The formal proof is as follows. For every PPT machine $\mathcal{A}$, let $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$, and let $(D, L, m_0, m_1) \leftarrow \mathcal{A}(\mathsf{crs})$. Further let $\mathsf{digest} \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$. Then we will prove that the two distributions $(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1))$ and $(\mathsf{crs}, \ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]}))$ are computationally indistinguishable. Consider the following hybrids.

- **Hybrid 0**: This is the real experiment, i.e., $(\mathsf{crs}, \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, m_0, m_1))$.
- **Hybrid 1**: Same has hybrid 0, except that e_0 and $\widetilde{\mathsf{e}}_0$ are computed as follows.

$\boxed{(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)}$
Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$.
Pick $\left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)$ as input keys for $\mathsf{C}^{\mathsf{read}}$
$\widetilde{\mathsf{C}}^{\mathsf{read}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{read}}[t, m_0, m_1], \left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)\right)$
Let Keys^d be 0^*
For $i = d-1$ downto 1:
  Pick Keys^i as input keys for $\mathsf{C}^{\mathsf{trav}}$
  Pick $r_i, \widetilde{r}_i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$
  $\widetilde{\mathsf{C}}_i \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_i, \mathsf{Keys}^{i+1}, \widetilde{\mathsf{Keys}}^{i+1}, r_i, \widetilde{r}_i], \mathsf{Keys}^i\right)$
$\boxed{\mathsf{sbl}^1 \leftarrow (\hat{\mathsf{D}}_0, \hat{\mathsf{D}}_1)}$
$\boxed{\mathsf{Labels}^1 \leftarrow \mathsf{Keys}^1_{\mathsf{sbl}^1}}$

$\boxed{\widetilde{\mathsf{Labels}}^1 \leftarrow \widetilde{\mathsf{Keys}}^1_{\mathsf{sbl}^1}}$

$\boxed{\mathsf{e}_0 \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^1, \mathsf{Labels}^1)}$

$\boxed{\widetilde{\mathsf{e}}_0 \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^1, \widetilde{\mathsf{Labels}}^1)}$

Output $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$

The differences between hybrid 0 and hybrid 1 have been marked with boxes. Indistinguishability between hybrid 0 and hybrid 1 can be argued from the multi-execution sender security of ℓOT_{const} via the following reduction. Given crs by the experiment and the adversarial input D, compute hybrid 1 until e_0 and $\widetilde{\mathsf{e}}_0$ are computed. In particular, compute $\hat{\mathsf{D}}, \mathsf{Keys}^1, \widetilde{\mathsf{Keys}}^1, \mathsf{sbl}^1, \mathsf{Labels}^1 = \mathsf{Keys}^1_{\mathsf{sbl}^1}, \widetilde{\mathsf{Labels}}^1 = \widetilde{\mathsf{Keys}}^1_{\mathsf{sbl}^1}$. Then choose sbl^1 as the database and $(\mathsf{Keys}^1, \widetilde{\mathsf{Keys}}^1)$ as the messages for ℓOT_{const}, and obtain the challenge $(\mathsf{e}_0^*, \widetilde{\mathsf{e}}_0^*)$, which is from one of the following two distributions:

$$\left(\ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^1, \mathsf{Keys}^1), \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^1, \widetilde{\mathsf{Keys}}^1)\right);$$
$$\left(\ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^1, \mathsf{Labels}^1), \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^1, \widetilde{\mathsf{Labels}}^1)\right).$$

If $(\mathsf{e}_0^*, \widetilde{\mathsf{e}}_0^*)$ is from the first distribution, then it results in hybrid 0; otherwise it results in hybrid 1. Hence the indistinguishability of the two distributions implies indistinguishability of the two hybrids.

- **Hybrid** $2k$ $(k = 1, 2, \ldots, d-1)$: Same has hybrid $2k-1$, except that $\widetilde{\mathsf{C}}_k$ is computed as follows.

  $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$
  Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$.
  Pick $\left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)$ as input keys for $\mathsf{C}^{\mathsf{read}}$
  $\widetilde{\mathsf{C}}^{\mathsf{read}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{read}}[t, m_0, m_1], \left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)\right)$
  Let Keys^d be 0^*
  For $i = d-1$ downto $\boxed{k+1}$:
    Pick Keys^i as input keys for $\mathsf{C}^{\mathsf{trav}}$
    Pick $r_i, \widetilde{r}_i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$
    $\widetilde{\mathsf{C}}_i \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_i, \mathsf{Keys}^{i+1}, \widetilde{\mathsf{Keys}}^{i+1}, r_i, \widetilde{r}_i], \mathsf{Keys}^i\right)$

  $\boxed{\mathsf{sbl}^k \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{k-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{k-1} 1})}$

  $\boxed{\mathsf{e}_k \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^k_{b_k}, \mathsf{Keys}^{k+1})}$

  $\boxed{\widetilde{\mathsf{e}}_k \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^k_{b_k}, \widetilde{\mathsf{Keys}}^{k+1})}$

  For $i = \boxed{k}$ downto 1:
    $\boxed{(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i) \leftarrow \mathsf{CircSim}(1^\lambda, \mathsf{C}^{\mathsf{trav}}, (\mathsf{e}_i, \widetilde{\mathsf{e}}_i))}$
    $\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 1})$

$\widetilde{\mathsf{Labels}}^i \leftarrow \widetilde{\mathsf{Keys}}^i_{\mathsf{sbl}^i}$

$\mathsf{e}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \mathsf{Labels}^i)$

$\widetilde{\mathsf{e}}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \widetilde{\mathsf{Labels}}^i)$

Output $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$

- **Hybrid** $2k+1$ ($k = 1, 2, \ldots, d-1$): Same has hybrid $2k$, except that e_k and $\widetilde{\mathsf{e}}_k$ are computed as follows.

$(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$

Parse $L = (b_1, b_2, \ldots, b_{d-1}, t)$.

Pick $\left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)$ as input keys for $\mathsf{C}^{\mathsf{read}}$

$\widetilde{\mathsf{C}}^{\mathsf{read}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{read}}[t, m_0, m_1], \left(\widetilde{\mathsf{Keys}}^1, \ldots, \widetilde{\mathsf{Keys}}^d\right)\right)$

Let Keys^d be 0^*

For $i = d-1$ downto $k+1$:

Pick Keys^i as input keys for $\mathsf{C}^{\mathsf{trav}}$

Pick $r_i, \widetilde{r}_i$ as random coins for $\ell OT_{\mathsf{const}}.\mathsf{Send}$

$\widetilde{\mathsf{C}}_i \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_i, \mathsf{Keys}^{i+1}, \widetilde{\mathsf{Keys}}^{i+1}, r_i, \widetilde{r}_i], \mathsf{Keys}^i\right)$

$\mathsf{sbl}^{k+1} \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_k 0}, \hat{\mathsf{D}}_{b_1 \ldots b_k 1})$

$\mathsf{Labels}^{k+1} \leftarrow \mathsf{Keys}^{k+1}_{\mathsf{sbl}^{k+1}}$

$\widetilde{\mathsf{Labels}}^{k+1} \leftarrow \widetilde{\mathsf{Keys}}^{k+1}_{\mathsf{sbl}^{k+1}}$

$\mathsf{e}_k \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^{k+1}, \mathsf{Labels}^{k+1})$

$\widetilde{\mathsf{e}}_k \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^{k+1}, \widetilde{\mathsf{Labels}}^{k+1})$

For $i = k$ downto 1:

$(\widetilde{\mathsf{C}}_i, \mathsf{Labels}^i) \leftarrow \mathsf{CircSim}(1^\lambda, \mathsf{C}^{\mathsf{trav}}, (\mathsf{e}_i, \widetilde{\mathsf{e}}_i))$

$\mathsf{sbl}^i \leftarrow (\hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 0}, \hat{\mathsf{D}}_{b_1 \ldots b_{i-1} 1})$

$\widetilde{\mathsf{Labels}}^i \leftarrow \widetilde{\mathsf{Keys}}^i_{\mathsf{sbl}^i}$

$\mathsf{e}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \mathsf{Labels}^i)$

$\widetilde{\mathsf{e}}_{i-1} \leftarrow \ell\mathsf{OTSim}_{\mathsf{const}}(\mathsf{crs}, \mathsf{sbl}^i, \widetilde{\mathsf{Labels}}^i)$

Output $\mathsf{e} = (\mathsf{e}_0, \widetilde{\mathsf{e}}_0, \widetilde{\mathsf{C}}_1, \ldots, \widetilde{\mathsf{C}}_{d-1}, \widetilde{\mathsf{C}}^{\mathsf{read}})$

We will first show that hybrids $2k-1$ and $2k$ are indistinguishable via a reduction to the security of the garbling scheme $\mathsf{GCircuit}$. Notice that the only difference between hybrids $2k-1$ and $2k$ is $(\widetilde{\mathsf{C}}_k, \mathsf{e}_{k-1})$. Consider the following two distributions:

$$(\widetilde{\mathsf{C}}_k, \mathsf{Labels}^k) \leftarrow \left(\mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{trav}}[\mathsf{crs}, b_k, \mathsf{Keys}^{k+1}, \widetilde{\mathsf{Keys}}^{k+1}, r_i, \widetilde{r}_k], \mathsf{Keys}^k\right), \mathsf{Keys}^k_{\mathsf{sbl}^k}\right);$$

$$(\widetilde{\mathsf{C}}_k, \mathsf{Labels}^k) \leftarrow \mathsf{CircSim}\left(1^\lambda, \mathsf{C}^{\mathsf{trav}}, (\mathsf{e}_k, \widetilde{\mathsf{e}}_k)\right),$$

where $\mathsf{e}_k \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^k_{b_k}, \mathsf{Keys}^{k+1})$ and $\widetilde{\mathsf{e}}_k \leftarrow \ell OT_{\mathsf{const}}.\mathsf{Send}(\mathsf{crs}, \mathsf{sbl}^k_{b_k}, \widetilde{\mathsf{Keys}}^{k+1})$. Notice that $(\mathsf{e}_k, \widetilde{\mathsf{e}}_k)$ is the output of $(\widetilde{\mathsf{C}}_k, \mathsf{Labels}^k)$ from the first distribution. By security of

the garbled circuit scheme, the above two distributions are computationally indistinguishable. Furthermore, if $\widetilde{\mathsf{C}}_k$ is generated using the first distribution and e_{k-1} is computed using Labels^k from the first distribution, then it results in hybrid $2k-1$; otherwise it results in hybrid $2k$. Hence the two hybrids are computationally indistinguishable.

Indistinguishability of hybrids $2k$ and $2k+1$ follows again from sender security of ℓOT_{const}, in the same fashion as the indistinguishability between hybrids 0 and 1.

- **Hybrid** $2d$: This is the simulated experiment, namely $(\mathsf{crs}, \ell\mathsf{OTSim}(\mathsf{crs}, D, L, m_{D[L]}))$.

  The difference between hybrids $2d-1$ and $2d$ is $\left(\widetilde{\mathsf{C}}^{\mathsf{read}}, \widetilde{\mathsf{e}}_0, \ldots, \widetilde{\mathsf{e}}_{d-1}\right)$. The indistinguishability would follow from the security of garbled circuit scheme, similarly as when we argue indistinguishability hybrids $2k-1$ and $2k$.

□

6 Warm-Up Application: Non-Interactive Secure Computation (NISC) on Large Inputs in RAM Setting

In this section, we consider the application of non-interactive secure computation in the RAM (random access machine) setting.

6.1 Background

We recall the needed background of RAM computation model and two-message oblivious transfer in this section. We will also use garbled circuits (see Section 5.1.1) as building blocks.

6.1.1 Random Access Machine (RAM) Model of Computation

Now we define the RAM model of computation. Parts of this subsection have been taken verbatim from [GLO15].

Notation for the RAM Model of Computation. The RAM model consists of a CPU and a memory storage of size M. The CPU executes a program that can access the memory by using read/write operations. In particular, for a program P with memory of size M we denote the initial contents of the memory data by $D \in \{0,1\}^M$. Additionally, the program gets a "short" input $x \in \{0,1\}^m$, which we alternatively think of as the initial state of the program. We use the notation $P^D(x)$ to denote the execution of program P with initial memory contents D and input x. The program P can read from and write to various locations in memory D throughout its execution.[10]

We will also consider the case where several different programs are executed sequentially and the memory persists between executions. We denote this process as $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$ to indicate that first $P_1^D(x_1)$ is executed, resulting in some memory contents D_1 and output y_1, then $P_2^{D_1}(x_2)$ is executed resulting in some memory contents D_2 and output y_2 etc. As an example,

[10] In general, the distinction between what to include in the program P, the memory data D and the short input x can be somewhat arbitrary. However as motivated by our applications we will typically be interested in a setting where the data D is large while the size of the program $|P|$ and input length m is small.

imagine that D is a huge database and the programs P_i are database queries that can read and possibly write to the database and are parameterized by some values x_i.

CPU-Step Circuit. Consider an execution of a RAM program which involves at most t CPU steps. We represent a RAM program P via t small *CPU-Step Circuits* each of which executes one CPU step. In this work we will denote one CPU step by:

$$C^P_{\mathsf{CPU}}(\mathsf{state}, \mathsf{rData}) = (\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData})$$

This circuit takes as input the current CPU state state and a bit rData. Looking ahead the bit rData will be read from the memory location that was requested by the previous CPU step. The circuit outputs an updated state state', a read or write bit $\mathsf{R/W}$, the next location to read/write from $L \in [M]$, and a bit wData to write into that location ($\mathsf{wData} = \perp$ when reading). The sequence of locations and read/write values collectively form what is known as the *access pattern*, namely $\mathsf{MemAccess} = \{(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau) : \tau = 1, \ldots, t\}$.

Note that in the description above without loss of generality we have made some simplifying assumptions. We assume that each CPU-step circuit always reads from or write some location in memory. This is easy to implement via a dummy read and write step. Moreover, we assume that the instructions of the program itself are hardwired into the CPU-step circuits.

Representing RAM computation by CPU-Step Circuits. The computation $P^D(x)$ starts with the initial state set as $\mathsf{state}_1 = x$. In each step $\tau \in \{1, \ldots t\}$, the computation proceeds as follows: If $\tau = 1$ or $\mathsf{R/W}^{\tau-1} = \mathsf{write}$, then $\mathsf{rData}^\tau := 0$; otherwise $\mathsf{rData}^\tau := D[L^{\tau-1}]$. Next it executes the CPU-Step Circuit $C^P_{\mathsf{CPU}}(\mathsf{state}^\tau, \mathsf{rData}^\tau) = (\mathsf{state}^{\tau+1}, \mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$. If $\mathsf{R/W}^\tau = \mathsf{write}$, then set $D[L^\tau] = \mathsf{wData}^\tau$. Finally, when $\tau = t$, then $\mathsf{state}^{\tau+1}$ is the output of the program.

6.1.2 Oblivious Transfer

[AIR01, NP01, HK12] gave two-message oblivious transfer (OT) protocols. We describe the definition below and refer the reader to [AIR01, NP01, HK12] for details.

Definition 6.1 (Two-Message Oblivious Transfer)**.** *A two-message oblivious transfer protocol* $\mathsf{OT} = (\mathsf{OT_1}, \mathsf{OT_2}, \mathsf{OT_3})$ *is a protocol between a sender* S *and a receiver* R *where* S *gets as input two strings* s_1, s_2 *of equal length and* R *gets as input a choice bit* $x \in \{0, 1\}$*. The algorithms have the following syntax:*

- $(m_1, \mathit{secret}) \leftarrow \mathsf{OT_1}(1^\lambda, x)$*: It takes as input the security parameter* 1^λ *and receiver's choice bit* $x \in \{0, 1\}$ *and outputs the first OT message* m_1 *(sent by the receiver) and receiver's secret state secret.*
- $m_2 \leftarrow \mathsf{OT_2}(m_1, s_0, s_1)$*: It takes as input the first OT message and the sender's input* (s_0, s_1)*, and outputs the second OT message* m_2 *(sent back to the receiver).*
- $s \leftarrow \mathsf{OT_3}(m_2, \mathit{secret})$*: It takes* m_2 *and secret as input, and outputs a string* s*.*

The following conditions are satisfied:

- ***Perfect Correctness:*** *For all security parameter λ, sender input strings (s_1, s_2) of equal length, and receiver's choice bit x, let $(m_1, secret) \leftarrow \mathsf{OT}_1(1^\lambda, x)$, $m_2 \leftarrow \mathsf{OT}_2(m_1, s_0, s_1)$, and $s \leftarrow \mathsf{OT}_3(m_2, secret)$, then it holds that*

$$\Pr[s = s_x] = 1.$$

- ***Receiver Security:*** *The following two distributions are computationally indistinguishable:*

$$\mathsf{OT}_1(1^\lambda, 0) \overset{c}{\approx} \mathsf{OT}_1(1^\lambda, 1).$$

- ***Sender Security:*** *There exists a PPT simulator OTSim such that for all sender input strings (s_1, s_2) of equal length and receiver's choice bit x, and any first message m_1 in the support of $\mathsf{OT}_1(1^\lambda, x)$, the following two distributions are statistically close:*

$$\mathsf{OT}_2(m_1, s_0, s_1) \overset{s}{\approx} \mathsf{OTSim}(1^\lambda, x, s_x, m_1).$$

We described the above definition with respect to one OT, but the same formalism naturally extends to support multiple parallel executions of OT. We will use the following shorthand notations (generalizing the above notions) to run multiple parallel executions. Let $\mathsf{Keys} = ((\mathsf{Key}_{1,0}, \mathsf{Key}_{1,1}), \ldots, (\mathsf{Key}_{n,0}, \mathsf{Key}_{n,1}))$ be a list of n string-pairs, and $x \in \{0,1\}^n$ be an n-bit choice string. Then we define

- $(m_1, \text{secret}) \leftarrow \mathsf{OT}_1(1^\lambda, x) = \left(\mathsf{OT}_1(1^\lambda, x_1), \ldots, \mathsf{OT}_1(1^\lambda, x_n)\right)$.
- $m_2 \leftarrow \mathsf{OT}_2(m_1, \mathsf{Keys}) = \left(\mathsf{OT}_2(m_{1,1}, \mathsf{Key}_{1,0}, \mathsf{Key}_{1,1}), \cdots, \mathsf{OT}_2(m_{1,n}, \mathsf{Key}_{n,0}, \mathsf{Key}_{n,1})\right)$.
- $\mathsf{Labels} \leftarrow \mathsf{OT}_3(m_2, \text{secret}) = (\mathsf{OT}_3(m_{2,1}, \text{secret}_1), \ldots, \mathsf{OT}_3(m_{2,n}, \text{secret}_n))$.

In the above $m_1 = (m_{1,1}, \ldots, m_{1,n})$, $m_2 = (m_{2,1}, \ldots, m_{2,n})$, $\text{secret} = (\text{secret}_1, \ldots, \text{secret}_n)$. Correctness guarantees that $\mathsf{Labels} = \mathsf{Keys}_x = \left(\mathsf{Key}_{1,x_1}, \ldots, \mathsf{Key}_{n,x_n}\right)$.

Moreover, we will use two important properties of the oblivious transfer [NP01] for our applications: (1) Security holds for multiple second OT messages with regard to the same first OT message. This will be crucial for extending NISC for RAM to support multiple senders with the same receiver. (2) The second OT message is re-randomizable. This will be crucial for the application of multi-hop homomorphic encryption for RAM.

6.2 Formal Model for NISC in RAM Setting

Suppose the receiver owns a large confidential database $D \in \{0,1\}^M$. It first publishes a *short* message, denoted by m_1, which hides D. Afterwards, if a sender wants to run a RAM program P (with input x) on D, it can send a single message m_2 to the receiver. For security we require that m_2 only reveals the output $P^D(x)$ and the memory access pattern $\mathsf{MemAccess}$ of the execution to the receiver. We require that once m_1 is published, the computational cost of both the sender (in computing m_2) and the receiver (in evaluation), as well as the size of m_2, should grow only with the running time of the RAM computation and the size of m_1, and is independent of the size of D.

Moreover, the sender can run a sequence of programs on a persistent database by sending one message per program to the receiver. Finally, the receiver can run the protocol in parallel with multiple senders, where the same m_1 is used. For ease of exposition, below we will describe the setting of one single sender executing one program with the receiver. We provide details on above extensions in Section 6.6.

The Model. A non-interactive secure RAM computation scheme $\mathsf{NISC\text{-}RAM} = (\mathsf{Setup}, \mathsf{EncData}, \mathsf{EncProg}, \mathsf{Dec})$ has the following syntax. It is a two-party protocol between a receiver holding a large secret database D and a sender holding secret program P of running time t and a short input x.

- **Setup:** $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$.
  On input the security parameter 1^λ, it outputs a common reference string.
- **Database Encryption:** $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$.
  On input the common reference string crs and a database $D \in \{0,1\}^M$, it outputs a message m_1 and a secret state $\widetilde{D}$. The receiver publishes m_1 as the short message corresponding to D.
- **Program Encryption:** $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$.
  It takes as input the crs, a message m_1, a RAM program P with input x and maximum run-time t. It then outputs another message m_2. The sender sends the message m_2.
- **Decryption:** $y \leftarrow \mathsf{Dec}^{\widetilde{D}}(\mathsf{crs}, m_2)$.
  The procedure Dec is modeled as a RAM program that can read and write to arbitrary locations of its database initially containing $\widetilde{D}$. This procedure is run by the receiver. On input the crs and m_2, it outputs y.

The following conditions are satisfied:

- **Correctness:** For every database $D \in \{0,1\}^M$ where $M = \mathsf{poly}(\lambda)$ for any polynomial function $\mathsf{poly}(\cdot)$, for every RAM program (P, x, t), it holds that
$$\Pr\left[\mathsf{Dec}^{\widetilde{D}}(\mathsf{crs}, m_2) = P^D(x)\right] = 1,$$
where $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$, $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$, $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$.
- **Receiver Privacy:** For every pair of databases $D_0 \in \{0,1\}^M, D_1 \in \{0,1\}^M$ where M is polynomial in λ, for every crs in the support of $\mathsf{Setup}(1^\lambda)$, let $(m_0, \widetilde{D}_0) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_0)$, $(m_1, \widetilde{D}_1) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_1)$. Then it holds that
$$(\mathsf{crs}, m_0) \overset{\mathrm{c}}{\approx} (\mathsf{crs}, m_1).$$
- **Sender Privacy:** There exists a PPT simulator $\mathsf{niscSim}$ such that for every database $D \in \{0,1\}^M$ where $M = \mathsf{poly}(\lambda)$ for any polynomial function $\mathsf{poly}(\cdot)$, and for every RAM program (P, x, t), let $y = P^D(x)$ be the output of the program, and $\mathsf{MemAccess}$ be the memory access pattern, then it holds that
$$(\mathsf{crs}, D, (m_1, \widetilde{D}), m_2) \overset{\mathrm{c}}{\approx} \mathsf{niscSim}(1^\lambda, D, (y, \mathsf{MemAccess}))$$
where $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$, $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$ and $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$.
- **Efficiency:** The length of m_1 is a fixed polynomial in λ independent of the size of the database. Moreover, the algorithm $\mathsf{EncData}$ runs in time $M \cdot \mathsf{poly}(\lambda, \log M)$, $\mathsf{EncProg}$ and Dec run in time $t \cdot \mathsf{poly}(\lambda, \log M)$.

6.3 Construction

Overview. We first give an overview of the construction. For ease of exposition, consider a read-only program where each CPU step outputs the next location to be read based on the value read from last location.

We first describe the $\mathsf{EncProg}$ procedure. As already mentioned in technical overview (see Section 2.2), our construction is based on high level ideas of garbled RAM (introduced by Lu and Ostrovsky [LO13]) to make sender and receiver complexity grow only with the running time of the program. In particular, the sender would generate a garbled RAM program consisting of a sequence of t garbled *step circuits*. Similar to the RAM computation model described in Section 6.1.1, every step circuit takes as input the current CPU state and the last read bit and outputs the updated state and the next read location, say L. Note that the next step circuit would take the new value read from database as input.

The main challenge in program garbling is revealing the correct labels for the next circuit based on the value of $D[L]$. Moreover, it is crucial for garble circuit security that the receiver does not learn the label corresponding to $1 - D[L]$. Prior works [LO13, GHL+14, GLOS15, GLO15] proposed several different solutions to the above problem. Here we present a new and arguably simpler solution for achieving this using laconic oblivious transfer.

Let digest be the hash value of D that would be fed into the first step circuit and passed along the sequence of circuits. That is, each circuit would take this digest as input and also output the correct input labels corresponding to the digest for the next circuit. Now, to transfer the correct label corresponding to the value in the database, a step circuit would output a laconic OT ciphertext (using algorithm Send) that encrypts the input keys of the next step circuit and uses digest as the hash value. Looking ahead, when the receiver evaluates the step circuit which outputs the laconic OT ciphertext, he can use D to decrypt it to obtain the correct labels (using the procedure $\mathsf{Receive}$ of laconic OT).

We would show that the sender privacy follows from the sender privacy of laconic OT and security of circuit garbling. In order to achieve receiver privacy, the receiver does not publish digest in the clear, but instead, the labels for digest of the first step circuit are transferred from the sender to the receiver via a two-message OT. In particular, the $\mathsf{EncData}$ procedure outputs the first OT message of digest, and $\mathsf{EncProg}$ will output the garbled step circuits along with the second OT message for digest's labels.

Finally, note that a general program can also write to the database, in which case we need to update the database as well as the step circuits need to know the updated digest for the correctness of laconic OT and future reads/writes. This is achieved via the updatability property of the laconic OT which allows a sender to generate a ciphertext that allows the receiver to learn messages corresponding to the updated digest. In our case, the messages encrypted would be the input digest keys of the next step circuit.

Next, we give a more formal construction of our scheme.

The Construction. Let $\ell OT = (\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive}, \mathsf{SendWrite}, \mathsf{ReceiveWrite})$ be an updatable laconic OT protocol as per Definition 3.2. Let $\mathsf{OT} = (\mathsf{OT}_1, \mathsf{OT}_2, \mathsf{OT}_3)$ be a two-message secure oblivious transfer, and let $\mathsf{GC} = (\mathsf{GCircuit}, \mathsf{Eval})$ be a circuit garbling scheme. The non-interactive secure RAM computation scheme $\mathsf{NISC\text{-}RAM} = (\mathsf{Setup}, \mathsf{EncData}, \mathsf{EncProg}, \mathsf{Dec})$ is constructed as follows.

Setup: $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$.
The set up algorithm is described in Figure 7. It generates the common reference string for the updatable laconic OT scheme.

Set up. $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$.

1. $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$.
2. Output crs.

Figure 7: Set up procedure of NISC-RAM

Database Encryption: $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$.
The algorithm is formally described in Figure 8. It hashes the database D using laconic OT Hash function and obtains digest. Then digest is encrypted using the OT_1 procedure of two-message OT protocol.

Database Encryption. $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$.

1. $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$.
2. $(m_1, \mathrm{secret}) \leftarrow \mathsf{OT}_1(1^\lambda, \mathsf{digest})$.
3. Output $\Big(m_1, \widetilde{D} = (\mathsf{digest}, \hat{\mathsf{D}}, \mathrm{secret})\Big)$.

Figure 8: Database encryption procedure of NISC-RAM

Program Encryption: $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$.
The program encryption procedure is formally described in Figure 9. As mentioned above, it generates t garbled *step circuits* $\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^t$, where every step circuit implements the functionality of a CPU-step circuit. We describe the structure of a step circuit C^{step} below. The program encryption also consists of the second OT message corresponding to the short message m_1 of the receiver (for digest) where the sender's messages consist of the input keys for the first garbled circuit. Finally, it also outputs the keys for decrypting the output of the last step circuit.

Now we elaborate on the logic of a step circuit. The pseudocode of a step circuit $\mathsf{C}^{\mathsf{step}}$ is formally described in Figure 10, and the structure is illustrated in Figure 11. The input of a step circuit can be partitioned into $(\mathsf{state}, \mathsf{rData}, \mathsf{digest})$, where state is the current CPU state, rData is the bit read from the database, and digest is the up-to-date digest of the database. If the previous step is a write, then $\mathsf{rData} = 0$. The program encryption outputs garbled circuits for these step circuits, hence, the first step of $\mathsf{EncProg}$ is to pick the input keys for all the circuits. The τ-th step circuit $\mathsf{C}_\tau^{\mathsf{step}}$ has hardwired in it the input keys $\mathsf{nextKeys} = (\mathsf{stateKeys}, \mathsf{dataKeys}, \mathsf{digestKeys})$ for the next step circuit $\mathsf{C}_{\tau+1}^{\mathsf{step}}$.

The logic of the step circuit is as follows: It first computes the new $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData})$. Then, in the case of a "read" it outputs $\mathsf{stateKeys}$ corresponding to state', labels for rData via laconic OT procedure $\mathsf{Send}(\cdot)$, and $\mathsf{digestKeys}$ corresponding to digest. The case of a write is similar, but now the labels of new updated digest are transferred via laconic OT procedure $\mathsf{SendWrite}(\cdot)$.

Program Encryption. $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$.

1. **Generate the garbled program for** P: Generate garbled circuits $\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^{t}$.
   (a) Sample $\mathsf{stateKeys}^\tau, \mathsf{dataKeys}^\tau, \mathsf{digestKeys}^\tau$ for each $\tau \in \{1, \ldots, t+1\}$.
   (b) For each $\tau \in \{1, \ldots, t\}$

$$\widetilde{\mathsf{C}}_\tau^{\mathsf{step}} \leftarrow \mathsf{GCircuit}\left(1^\lambda, \mathsf{C}^{\mathsf{step}}[\mathsf{crs}, P, \mathsf{Keys}^{\tau+1}], \mathsf{Keys}^\tau\right),$$

   where $\mathsf{Keys}^\tau = (\mathsf{stateKeys}^\tau, \mathsf{dataKeys}^\tau, \mathsf{digestKeys}^\tau)$.
   (c) For $\tau = 1$, embed labels $\mathsf{dataKeys}_0^1$ and $\mathsf{stateKeys}_x^1$ in $\widetilde{\mathsf{C}}_1^{\mathsf{step}}$.
2. Compute $\mathsf{L} \leftarrow \mathsf{OT}_2\left(m_1, \mathsf{digestKeys}^1\right)$.
3. Output $m_2 = \left(\mathsf{L}, \{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^{t}, \mathsf{stateKeys}^{t+1}\right)$.

Figure 9: Program encryption procedure of NISC-RAM

Hardwired Parameters: $[\mathsf{crs}, P, \mathsf{nextKeys} = (\mathsf{stateKeys}, \mathsf{dataKeys}, \mathsf{digestKeys})]$.
Input: $(\mathsf{state}, \mathsf{rData}, \mathsf{digest})$.

$(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) := C_{\mathsf{CPU}}^P(\mathsf{state}, \mathsf{rData})$.

if $\mathsf{R/W} = \mathsf{read}$ **then**
  $\mathsf{e}_{\mathsf{data}} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, \mathsf{dataKeys})$.
  return $\left((\mathsf{stateKeys}_{\mathsf{state}'}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestKeys}_{\mathsf{digest}}), \mathsf{R/W}, L\right)$.
else
  $\mathsf{e}_{\mathsf{digest}} \leftarrow \mathsf{SendWrite}(\mathsf{crs}, \mathsf{digest}, L, \mathsf{wData}, \mathsf{digestKeys})$.
  return $((\mathsf{stateKeys}_{\mathsf{state}'}, \mathsf{dataKeys}_0, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData}), \mathsf{R/W}, L)$.

Figure 10: Pseudocode of a step circuit $\mathsf{C}^{\mathsf{step}}[\mathsf{crs}, P, \mathsf{nextKeys}]$.

Decryption: $y \leftarrow \mathsf{Dec}^{\widetilde{D}}(\mathsf{crs}, m_2)$.
The decryption procedure is described in Figure 13. At a high level the receiver evaluates the garbled step circuits one by one from $\widetilde{\mathsf{C}}_1^{\mathsf{step}}$ to $\widetilde{\mathsf{C}}_t^{\mathsf{step}}$, and uses the database to decrypt ℓOT ciphertexts between two consecutive circuits. The output of the last step circuit can be decrypted using $\mathsf{stateKeys}^{t+1}$ and hence y is obtained.

More precisely, the receiver first obtains the $\mathsf{digestLabels}$ for the first step circuit by running OT_3. Note that the first garbled step circuit already has labels for the rData and state embedded. Hence the receiver can obtain all the labels for the first step circuit and evaluate it. Then the receiver executes the circuits $\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^{t}$ one by one, and learns the labels for the next circuit by running the receiver algorithms of laconic OT on its database.

6.4 Correctness

For correctness, we require that for every database $D \in \{0,1\}^M$, for every RAM program (P, x, t), it holds that

$$\Pr\left[\mathsf{Dec}^{\widetilde{D}}(\mathsf{crs}, m_2) = P^D(x)\right] = 1,$$

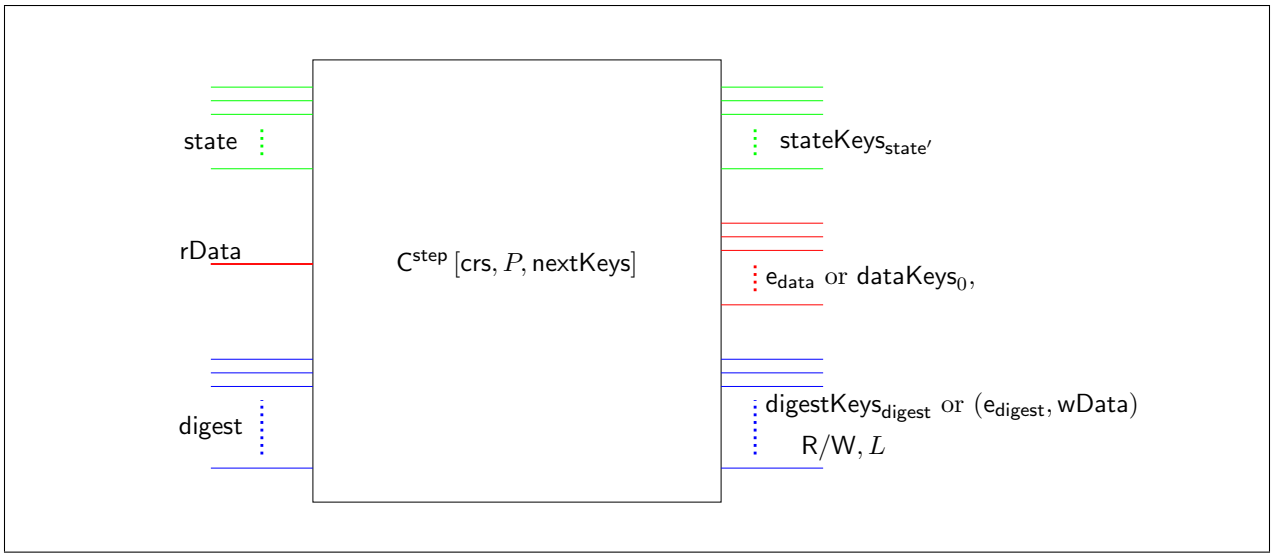


Figure 11: A step circuit $\mathsf{C}^{\mathsf{step}}[\mathsf{crs}, P, \mathsf{nextKeys}]$

where $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$, $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$, $m_2 \leftarrow \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))$. Correctness follows from Lemma 6.3 that we will prove below.

Claim 6.2. *The first garbled step circuit* $\widetilde{\mathsf{C}}_1^{\mathsf{step}}$ *gets evaluated on* $(x, 0, \mathsf{digest})$*, where* $(\mathsf{digest}, \hat{\mathsf{D}}) = \mathsf{Hash}(\mathsf{crs}, D)$*.*

Proof. Since $(m_1, \mathsf{secret}) \leftarrow \mathsf{OT}_1(1^\lambda, \mathsf{digest})$, $\mathsf{L} \leftarrow \mathsf{OT}_2\left(m_1, \mathsf{digestKeys}^1\right)$, and $\mathsf{digestLabels}^1 \leftarrow \mathsf{OT}_3(\mathsf{L}, \mathsf{secret})$, by correctness of OT, $\mathsf{digestLabels}^1 = \mathsf{digestKeys}^1_{\mathsf{digest}}$. Moreover, $\widetilde{\mathsf{C}}_1^{\mathsf{step}}$ already has labels $\mathsf{stateKeys}^1_x$ and $\mathsf{dataKeys}^1_0$ embedded in it, by correctness of the circuit garbling scheme, $\widetilde{\mathsf{C}}_1^{\mathsf{step}}$ gets evaluated on $(x, 0, \mathsf{digest})$. □

Lemma 6.3. *Consider the execution of* $P^D(x)$*. Let* $(\mathsf{state}^\tau, \mathsf{rData}^\tau)$ *be the input to the* τ*-th CPU step. Let* D^τ *be the database at the beginning of step* τ*, and let* $(\mathsf{digest}^\tau, \hat{\mathsf{D}}^\tau) = \mathsf{Hash}(\mathsf{crs}, D^\tau)$*. During the* Dec *procedure, for every* $\tau \in [t]$*,* $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$ *is evaluated on inputs* $(\mathsf{state}^\tau, \mathsf{rData}^\tau, \mathsf{digest}^\tau)$*. Moreover, the state of the database held by the receiver at the beginning of evaluating* $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$ *is* $\hat{\mathsf{D}}^\tau$*.*

Proof. We will prove this lemma by induction on τ. The base case follows from Claim 6.2. Assume that the lemma holds for $\tau = \rho$, then we prove that the lemma holds for $\rho + 1$ in the following. We know that $(\hat{\mathsf{D}}^\rho, \mathsf{digest}^\rho) = \mathsf{Hash}(\mathsf{crs}, D^\rho)$, and that $\widetilde{\mathsf{C}}_\rho^{\mathsf{step}}$ is executed on $(\mathsf{state}^\rho, \mathsf{rData}^\rho, \mathsf{digest}^\rho)$. By correctness of GC, $\widetilde{\mathsf{C}}_\rho^{\mathsf{step}}$ implements its code of a CPU step, namely $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) = C^P_{\mathsf{CPU}}(\mathsf{state}^\rho, \mathsf{rData}^\rho)$. Also notice that $\mathsf{nextKeys} = (\mathsf{stateKeys}, \mathsf{dataKeys}, \mathsf{digestKeys})$ hardwired in $\widetilde{\mathsf{C}}_\rho^{\mathsf{step}}$ are the input keys for $\widetilde{\mathsf{C}}_{\rho+1}^{\mathsf{step}}$. There are two cases:

- $\mathsf{R/W} = \mathsf{read}$: In this case, it follows directly from the Dec procedure that $\mathsf{stateLabels}^{\rho+1} = \mathsf{stateKeys}_{\mathsf{state}'}$ and $\mathsf{digestLabels}^{\rho+1} = \mathsf{digestKeys}_{\mathsf{digest}}$. Since $\mathsf{e}_{\mathsf{data}} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}^\rho, L, \mathsf{dataKeys})$ and $\mathsf{dataLabels}^{\rho+1} = \mathsf{Receive}^{\hat{\mathsf{D}}^\rho}(\mathsf{crs}, \mathsf{e}_{\mathsf{data}}, L)$, by correctness of the ℓOT scheme, $\mathsf{dataLabels}^{\rho+1} = \mathsf{dataKeys}_{D^\rho[L]}$. Hence $\widetilde{\mathsf{C}}_{\rho+1}^{\mathsf{step}}$ is evaluated on inputs $(\mathsf{state}', D^\rho[L], \mathsf{digest})$, which is exactly $(\mathsf{state}^{\rho+1}, \mathsf{rData}^{\rho+1}, \mathsf{digest}^{\rho+1})$. And $(\hat{\mathsf{D}}^\rho, \mathsf{digest}^\rho)$ remains unchanged.

Decryption. $y \leftarrow \mathsf{Dec}^{\widetilde{D}}(\mathsf{crs}, m_2)$.

1. Parse $\widetilde{D} = (\mathsf{digest}, \hat{\mathsf{D}}, \mathrm{secret})$.
2. Parse $m_2 = \left(\mathsf{L}, \{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^{t}, \mathsf{stateKeys}^{t+1}\right)$.
3. Compute $\mathsf{digestLabels}^1 \leftarrow \mathsf{OT}_3(\mathsf{L}, \mathrm{secret})$.
4. Parse $\widetilde{\mathsf{C}}_1^{\mathsf{step}} = (\widetilde{\mathsf{C}}_1^{\mathsf{step}}, \mathsf{dataLabels}^1, \mathsf{stateLabels}^1)$.
5. For $\tau = 1$ to t do the following:

   $(X, \mathsf{R/W}, L) := \mathsf{Eval}\left(\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, (\mathsf{stateLabels}^\tau, \mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau)\right)$.

   if $\mathsf{R/W} = \mathsf{read}$ **then**
   Parse $X = (\mathsf{stateLabels}^{\tau+1}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestLabels}^{\tau+1})$
   $\mathsf{dataLabels}^{\tau+1} = \mathsf{Receive}^{\hat{\mathsf{D}}}(\mathsf{crs}, \mathsf{e}_{\mathsf{data}}, L)$
   else
   Parse $X = (\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData})$
   $\mathsf{digestLabels}^{\tau+1} = \mathsf{ReceiveWrite}^{\hat{\mathsf{D}}}(\mathsf{crs}, L, \mathsf{wData}, \mathsf{e}_{\mathsf{digest}})$
6. Use $\mathsf{stateKeys}^{t+1}$ to decode $\mathsf{stateLabels}^{t+1}$ and obtain y.

Figure 13: Decryption procedure of NISC-RAM

- $\mathsf{R/W} = \mathsf{write}$: In this case, it follows from the Dec procedure that $\mathsf{stateLabels}^{\rho+1} = \mathsf{stateKeys}_{\mathsf{state}'}$ and $\mathsf{dataLabels}^{\rho+1} = \mathsf{dataKeys}_0$. Since $\mathsf{e}_{\mathsf{digest}} \leftarrow \mathsf{SendWrite}(\mathsf{crs}, \mathsf{digest}^\rho, L, \mathsf{wData}, \mathsf{digestKeys})$ and $\mathsf{digestLabels}^{\rho+1} = \mathsf{ReceiveWrite}^{\hat{\mathsf{D}}^\rho}(\mathsf{crs}, L, \mathsf{wData}, \mathsf{e}_{\mathsf{digest}})$, by correctness of the ℓOT scheme, $\mathsf{digestLabels}^{\rho+1} = \mathsf{digestKeys}_{\mathsf{digest}'}$ where $(\hat{\mathsf{D}}', \mathsf{digest}') = \mathsf{Hash}(\mathsf{crs}, D')$ for an updated database D' (D' is identical to D^ρ except that $D'[L] = \mathsf{wData}$). Hence $\widetilde{\mathsf{C}}_{\rho+1}^{\mathsf{step}}$ is evaluated on inputs $(\mathsf{state}', 0, \mathsf{digest}')$, which is exactly $(\mathsf{state}^{\rho+1}, \mathsf{rData}^{\rho+1}, \mathsf{digest}^{\rho+1})$. And $(\hat{\mathsf{D}}^\rho, \mathsf{digest}^\rho)$ gets updated to $(\hat{\mathsf{D}}', \mathsf{digest}')$, which is exactly $(\hat{\mathsf{D}}^{\rho+1}, \mathsf{digest}^{\rho+1})$.

□

6.5 Security Proof

In this section we prove sender privacy and receiver privacy as defined in Section 6.2 under the decisional Diffie-Hellman (DDH) assumption. The receiver privacy follows directly from the receiver security of OT. Below we prove sender privacy by describing a PPT simulator niscSim such that for every database $D \in \{0,1\}^M$ where M is polynomial in λ, and for every RAM program (P, x, t), let $y = P^D(x)$ be the output of the program, and MemAccess be the memory access pattern, then it holds that

$$\left(\mathsf{crs}, (m_1, \widetilde{D}), \mathsf{EncProg}(\mathsf{crs}, m_1, (P, x, t))\right) \overset{\mathrm{c}}{\approx} \left(\mathsf{crs}, (m_1, \widetilde{D}), \mathsf{niscSim}(\mathsf{crs}, m_1, D, y, \mathsf{MemAccess})\right),$$

where $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$, $(m_1, \widetilde{D}) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$. Notice that this definition is slightly different from the definition in Section 6.2, but in the semi-honest case it implies a simulator as defined in Section 6.2

1. Sample input keys $(\mathsf{stateKeys}^{t+1}, \mathsf{dataKeys}^{t+1}, \mathsf{digestKeys}^{t+1})$ for $\mathsf{C}^{\mathsf{step}}$.

2. Parse $\mathsf{MemAccess}$ as $\{(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau) : \tau \in [t]\}$, where $(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$ is partial output of the τ-th CPU step circuit. Compute $(\mathsf{rData}^\tau, D^\tau, \mathsf{digest}^\tau)$ at the beginning of step τ for every $\tau \in [t+1]$.

3. Compute $(\mathsf{stateLabels}^{t+1}, \mathsf{dataLabels}^{t+1}, \mathsf{digestLabels}^{t+1})$:
   $\mathsf{stateLabels}^{t+1} \leftarrow \mathsf{stateKeys}^{t+1}_{y}$.
   $\mathsf{digestLabels}^{t+1} \leftarrow \mathsf{digestKeys}^{t+1}_{\mathsf{digest}^{t+1}}$.
   $\mathsf{dataLabels}^{t+1} \leftarrow \mathsf{dataKeys}^{t+1}_{\mathsf{rData}^{t+1}}$.

4. For $\tau = t$ downto 1, proceed as follows:
   if $\mathsf{R/W}^\tau = \mathsf{read}$ **then**
   $\mathsf{e}_{\mathsf{data}} \leftarrow \ell\mathsf{OTSim}\left(\mathsf{crs}, D^\tau, L^\tau, \mathsf{dataLabels}^{\tau+1}\right)$.
   $X \leftarrow (\mathsf{stateLabels}^{\tau+1}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestLabels}^{\tau+1})$.
   else
   $\mathsf{e}_{\mathsf{digest}} \leftarrow \ell\mathsf{OTSimWrite}\left(\mathsf{crs}, D^\tau, L^\tau, \mathsf{wData}^\tau, \mathsf{digestLabels}^{\tau+1}\right)$.
   $X \leftarrow (\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData}^\tau)$.

   $\left(\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau, \mathsf{stateLabels}^\tau, \mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau\right) \leftarrow \mathsf{CircSim}\left(1^\lambda, \mathsf{C}^{\mathsf{step}}, (X, \mathsf{R/W}^\tau, L^\tau)\right)$.

5. $\mathsf{L} \leftarrow \mathsf{OT}_2\left(m_1, (\mathsf{digestLabels}^1, \mathsf{digestLabels}^1)\right)$.

6. Output $\left(\mathsf{L}, \{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau\}^t_{\tau=1}, \mathsf{stateKeys}^{t+1}\right)$.

We show that the above simulation is indistinguishable from the real execution through a sequence of hybrids where the first hybrid outputs the real execution and the last hybrid outputs the simulated one.

- H_{2i} for $i \in \{0, 1, \ldots, t\}$: Notice that in the output, there are t garbled step circuits $\{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau\}^t_{\tau=1}$. In hybrid H_{2i}, the garbled step circuits from 1 to i are simulated while the remaining step circuits ($i+1$ to t) are generated honestly. Given the program, all the intermediate outputs of every step circuit can all be computed. Given the correct output of circuit $\mathsf{C}^{\mathsf{step}}_i$, the step circuits from 1 to i can be simulated one by one from the i-th to the first similarly as $\mathsf{niscSim}$. More formally, it proceeds as follows.

  1. Execute $P^D(x)$ to obtain $(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$ for every $\tau \in [t]$ and $\mathsf{state}^{t+1} = y$. Compute $(\mathsf{rData}^\tau, D^\tau, \mathsf{digest}^\tau)$ at the beginning of step τ for every $\tau \in [t+1]$.
  2. Generate the garble circuits $\{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau\}^t_{\tau=i+1}$ honestly (same as Step 1 in $\mathsf{EncProg}$).
  3. Let $\left(\mathsf{stateKeys}^{i+1}, \mathsf{dataKeys}^{i+1}, \mathsf{digestKeys}^{i+1}\right)$ be the input keys of $\widetilde{\mathsf{C}}^{\mathsf{step}}_{i+1}$.
  4. Compute $(\mathsf{stateLabels}^{i+1}, \mathsf{dataLabels}^{i+1}, \mathsf{digestLabels}^{i+1})$:
     $\mathsf{stateLabels}^{i+1} \leftarrow \mathsf{stateKeys}^{i+1}_{\mathsf{state}^{i+1}}$.
     $\mathsf{digestLabels}^{i+1} \leftarrow \mathsf{digestKeys}^{i+1}_{\mathsf{digest}^{i+1}}$.
     $\mathsf{dataLabels}^{i+1} \leftarrow \mathsf{dataKeys}^{i+1}_{\mathsf{rData}^{i+1}}$.

5. For $\tau = i$ downto 1, proceed as in Step 4 of the simulator $\mathsf{niscSim}$.
6. $\mathsf{L} \leftarrow \mathsf{OT}_2\left(m_1, (\mathsf{digestLabels}^1, \mathsf{digestLabels}^1)\right)$.
7. Output $\left(\mathsf{L}, \{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=1}^t, \mathsf{stateKeys}^{t+1}\right)$.

- H_{2i+1} for $i \in \{0, \ldots, t-1\}$: Hybrid H_{2i+1} is identical to H_{2i} except that H_{2i+1} simulates $\widetilde{\mathsf{C}}_{i+1}^{\mathsf{step}}$ based on the real output of $\mathsf{C}_{i+1}^{\mathsf{step}}$. In particular, H_{2i+1} is the same as H_{2i} except that Steps 2, 3, 4 proceed as follows:

   2. Generate the garble circuits $\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\}_{\tau=i+2}^t$ honestly (same as Step 1 in $\mathsf{EncProg}$).
   3. Let $\left(\mathsf{stateKeys}^{i+2}, \mathsf{dataKeys}^{i+2}, \mathsf{digestKeys}^{i+2}\right)$ be the input keys of $\widetilde{\mathsf{C}}_{i+2}^{\mathsf{step}}$.
   4. **if** $\mathsf{R/W}^{i+1} = \mathsf{read}$ **then**
      $\mathsf{e}_{\mathsf{data}} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}^{i+1}, L^{i+1}, \mathsf{dataKeys}^{i+2})$.
      $X \leftarrow (\mathsf{stateKeys}^{i+2}_{\mathsf{state}^{i+2}}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestKeys}^{i+2}_{\mathsf{digest}})$.
      else
      $\mathsf{e}_{\mathsf{digest}} \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, \mathsf{digest}^{i+1}, L^{i+1}, \mathsf{wData}^{i+1}, \mathsf{digestKeys}^{i+2}\right)$.
      $X \leftarrow (\mathsf{stateKeys}^{i+1}_{\mathsf{state}^{i+1}}, \mathsf{dataKeys}_0^{i+1}, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData}^{i+1})$.
      $\left(\widetilde{\mathsf{C}}_{i+1}^{\mathsf{step}}, \mathsf{stateLabels}^{i+1}, \mathsf{dataLabels}^{i+1}, \mathsf{digestLabels}^{i+1}\right) \leftarrow \mathsf{CircSim}\left(1^\lambda, \mathsf{C}^{\mathsf{step}}, (X, \mathsf{R/W}^{i+1}, L^{i+1})\right)$.

It is easy to see that H_0 is the output of the real execution, and H_{2t} is the simulated output. Now we prove that the consecutive hybrids are computationally indistinguishable. Below we prove that $\mathsf{H}_{2i} \overset{c}{\approx} \mathsf{H}_{2i+1} \overset{c}{\approx} \mathsf{H}_{2(i+1)}$ for every $i \in \{0, \ldots, t-1\}$. Since hybrid H_{2i+1} simulates $\widetilde{\mathsf{C}}_{i+1}^{\mathsf{step}}$ based on the real output of $\mathsf{C}_{i+1}^{\mathsf{step}}$, the output of $\widetilde{\mathsf{C}}_{i+1}^{\mathsf{step}}$ is identical for hybrids H_{2i} and H_{2i+1}. That said, indistinguishibility of hybrids H_{2i} and H_{2i+1} follows from the garbled circuit security. Next, indistinguishability between H_{2i+1} and $\mathsf{H}_{2(i+1)}$ follows from the sender's privacy property of the updatable laconic OT since the laconic OT responses are simulated in $\mathsf{H}_{2(i+1)}$. This concludes the proof.

6.6 Extension

For simplicity of exposition, the protocol we described so far is for a single sender executing a single program with the receiver. It can be extended to the setting where a sender can execute a sequence of programs on a persistent database. Moreover, the message m_1 published by the receiver can be used by multiple senders, in which case the receiver maintains a different copy of the database for every sender.

Executing multiple programs on a persistent database. After receiving the first message m_1 from the receiver, a sender can run multiple programs on a persistent database (with initial content D) by sending one message per program to the receiver. For security we require only the output and the memory access pattern of every program execution are revealed to the receiver. We also require that once m_1 is published, the computational cost of both the sender and the receiver for every program should grow only with the running time of the RAM computation, and is independent of the size of D. The $\mathsf{NISC\text{-}RAM}$ scheme we constructed in Section 6.3 can be naturally extended to the multi-program setting. We explain the extension by describing the changes of $\mathsf{EncProg}$ and Dec procedures for the second program. Encryption and evaluation of more programs would follow analogously.

- EncProg: When encrypting the first program, the sender should store locally $\mathsf{digestKeys}^* = \mathsf{digestKeys}^{t+1}$. Then, when encrypting the second program, there are two changes in EncProg compared to encrypting the first program: (1) $\mathsf{digestKeys}^*$ is used as the digest keys of the first step circuit, (2) L is not generated.

- Dec: When evaluating the first program, the receiver should store locally $\mathsf{digestLabels}^* = \mathsf{digestLabels}^{t+1}$. Then, when evaluating the second program, the sender should use $\mathsf{digestLabels}^*$ as the digest labels for the first step circuit.

Multiple senders with a single receiver. The above protocol also works for multiple parallel senders. That is, after the receiver publishes the first message m_1, every sender S can send a message m_S to the receiver enabling the execution of $P_S^D(x_S)$, where D is the initial database of the receiver, and (P_S, x_S) is the program of S. Security follows from the security of OT which supports multiple second OT messages with the same first OT message. Moreover, every sender can execute a sequence of programs on a persistent database. In this case, the receiver keeps a different copy of her initial database for every sender.

7 Main Application: Multi-Hop Homomorphic Encryption for RAM Programs

7.1 Our Model

Consider a server S and a collection of clients $Q_1, Q_2, \ldots$ with private databases $D_1, D_2, \ldots$, respectively. The clients ship their encrypted databases to S to be computed on later in multiple executions in a persistent manner. At the beginning of any execution, the server S encrypts his private input x as ct_0, chooses a subset of clients $Q_{i_1}, \ldots, Q_{i_n}$ and sends the ct_0 to client Q_{i_1}. Next, for all $j \in [n]$, client Q_{i_j} homomorphically evaluates an arbitrary program P_j of his choice on ct_{j-1} to obtain ct_j. Finally, client Q_{i_n} sends ct_n to the server S. The server decrypts this ciphertext using his secret key of encryption as well as encrypted databases sent earlier to learn $P_n^{D_{i_n}}\left(\ldots P_1^{D_{i_1}}(x)\ldots\right)$. During this execution, the databases get updated and future execution of any client happens on respective updated databases.

We require that the size of the ciphertext only grows with the cumulative running time of all programs in an execution and is independent of the size of the databases. For security, we require program and data privacy for all honest clients against an adversary that corrupts the server and any subset of the clients. Next, we describe the model formally.

We say that an ordered sequence of RAM programs $P_1, \cdots, P_n$ are *compatible* if the output length of P_i is the same as the input length of P_{i+1} for every $i \in [n-1]$. A multi-hop RAM homomorphic encryption scheme $\mathsf{mhop\text{-}RAM} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{InpEnc}, \mathsf{EncData}, \mathsf{Eval}, \mathsf{Dec})$ has the following syntax. We define the algorithms w.r.t. clients $Q_1, \ldots, Q_n$.

- **Setup:** $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$.
  On input the security parameter 1^λ, it outputs a common reference string.

- **Key Generation:** $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$.
  On input the security parameter 1^λ, it outputs a public/secret key pair $(\mathsf{pk}, \mathsf{sk})$.

- **Database Encryption:** $(\widetilde{D} = (\hat{D}, \mathsf{digest})) \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$.
  On input the common reference string crs and database $D \in \{0,1\}^M$, it outputs an encrypted database $\widetilde{D} = (\hat{D}, \mathsf{digest})$, where digest is a short digest of the database.

- **Input Encryption:** $(\mathsf{ct}, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$.
  On input S's input x, it outputs a ciphertext ct and secret state for S denoted by x_secret.

- **Homomorphic Evaluation:** $\mathsf{ct}' \leftarrow \mathsf{Eval}(\mathsf{crs}, i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}, \mathsf{sk}, (P,t), \mathsf{digest})$.
  It takes as input the crs, the client number i, the public keys of the clients later in the sequence, i.e., $Q_{i+1}, \cdots, Q_n$, a ciphertext from the previous client, Q's secret key sk, Q's RAM program P with maximum run-time t and the digest digest of the database D of Q. It then outputs a new ciphertext ct'.

- **Decryption:** $y = \mathsf{Dec}^{\widetilde{D}_1, \cdots, \widetilde{D}_n}(\mathsf{crs}, \mathrm{x\_secret}, \mathsf{ct})$.
  On input the crs, server's state x_secret, the final ciphertext ct from client Q_n, and RAM access to encrypted databases $\widetilde{D}_1, \cdots, \widetilde{D}_n$, it outputs y. The procedure Dec is itself modeled as a RAM program that can read and write to arbitrary locations of its database initially containing $\widetilde{D}_1, \cdots, \widetilde{D}_n$.

Next, we describe how these algorithms are used in a real execution.

Real Scenario. In our multi-hop scheme for RAM programs, after the initialization phase that generates the common-random string crs, each cleint runs key generation to generate the public key and the secret key, followed by the database encryption. The encrypted database is sent to the server, and the cleint stores the digest of the database locally. After this initialization phase, the server S can initiate various executions of RAM computations with different subsets of the clients. After each execution, the database of a client gets updated by the server during the decryption phase. It is ensured that the server also learns the updated digest of the database that is communicated to the clients during the start of the next execution.

At the onset of any execution, the server S encrypts his input and sends the ciphertext ct_0 to the first client Q_1 and maintains x_secret to be used later. The client Q_1 generates the ciphertext ct_1 using his program P_1 and digest digest_1 and sends it to Q_2. Similarly, when a client Q_i receives ct_{i-1} from Q_{i-1}, it uses program P_i and digest_i to generate ct_i and sends it to Q_{i+1}. This continues and finally, Q_n sends ct_n back to the server S. Then, the server runs the decryption procedure on ct_n using all the encrypted databases and secret state x_secret to obtain output y.

For the case of multiple executions, each of the above procedures take the session identifier sid as additional input. We denote by $\widetilde{D}_1^{(\mathsf{sid})}, \cdots, \widetilde{D}_n^{(\mathsf{sid})}$ the encrypted databases before the execution with session identifier sid. Initially $\widetilde{D}_1^{(1)} = \widetilde{D}_1, \cdots, \widetilde{D}_n^{(1)} = \widetilde{D}_n$.

We require the algorithms above to satisfy the correctness, sender-privacy, client-privacy and efficiency properties described below.

Correctness. We require that in a sequence of executions, each output of homomorphic evaluation equals the output of the corresponding computation in the clear. We formalize this as follows: For every set of keys $\{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i=1}^n$ in the support of KeyGen, and any collection of initial databases $D_1, \cdots, D_n$, for any unbounded polynomial N number of executions the following holds:

For $\mathsf{sid} \in \mathsf{N}$, let $P_1^{(\mathsf{sid})}, \ldots, P_n^{(\mathsf{sid})}$ be the sequence of programs, $x^{(\mathsf{sid})}$ be the server's input and $D_i^{(\mathsf{sid})}$ be the resulting database after executing the session sid-1 in the clear, then

$$\mathsf{Pr}\left[\mathsf{Dec}^{\widetilde{D}_1^{(\mathsf{sid})},\cdots,\widetilde{D}_n^{(\mathsf{sid})}}\left(\mathsf{crs}, \mathrm{x\_secret}^{(\mathsf{sid})}, \mathsf{ct}_n^{(\mathsf{sid})}\right) = {P_n^{(\mathsf{sid})}}^{D_n^{(\mathsf{sid})}}\left(\cdots\left({P_1^{(\mathsf{sid})}}^{D_1^{(\mathsf{sid})}}(x^{(\mathsf{sid})})\right)\cdots\right)\right] = 1,$$

where $\widetilde{D}_i^{(\mathsf{sid})}$ is the resulting garbled database after executing sid-1 homomorphic evaluations, $(\mathsf{ct}_0^{(\mathsf{sid})}, \mathrm{x\_secret}^{(\mathsf{sid})}) \leftarrow \mathsf{InpEnc}(x^{(\mathsf{sid})}), \mathsf{ct}_i^{(\mathsf{sid})} \leftarrow \mathsf{Eval}(\mathsf{crs}, i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}_{i-1}^{(\mathsf{sid})}, \mathsf{sk}, P_i^{(\mathsf{sid})}, t_i^{(\mathsf{sid})}, \mathsf{digest}_i^{(\mathsf{sid})})$.

Server Privacy (Semantic Security). For server privacy, we require that for every pair of inputs (x_0, x_1), let $(\mathsf{CT}_b, \mathrm{x\_secret}^b) \leftarrow \mathsf{InpEnc}(x_b)$ for $b \in \{0,1\}$, then

$$\mathsf{CT}_0 \overset{\mathrm{c}}{\approx} \mathsf{CT}_1.$$

Client Privacy (Program Privacy) with Unprotected Memory Access (UMA). We define client privacy against a semi-honest adversary that corrupts the server S as well as an arbitrary subset of clients $\mathcal{I} \subset [n]$. Intuitively, we require *program-privacy* for the honest clients such that the adversary cannot learn anything beyond the output of the honest client's program on one input. We formalize this as follows:

There exists a PPT simulator $\mathsf{ihopSim}$ such that the following holds. Let $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$, for every set of keys $\{(\mathsf{pk}_i, \mathsf{sk}_i)\}_{i=1}^n$ in the support of KeyGen, and any collection of initial databases $D_1, \cdots, D_n$, for any unbounded polynomial N number of executions: For $\mathsf{sid} \in \mathsf{N}$, let $P_1^{(\mathsf{sid})}, \ldots, P_n^{(\mathsf{sid})}$ be the sequence of programs, $x^{(\mathsf{sid})}$ be the server's input, then

$$\left(\mathsf{crs}, (\widetilde{D}_1, \ldots, \widetilde{D}_n), \left\{\mathsf{ct}_0^{(\mathsf{sid})}, \mathsf{ct}_1^{(\mathsf{sid})}, \ldots, \mathsf{ct}_n^{(\mathsf{sid})}\right\}_{\mathsf{sid}\in[\mathsf{N}]}\right) \overset{\mathrm{c}}{\approx}$$
$$\left(\mathsf{crs}, \mathsf{ihopSim}\left(\mathsf{crs}, \left\{\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}, \left(\{D_j, P_j^{(\mathsf{sid})}\}_{j\in\mathcal{I}}, x^{(\mathsf{sid})}\right), \left\{D_j, \mathsf{MemAccess}_j^{(\mathsf{sid})}, y_j^{(\mathsf{sid})}\right\}_{j\in[n]\setminus\mathcal{I}}\right\}_{\mathsf{sid}\in[\mathsf{N}]}\right)\right)$$

where $\widetilde{D}_i, \mathsf{ct}_i^{(\mathsf{sid})}$ corresponds to outputs in the real execution given all the programs and databases and $y_j^{(\mathsf{sid})} = {P_j^{(\mathsf{sid})}}^{D_j^{(\mathsf{sid})}}\left(\cdots\left({P_1^{(\mathsf{sid})}}^{D_1^{(\mathsf{sid})}}(x^{(\mathsf{sid})})\right)\cdots\right)$.

Remark 7.1. *We note that the above definition also captures the security against a semi-malicious adversary who may choose his randomness for* KeyGen *maliciously but behaves honestly in the protocol.*

Client Privacy (Program Privacy) with Full Security. In the case of full client privacy, the simulator $\mathsf{ihopSim}$ does not get the database and the access pattern for the honest clients. That is, the simulator $\mathsf{ihopSim}$ takes as input $\left\{\{\mathsf{pk}_i, \mathsf{sk}_i\}_{i\in[n]}, (\{D_j, P_j^{(\mathsf{sid})}\}_{j\in\mathcal{I}}, x^{(\mathsf{sid})}), \left\{1^{M_j}, 1^{t_j^{(\mathsf{sid})}}, y_j^{(\mathsf{sid})}\right\}_{j\in[n]\setminus\mathcal{I}}\right\}_{\mathsf{sid}\in[\mathsf{N}]}$, where M_j is the size of D_j and $t_j^{(\mathsf{sid})}$ is the running time of $P_j^{(\mathsf{sid})}$.

Efficiency. We require the following efficiency guarantees from $\mathsf{mhop\text{-}RAM}$. Let $M_i = |D_i|$.

- $|\widetilde{D}_i| = M_i \cdot \mathsf{poly}(\lambda, \log M_i)$ for all $i \in [n]$.
- $|\mathsf{ct}_0| = |x| \cdot \mathsf{poly}(\lambda)$, where ct_0 is the output of $\mathsf{InpEnc}(x)$.
- $|\mathsf{ct}_i| = \sum_{j=1}^{i} n \cdot t_j \cdot \mathsf{poly}(\lambda, \log M_j, \log t_j)$ for all $i \in [n]$.

Extension. This definition (and our construction) can be extended to the setting where in each execution all the clients do not necessarily join the homomorphic evaluation. We allow for different set of clients to participate in different executions. In particular, before the first execution, the initial database of every client is encrypted. Later before each execution, a sequence of distinct clients $\langle i_1, \cdots, i_m \rangle$ can be specified.

The input encryption is the same as above, while the homomorphic evaluation is executed in the specified order (as specified by the server) as $\mathsf{ct}_j \leftarrow \mathsf{Eval}(\mathsf{crs}, j, \{\mathsf{pk}_{i_u}\}_{u=j+1}^{n}, \mathsf{sk}_{i_j}, \mathsf{ct}_{j-1}, P_{i_j}, t_{i_j}, \mathsf{digest}_{i_j})$ for every $j \in [m]$. And the decryption is executed as $y = \mathsf{Dec}^{\widetilde{D}_{i_1}, \cdots, \widetilde{D}_{i_m}}(\mathsf{crs}, \mathrm{x\_secret}, \mathsf{ct}_m)$, where $\widetilde{D}_{i_1}, \cdots, \widetilde{D}_{i_m}$ are the up-to-date garbled databases of clients $Q_{i_1}, \ldots, Q_{i_m}$. The correctness and privacy properties can be naturally extended to this setting.

7.2 Building Blocks Needed

In this section we introduce building blocks needed in our construction. In addition to these building blocks, we will also need RAM computation model (see Section 6.1.1) and two-message oblivious transfer (see Section 6.1.2). We use $[n]$ to denote the set $\{1, \ldots, n\}$.

7.2.1 2-message Secure Function Evaluation based on Garbled Circuits

A two-message secure function evaluation (SFE) based on garbled circuits is as follows: Let $\mathcal{U}(\cdot, \cdot)$ be a particular "universal circuit evaluator" that takes as input the description of a circuit C and an argument x, and returns $\mathcal{U}(C, x)$. We write $C(x)$ as a shorthand for $\mathcal{U}(C, x)$. Let Alice be the client with private input x and Bob have private input a circuit C. The protocol is as follows:

1. $(m_1, \mathrm{x\_secret}) \leftarrow \mathsf{SFE}_1(x)$: Alice computes $(m_1, \mathrm{x\_secret}) \leftarrow \mathsf{OT}_1(x)$ and sends m_1 to Bob.
2. $m_2 \leftarrow \mathsf{SFE}_2(C, m_1)$: Bob computes $\widetilde{\mathsf{C}} \leftarrow \mathsf{GCircuit}\left(\mathsf{C}, \{\mathsf{key}_b^w\}_{w \in \mathsf{inp(C)}, b \in \{0,1\}}\right)$ and $\mathsf{L} \leftarrow \mathsf{OT}_2\left(m_1, \{\mathsf{key}_b^w\}_{w \in \mathsf{inp(C)}, b \in \{0,1\}}\right)$. Sends $m_2 := (\widetilde{\mathsf{C}}, \mathsf{L})$.
3. $y = \mathsf{SFE}_{\mathsf{out}}(\mathrm{x\_secret}, m_2)$: Alice locally computes the output: $\{\mathsf{key}_{x_w}^w\}_{w \in \mathsf{inp(C)}} = \mathsf{OT}_3(\mathsf{L}, \mathrm{s\_secret})$, and $y = \mathsf{Eval}\left(\widetilde{\mathsf{C}}, \{\mathsf{key}_{x_w}^w\}_{w \in \mathsf{inp(C)}}\right)$.

The correctness of the above protocol follows from the correctness of Yao garbled circuits. It can be shown that the above protocol is a secure function evaluation protocol satisfying both semi-honest client privacy and semi-honest server privacy.

7.2.2 Re-Randomizable Secure Function Evaluation based on Garbled Circuits

[GHV10] defined the tool of "re-randomizable secure function evaluation" that was used to realize multi-hop homomorphic computation for circuits. This tool was constructed under the DDH assumption by instantiating Yao's garbled circuits with a special encryption scheme (BHHO [BHHO08]) and using re-randomizable two-message oblivious transfer [NP01].

Definition 7.2. *A secure function evaluation protocol is said to be re-randomizable if there exists an efficient procedure* $\mathsf{Re\text{-}rand}$ *such that for every input* x *and function* f *and every* $(m_1, x\_secret) \in \mathsf{SFE}_1(x)$ *and* $m_2 \in \mathsf{SFE}_2(C, m_1)$*, the distributions* $\{x, C, m_1, x\_secret, m_2, \mathsf{Re\text{-}rand}(m_1, m_2)\}$ *and* $\{x, C, m_1, x\_secret, m_2, \mathsf{SFE}_2(C, m_1)\}$ *are computationally indistinguishable.*

[GHV10] proved the following:

Theorem 7.3 ([GHV10])**.** *Under the DDH assumption, there exists a re-randomizable secure function evaluation protocol satisfying Definition 7.2.*

Below, we abstract out the scheme of [GHV10] by stating some of the procedures implicitly provided by [GHV10] that will be needed for this paper.

Definition 7.4 (Re-randomizable Yao garbled circuits.)**.** *The scheme in [GHV10] provides the following algorithms (implicitly) for their re-randomizable Yao scheme.*

1. $\mathsf{Keys} = \mathsf{SampleKeys}(1^\lambda, \mathsf{W}; r)$*: Takes as input a set of input wires* W *as well randomness* r *and outputs the input-keys for set of wires* W *for re-randomizable Yao. Note that it is a deterministic function given the randomness* r*. When clear from context, we will skip mentioning the inputs in the calls to this function.*

2. $\widetilde{\mathsf{C}} \leftarrow \mathsf{ReGCircuit}(C, \mathsf{InpKeys})$*: Takes as input a circuit* C *and* $\mathsf{InpKeys}$ *for the input wires of* C *and outputs a re-randomizable garbled circuit* $\widetilde{\mathsf{C}}$ *where input wires have keys as* $\mathsf{InpKeys}$*.*

3. $\widetilde{\mathsf{C}}' \leftarrow \mathsf{ReGCircuit}'(C, \mathsf{InpKeys}, \mathsf{OutKeys})$*: Takes as input a circuit* C*,* $\mathsf{InpKeys}$ *for input wires of* C *and* $\mathsf{OutKeys}$ *for output wires of* C*, and outputs a re-randomizable garbled circuit* $\widetilde{\mathsf{C}}$ *where input wires have keys as* $\mathsf{InpKeys}$ *and output wires have keys as* $\mathsf{OutKeys}$*.*

4. $\mathsf{Keys}^\dagger = \mathsf{Transform}(\mathsf{Keys}, r)$*: Takes as input* Keys *and randomness* r *and outputs randomized keys* $\mathsf{Keys}^\dagger$*. Also, we use* $\mathsf{Transform}(\mathsf{Keys}, \{r_1, \ldots, r_k\})$ *to denote*

$$\mathsf{Transform}(\mathsf{Transform}(\ldots(\mathsf{Transform}(\mathsf{Keys}, r_1), \ldots), r_{k-1}), r_k)$$

5. $(\widetilde{\mathsf{C}}', \mathsf{L}') \leftarrow \mathsf{Re\text{-}rand}\left((\widetilde{\mathsf{C}}, \mathsf{L}), \{r_w\}_{w \in \mathsf{Wires}(C)}\right)$*: Takes as input a re-randomizable garbled circuit* $\widetilde{\mathsf{C}}$ *and OT second messages* L *for the keys of input wires of* C *and randomness to re-randomize each wire of* C *and outputs a new functionally equivalent re-randomizable garbled circuit* $\widetilde{\mathsf{C}}'$ *and consistent OT second messages* L'*. This procedure satisfies the property of re-randomizable SFE. Moreover, the guarantee is that after randomization, for any wire* w*, the new keys for* w *in* $\widetilde{\mathsf{C}}'$ *are* $\mathsf{Transform}(\mathsf{key}^w, r_w)$*. Finally, re-randomization of OT messages only requires*[11] $\{r_w\}_{w \in \mathsf{inp}(\mathsf{C})}$*.*

For our multi-hop homomorphic scheme for RAM it will be useful to define $\mathsf{SampleKeys}(\cdot)$ as follows: $\mathsf{SampleKeys}(1^\lambda, \mathsf{W}, r) = \mathsf{Transform}(\mathsf{SampleKeys}(1^\lambda, \mathsf{W}, 0^\star), r)$.

[11] In fact, each OT message for keys of a wire can be randomized consistently just given the randomness used for that wire.

7.3 Our Construction of Multi-hop RAM Scheme

Below, in Section 7.3.1 we provide our construction for multi-hop scheme for RAM satisfying UMA-security for clients (see Section 7.1). We give a general transformation for UMA to full security in Section 7.3.5.

7.3.1 UMA Secure Construction

In this section, we first describe the UMA-secure scheme for one execution and then explain how this scheme can be extended naturally for multiple executions in Section 7.3.3. Also, as we shall see our scheme can easily be extended to the setting where different subset of parties participate in each session.

UMA-secure multi-hop RAM scheme for a single execution involving all parties. Let $Q_1, \ldots, Q_n$ be the clients holding databases $D_1, \ldots, D_n$, respectively, and S be the server. Let the server's private input be x and secret programs of clients be $P_1, \ldots, P_n$, respectively. Let $\ell OT = (\mathsf{crsGen}, \mathsf{Commit}, \mathsf{Send}, \mathsf{Receive}, \mathsf{SendWrite}, \mathsf{ReceiveWrite})$ be an updatable laconic OT scheme with sender privacy as defined in Definition 3.2. Let $\mathsf{Re\text{-}GC} = (\mathsf{SampleKeys}, \mathsf{ReGCircuit}, \mathsf{ReGCircuit}', \mathsf{Transform}, \mathsf{Re\text{-}rand})$ be a re-randomizable scheme for Yao's garbled circuits given by [GHV10] (see Definition 7.4). Let $\mathsf{OT} = (\mathsf{OT}_1, \mathsf{OT}_2, \mathsf{OT}_3)$ be a two-message oblivious transfer protocol as defined in Section 6.1.2.

The multi-hop RAM scheme $\mathsf{mhop\text{-}RAM} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{EncData}, \mathsf{InpEnc}, \mathsf{Eval}, \mathsf{Dec})$ is as follows: The algorithms $\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{EncData}$ are formally described in Figure 14.

Setup: $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$
Setup algorithm generates the common reference string for laconic OT.

Key Generation: $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
Each client runs this algorithm once to generate the secret-key sk and public-key pk. A client Q picks a PRF seed s as the secret key. Next, it generates the public key as the first message of OT for s and secret-key as the secret state for OT as well as PRF key.

Looking ahead, the client Q will use the PRF key s to garble his own P and to randomize the garbled program generated by all previous clients in any execution.

Database Encryption: $\widetilde{D} \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$
Each client runs this algorithm at the beginning to garble the database and sends the garbled database to the server S. The garbled database is generated by executing the Hash procedure of laconic OT. This outputs an encoded database $\hat{\mathsf{D}}$ and a digest digest, both of which are given to the server S.

Input Encryption: $(\mathsf{ct}, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$
In each execution, the server S encrypts its input x as follows: It computes the first message of OT as the ciphertext and stores the secret state of OT to be used for decryption of computation later. The ciphertext is sent the first client, w.l.o.g. Q_1. The algorithm InpEnc is described formally in Figure 15.

Set up. $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$.

1. Sample $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$.
2. Output crs.

Key Generation. $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$.

1. Sample a PRF key $s \leftarrow \{0,1\}^\lambda$ and generate $(\mathsf{pk}, \mathsf{s\_secret}) \leftarrow \mathsf{OT}_1(s)$.
2. Output $(\mathsf{pk}, \mathsf{sk} := (s, \mathsf{s\_secret}))$.

Database Encryption. $\widetilde{D} \leftarrow \mathsf{EncData}(\mathsf{crs}, D)$.

1. $(\mathsf{digest}, \hat{\mathsf{D}}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$.
2. Output $\widetilde{D} = (\mathsf{digest}, \hat{\mathsf{D}})$.

Figure 14: Formal description of the set up, key generation and database encryption algorithms.

Input Encryption. $(\mathsf{ct}, \mathsf{x\_secret}) \leftarrow \mathsf{InpEnc}(\mathsf{crs}, x)$.

1. $(\mathsf{ct}, \mathsf{x\_secret}) \leftarrow \mathsf{OT}_1(x)$.

Figure 15: Input encryption algorithm.

Homomorphic Evaluation: $\mathsf{ct}' \leftarrow \mathsf{Eval}\left(\mathsf{crs}, i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}, \mathsf{sk} = (s, \mathsf{s\_secret}), (P, t), \mathsf{digest}\right)$. This algorithm is executed by client Q_i to generate the next ciphertext ct' given ct from client Q_{i-1}, and is described formally in Figure 16. This is the most involved procedure in our construction, and hence, we first provide an informal description. At a very high level, as illustrated in Figure 17, the client Q_i generate the garbled program for P consisting of t garbled circuits and also re-randomize all the circuits in ct. As mentioned before, this re-randomization step is crucial to get program privacy for this client. Moreover, the re-randomization has to be done carefully so that the previous ct is consistent with the new garbled program.[12]

This procedure consists of four main steps: Let T be the number of step-circuits in ct.

1. Garble the new program P: For each $\tau \in [T+1, T+t]$, client does the following: It generates a "super-circuit" that is illustrated in Figure 18 consisting of a CPU step circuit $\mathsf{C}_\tau^{\mathsf{step}}$ (see Figure 19) and PRF circuits $\mathsf{C}_{\tau,i+1}^{\mathsf{PRF}}, \ldots, \mathsf{C}_{\tau,n}^{\mathsf{PRF}}$ (see Figure 20). A step circuit, encodes the logic of a CPU step of a program P and PRF circuits provide a part of the randomness used in re-randomization. We will elaborate on the functionality of PRF circuits later.

   The garbled program will consists of garbled circuits corresponding to all the step circuits and PRF circuits. The first step is to pick the keys for the input wires of all of these circuits. Next, we begin by describing the step circuits.

   Step Circuits $\mathsf{C}_\tau^{\mathsf{step}}$ (Figure 19) : The inputs of a step circuit (see Figure 18) can be partitioned into $((\mathsf{state}, \mathsf{rData}, \mathsf{digest}), \mathsf{Rd})$, where state is the current CPU state, rData is the bit-read from database, and digest is the up-to-date digest of the database. Rd corresponds to the randomness given as input to the step-circuit computed from the PRF circuits. A step

[12] We do this by keeping track of the randomness used in randomizing the input wires for each garbled circuit.

Homomorphic Evaluation.

$$\mathsf{ct}' \leftarrow \mathsf{Eval}\left(\mathsf{crs}, i, \{\mathsf{pk}_j\}_{j=i+1}^{n}, \mathsf{ct} = \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}, \mathsf{L}_{\tau,j}\}_{j=i}^{n}\right\}_{\tau\in[T]}\right), \mathsf{sk} = (s, \mathrm{s\_secret}), (P,t), \mathsf{digest}\right).$$

1. **Generate the "new" garbled program for P:** Generate garbled circuits $\left\{\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}\}_{j=i+1}^{n}\right\}_{\tau=T+1}^{T+t}$.
   (a) Set $\mathsf{stateKeys}^{\tau}, \mathsf{dataKeys}^{\tau}, \mathsf{digestKeys}^{\tau}, \mathsf{RdKeys}^{\tau,j}, \mathsf{PKeys}^{\tau,j}$ for each $\tau \in [T+1, T+t]$ and $j \in [i+1, n]$ as $\mathsf{SampleKeys}(F_s(\mathtt{GC\_}\star||\tau))$ where $F_s(\mathtt{GC\_}\star||\tau)$ is the randomness used and $\star \in \{\mathtt{STATE}, \mathtt{DATA}, \mathtt{DIGEST}, \mathtt{RD}, \mathtt{P}\}$, respectively.
   Set $\mathsf{stateKeys}^{\tau}, \mathsf{dataKeys}^{\tau}, \mathsf{digestKeys}^{\tau}$ to $\mathsf{SampleKeys}(0^*)$ for $\tau = T+t+1$.
   (b) *Garble* $\mathsf{C}^{\mathsf{step}}$ *circuits:* For each $\tau \in [T+1, T+t]$

$$\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}} \leftarrow \mathsf{ReGCircuit}\left(\mathsf{C}^{\mathsf{step}}[i, \mathsf{crs}, P, \mathsf{Keys}^{\tau+1}, F_s(\mathtt{PSI}||\tau)], \left(\mathsf{Keys}^{\tau}, \{\mathsf{RdKeys}^{\tau,j}\}_{j=i+1}^{n}\right)\right),$$

   where $\mathsf{Keys}^{\tau} = (\mathsf{stateKeys}^{\tau}, \mathsf{dataKeys}^{\tau}, \mathsf{digestKeys}^{\tau})$.
   Embed labels $\mathsf{dataKeys}_0^{T+1}$ and $\mathsf{digestKeys}_{\mathsf{digest}}^{T+1}$ in $\widetilde{\mathsf{C}}_{T+1}^{\mathsf{step}}$.
   (c) *Garble* $\mathsf{C}^{\mathsf{PRF}}$ *circuits:* For each $[T+1, T+t]$ and $j \in [i+1, n]$, compute

$$\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}} \leftarrow \mathsf{ReGCircuit}'\left(\mathsf{C}^{\mathsf{PRF}}[\tau], \mathsf{PKeys}^{\tau,j}, \mathsf{RdKeys}^{\tau,j}\right).$$

2. **Generate the OT second messages for newly generated circuits**: For all $\tau \in [T+1, T+t]$ and $j \in [i+1, n]$ compute $\mathsf{L}_{\tau,j} \leftarrow \mathsf{OT}_2\left(\mathsf{pk}_j, \mathsf{PKeys}^{\tau,j}\right)$.
3. **Obtain partial labels for previous circuits**:
   (a) For every $\tau \in [T]$, compute $\mathsf{M}_{\tau,i} = \mathsf{OT}_3\left(\mathsf{L}_{\tau,i}, \mathrm{s\_secret}\right)$ and $\widetilde{\mathsf{C}}_{\tau,i}^{\mathsf{PRF}}$ using input labels $\mathsf{M}_{\tau,i}$ and embed the labels in $\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}}$.
   (b) If $i = 1$, then $\mathsf{L}_0 \leftarrow \mathsf{OT}_2(\mathsf{L}_0, \mathsf{stateKeys}^1)$.
4. **Re-randomize previous garbled circuits:** If $i > 1$, do the following:
   (a) For each $\tau \in [T]$, re-randomize the circuit $\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}}$ using $\mathsf{Re\text{-}rand}(\cdot)$ (see Definition 7.4) such that the input wire keys are randomized using $F_s(\mathtt{GC\_} \star ||\tau)$, where $\star \in \{\mathtt{STATE}, \mathtt{DATA}, \mathtt{DIGEST}, \mathtt{RD}\}$ for different input wires appropriately.
   (b) For each $\tau \in [T]$, re-randomize the circuits $\{\mathsf{C}_{\tau,j}^{\mathsf{PRF}}\}_{j\in[i+1,n]}$ and $\{\mathsf{L}_{\tau,j}\}_{j\in[i+1,n]}$ using $\mathsf{Re\text{-}rand}(\cdot)$ such that the input wires are randomized using $F_s(\mathtt{GC\_P}||\tau)$ and output wires are randomized using $F_s(\mathtt{GC\_RD}||\tau)$.
   (c) Re-randomize L_0 using $F_s(\mathtt{GC\_STATE}||1)$.
5. Output $\mathsf{ct}' = \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}_{\tau}^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}, \mathsf{L}_{\tau,j}\}_{j=i+1}^{n}\right\}_{\tau=1}^{T+t}\right)$.

Figure 16: Homomorphic evaluation algorithm

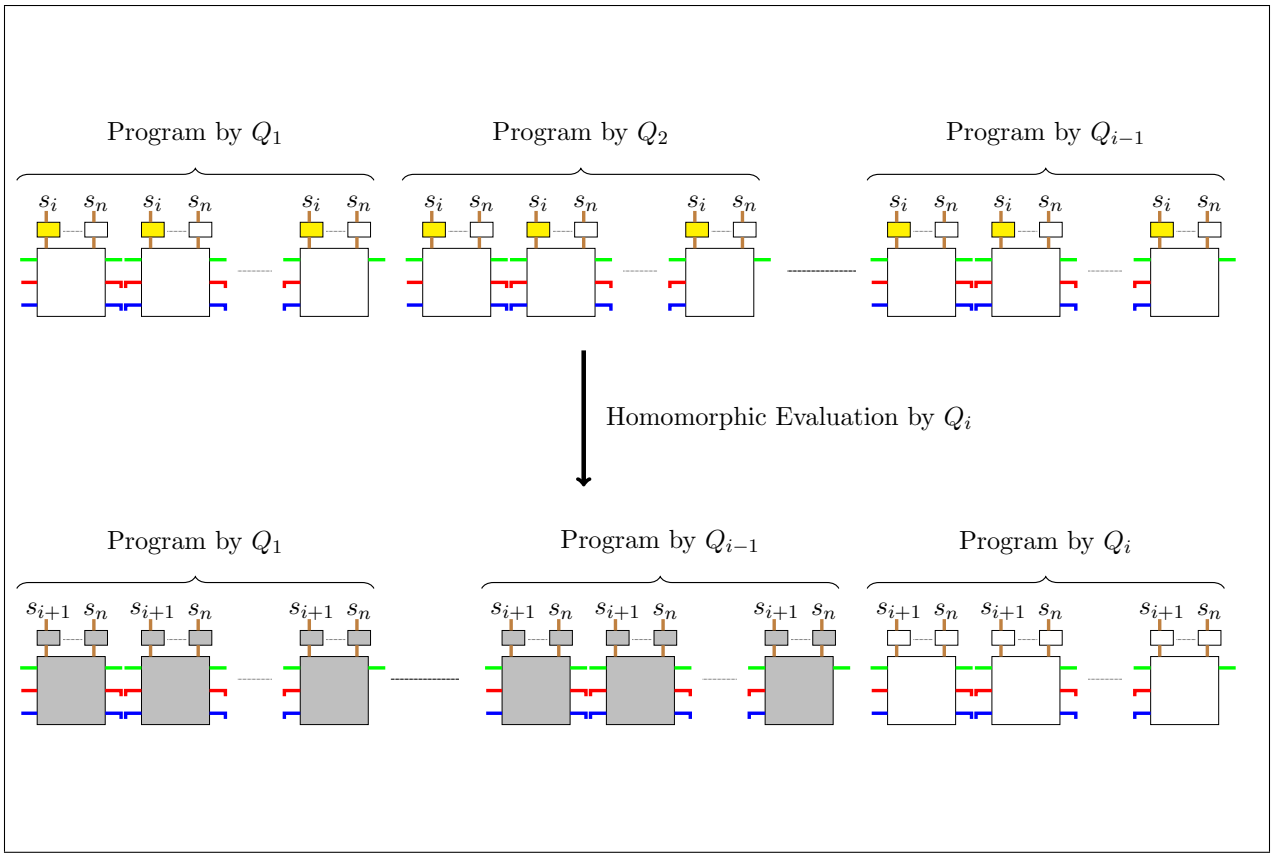


Figure 17: Homomorphic Evaluation by Q_i: Q_i contributes new circuits (denoted in white in the lower layer) and processes the input circuits as follows: (i) computes the yellow circuits, and (ii) re-randomizes all input circuits. The re-randomized circuits are shown in gray color.

circuit executes one CPU step and passes on the updated state, new bit read, and new digest to the next step circuit. Note that we do not achieve this by passing the output wires of τ into input wires of $\tau + 1$. That is, the output wire of τ will not have same keys as input wires of $\tau + 1$ (Note that the two consecutive step circuits are not connected by solid lines in Figure 17.). Hence, the step circuit τ will have the keys of the next circuit hard-coded inside it.

Next, we explain the logic of a step-circuit. First, it computes the new $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData})$. Next, it computes the transformed keys $\mathsf{nextKeys}^\dagger$ of the next step-circuit using the hard-coded keys and the input randomness (this uses the transform functionality of re-randomizable Yao from Section 7.2.2). Then, in the case of a "read" it outputs $\mathsf{stateKeys}^\dagger$ corresponding to new state', labels for data via laconic OT procedure $\mathsf{Send}(\cdot)$ for location L where the sender's inputs are $\mathsf{dataKeys}_0^\dagger, \mathsf{dataKeys}_1^\dagger$ and $\mathsf{digestKeys}^\dagger$ corresponding to digest. The case of a write is similar, but now the labels of new updated digest are transferred via laconic OT procedure $\mathsf{SendWrite}(\cdot)$. Note that it follows via correctness of reads and writes of the laconic OT that the evaluator would be able to recover the correct labels for the read-data and the new digest.

The down-bend in output and input wires of step-circuits for data and digest in Figure 17 represents that these keys are not output in the clear, but are output using laconic OT.

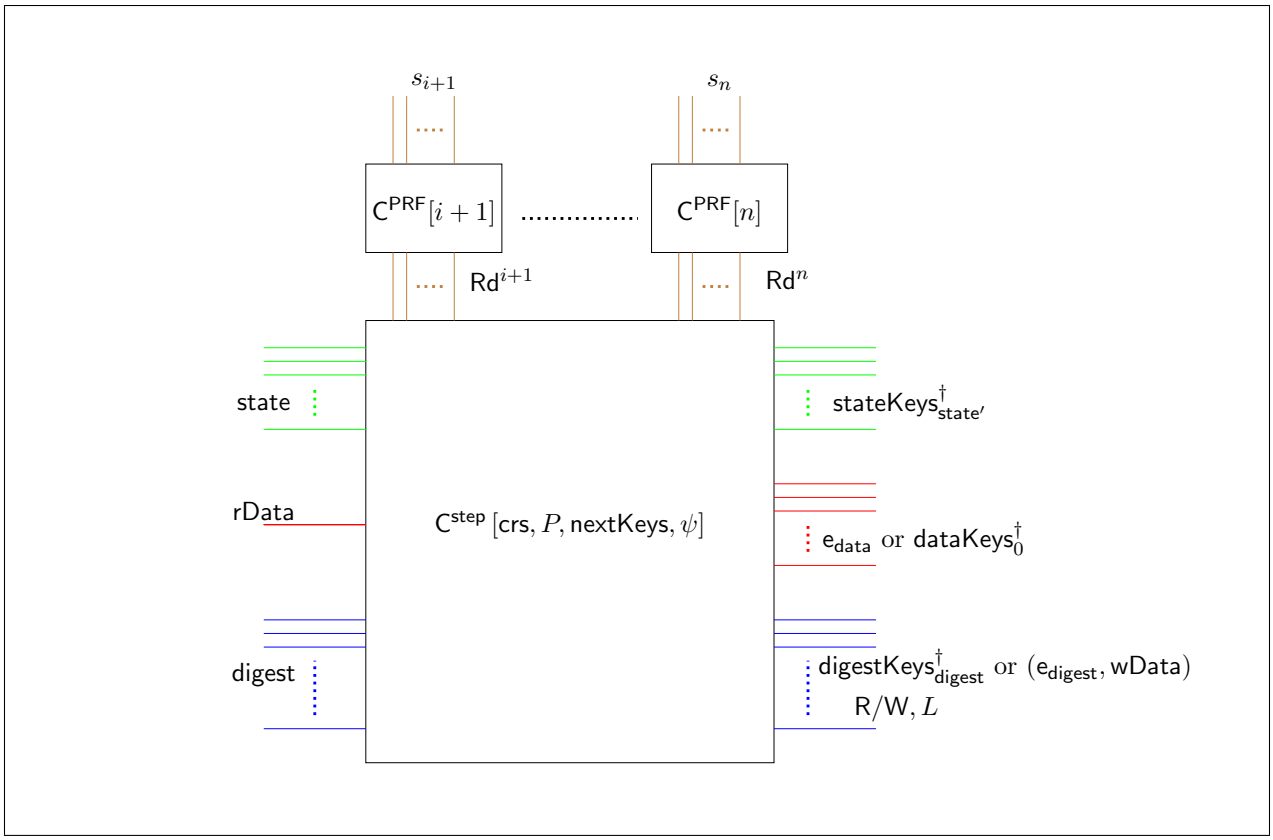


Figure 18: One step circuit along with the attached PRF circuits.

Correct labels will be learnt during execution using the encoded database $\widetilde{D}_i$ and laconic OT procedures.

PRF circuits $\mathsf{C}^{\mathsf{PRF}}_{\tau,j}$ (Figure 20) : This circuit takes as input a PRF key s_j of client Q_j and outputs the PRF value corresponding to time-step τ. The use of these circuits will be clear when we describe the re-randomization step below.

All these circuits are garbled such that the keys for output wires of PRF circuits are same as keys for Rd input keys of step circuits. In Figure 17, this is depicted by joining the output wires of PRF circuits with Rd input wires of step circuit with a solid line. The garbled program consists of garbled step circuits and garbled PRF circuits $\{\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}, \{\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,j}\}_{j\in\{i+1,\ldots,n\}}\}_{\tau\in[T+1,T+t]}$. The client also embeds labels for $\mathsf{rData} = 0$ and digest_i in the first step circuit.

2. Generate OT messages for $\mathsf{C}^{\mathsf{PRF}}_{\tau,j}$: Recall that this circuit takes as input a PRF key s_j of client Q_j whose OT first message is present in pk_j. Client Q_i generates the OT second message $\mathsf{L}_{\tau,j}$ for the input keys of $\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,j}$.

3. Evaluating the PRF circuits for itself: Note that the ciphertext ct consists of a sequence of step circuits and PRF circuits for each step circuit corresponding to $j \in \{i, \ldots, n\}$. See Figure 17 where the PRF circuits for client Q_i are depicted in yellow. Q_i computes the input labels for $\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,i}$ using the OT message $\mathsf{L}_{\tau,i}$ and embeds the output labels in to $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau}$ for

Hard-coded parameters: $[i, \mathsf{crs}, P, \mathsf{nextKeys} = (\mathsf{stateKeys}, \mathsf{dataKeys}, \mathsf{digestKeys}), \psi]$.
Input: $((\mathsf{state}, \mathsf{rData}, \mathsf{digest}), (\{\omega_j, \phi_j\}_{j>i}))$.

$(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) := C^P_{\mathsf{CPU}}(\mathsf{state}, \mathsf{rData})$.
$\mathsf{nextKeys}^\dagger := \mathsf{Transform}(\mathsf{nextKeys}, \{\omega_j\}_{j>i})$.
Parse $\mathsf{nextKeys}^\dagger$ as $(\mathsf{stateKeys}^\dagger, \mathsf{dataKeys}^\dagger, \mathsf{digestKeys}^\dagger)$.

if $\mathsf{R/W} = \mathsf{read}$ **then**
  $\mathsf{e}_{\mathsf{data}} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, \mathsf{dataKeys}^\dagger; \psi \oplus \bigoplus_{j>i} \phi_j)$.
  return $\Big((\mathsf{stateKeys}^\dagger_{\mathsf{state}'}, \quad \mathsf{e}_{\mathsf{data}}, \quad \mathsf{digestKeys}^\dagger_{\mathsf{digest}}), \mathsf{R/W}, L\Big)$.
else
  $\mathsf{e}_{\mathsf{digest}} \leftarrow \mathsf{SendWrite}\Big(\mathsf{crs}, \mathsf{digest}, L, \mathsf{wData}, \mathsf{digestKeys}^\dagger; \psi \oplus \bigoplus_{j>i} \phi_j\Big)$.
  return $\Big((\mathsf{stateKeys}^\dagger_{\mathsf{state}'}, \quad \mathsf{dataKeys}^\dagger_0, \quad \mathsf{e}_{\mathsf{digest}}, \mathsf{wData} \quad), \mathsf{R/W}, L\Big)$.

Figure 19: Pseudocode of the step circuit $\mathsf{C}^{\mathsf{step}}[i, \mathsf{crs}, P, \mathsf{nextKeys}, \psi]$.

Hard-coded parameters: $[\tau]$.
Input: s.
Output: $\Big(\{F_s(\texttt{GC\_}\star||\tau+1)\}_{\star\in\{\texttt{STATE},\texttt{DATA},\texttt{DIGEST}\}}, F_s(\texttt{LACONIC\_OT}||\tau)\Big)$.

Figure 20: PRF circuit $\mathsf{C}^{\mathsf{PRF}}[\tau]$.

all $\tau \in [T]$. In other words, Q_i consumes the first PRF circuits from each step of previous ciphertext ct.

4. Re-randomize the previous circuits: After consuming the first PRF circuit from each step, Q_i randomizes all the remaining circuits using appropriate randomness. Note that the input keys of $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ are randomized using the exact randomness that was fed into $\mathsf{C}^{\mathsf{step}}_\tau$ via the PRF circuit for Q_i. This makes sure that the hard-coded input keys of step $\tau+1$ are randomized consistently in the same way as how Q_i will randomize the circuit $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$.

   Hence, to conclude, the PRF circuits are present to provide the randomness needed to randomize the hard-coded keys inside the step circuits [13].

Homomorphic Decryption: $y = \mathsf{Dec}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}\left(\mathsf{crs}, \mathrm{x\_secret}, \mathsf{ct} = \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau\right\}_{\tau\in[T]}\right)\right)$.

The algorithm is described formally in Figure 21. It takes as input the secret state of the server x_secret and the final ciphertext ct_n consisting of OT message for x and sequence of T step-circuits, where $T = \sum_{i\in[n]} t_i$. Note that all the PRF circuits have been evaluated already by correct parties and correct labels for RdKeys have been embedded into the step-circuits. The server does the following for decryption:

1. It obtains the $\mathsf{stateLabels}$ for the first circuit by running OT_3. Note that the first circuit of program of any client has labels for data and digest already embedded. Hence, now the server knows all the labels for the first circuit.

[13] Note that randomization of garbled circuits preserves the functionality. Since the keys for the next circuit are transferred using the laconic OT, we need to feed in correct keys into the Send functions of laconic OT.

Homomorphic Decryption.

$y = \mathsf{Dec}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}\left(\mathsf{crs}, \mathrm{x\_secret}, \mathsf{ct} = \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}\right\}_{\tau\in[T]}\right)\right)$, where $T = \sum_{i\in[n]} t_i$.

1. For all $i \in [n]$, parse $\widetilde{D}_i = \left(\mathsf{digest}_i, \hat{\mathsf{D}}_i\right)$.
2. Compute $\mathsf{M}_0 = \mathsf{OT}_3\left(\mathsf{L}_0, \mathrm{x\_secret}\right)$.
3. Parse $\widetilde{\mathsf{C}}_1^{\mathsf{step}} = (\widetilde{\mathsf{C}}_1^{\mathsf{step}}, \mathsf{dataLabels}, \mathsf{digestLabels})$.
4. Define $\mathsf{Labels}^1 = (\mathsf{stateLabels}^1 = \mathsf{M}_0, \mathsf{dataLabels}^1 = \mathsf{dataLabels}, \mathsf{digestLabels}^1 = \mathsf{digestLabels})$.
5. For $\tau = 1$ to T do the following:

   Define i s.t. $\tau \in \left[\sum_{j\in[i-1]} t_j + 1, \sum_{j\in[i]} t_j\right]$.

   $(X, \mathsf{R/W}, L) := \mathsf{ReEval}\left(\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, \mathsf{Labels}^\tau\right)$.

   if $\mathsf{R/W} = \mathsf{read}$ **then**

   Parse $X = (\mathsf{stateLabels}^{\tau+1}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestLabels}^{\tau+1})$.

   $\mathsf{dataLabels}^{\tau+1} := \mathsf{Receive}^{\hat{\mathsf{D}}_i}(\mathsf{crs}, \mathsf{e}_{\mathsf{data}}, L)$.

   else

   Parse $X = (\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData})$.

   $\mathsf{digestLabels}^{\tau+1} := \mathsf{ReceiveWrite}^{\hat{\mathsf{D}}_i}(\mathsf{crs}, L, \mathsf{wData}, \mathsf{e}_{\mathsf{digest}})$.

   if $\tau = \sum_{j\in[i]} t_j$ and $\tau < T$ **then**

   Parse $\widetilde{\mathsf{C}}_{\tau+1}^{\mathsf{step}} = (\widetilde{\mathsf{C}}_{\tau+1}^{\mathsf{step}}, \mathsf{dataLabels}, \mathsf{digestLabels})$

   Set $\mathsf{dataLabels}^{\tau+1} = \mathsf{dataLabels}$ and $\mathsf{digestLabels}^{\tau+1} = \mathsf{digestLabels}$.

   $\mathsf{Labels}^{\tau+1} := (\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{digestLabels}^{\tau+1})$.
6. Decode output y using $\left(\mathsf{stateLabels}^{T+1}, \mathsf{SampleKeys}(0^*)\right)$.

Figure 21: Decryption algorithm for multi-hop scheme for RAM.

2. For $\tau \in [T]$, the server executes the circuit $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$, and learns the labels for the next circuit via running the receiver algorithms of laconic OT correctly.

7.3.2 Correctness

Here we first prove correctness (as defined in Section 7.1) for a single execution. In fact, we would prove something stronger that would help us extend the scheme to multiple executions in a straightforward manner. We prove the following two properties:

Property 1. For the above scheme, $y = P_n^{D_n}\left(\ldots P_1^{D_1}(x)\ldots\right)$, where programs, databases and input x are as defined above.

Property 2. Let $\hat{\mathsf{D}}_i', \mathsf{digest}_i'$ denote the updated encoded database and digest with the server after

the execution. We show that these are equal to $\mathsf{Hash}(\mathsf{crs}, D'_i)$, where D' results after executing $P_i^{D_i}(P_{i-1}^{D_{i-1}}(\ldots P_1^{D_1}(x)\ldots))$.

Below we prove correctness via a sequence of facts and claims.

Fact 7.5. *At any point in homomorphic evaluation, the circuit $\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,j}$ and the second OT message for its input keys $\mathsf{L}_{\tau,j}$ are consistent.*

This follows from correctness of $\mathsf{OT}_2(\cdot,\cdot)$ procedure when it is generated and the fact that re-randomization happens consistently in Re-rand procedure of re-randomizable garbled circuits.

Fact 7.6. *During the homomorphic evaluation of client Q_i, in Step 3a, Figure 16 while obtaining partial labels, $\mathsf{M}_{\tau,i} = \mathsf{PKeys}^{\tau,i}_s$, where s is the PRF key of Q_i.*

This follows from the correctness of OT protocol as well as Fact 7.5.

Fact 7.7. *During the homomorphic evaluation of client Q_i, in Step 3a, Figure 16 the labels embedded in circuit $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$ correspond to $\mathsf{RdKeys}^{\tau,i}_{\omega_i,\phi_i}$ where $\phi_i = F_s(\mathtt{LACONIC\_OT}||\tau)$ and $\omega_i = \{F_s(\mathtt{GC\_}\star||\tau+1)\}_{\star\in\{\mathtt{STATE},\mathtt{DATA},\mathtt{DIGEST}\}}$.*

This is because the functionality of the $\mathsf{C}^{\mathsf{PRF}}$ is preserved in randomization so far, Fact 7.6 and because the output keys of $\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,i}$ and $\mathsf{RdKeys}^{\tau,i}$ are same when they are generated and are re-randomized using same randomness in Step 4b of Figure 16.

Recall that ct_n consists of garbled step-circuits of client Q_1 followed by Q_2 and so on. We prove the following fact about garbled step circuits belonging to some client Q_i in final ciphertext ct_n.

Claim 7.8. *Consider circuits $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$ and $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ such that both belong to program P_i for some i. Since all the PRF circuits $\widetilde{\mathsf{C}}^{\mathsf{PRF}}$ have been evaluated, the value $\mathsf{nextKeys}^\dagger$ in $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$ is well defined. Then, $\mathsf{nextKeys}^\dagger = \mathsf{Keys}^{\tau+1}$ where $\mathsf{Keys}^{\tau+1}$ corresponds to the input keys for $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ in ct_n.*

Proof. Initially, Q_i picks $\mathsf{Keys}^{\tau+1}$ as $\mathsf{SampleKeys}(F_s(\mathtt{GC\_}\star||\tau+1))$, where $\star \in \{\mathtt{STATE},\mathtt{DATA},\mathtt{DIGEST}\}$ and uses them in garbling of $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ as well as are hardcoding inside $\mathsf{C}^{\mathsf{step}}_\tau$.

Then, $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ is randomized by clients $Q_{i+1},\ldots,Q_n$ such that the $\mathsf{stateKeys}, \mathsf{dataKeys}, \mathsf{digestKeys}$ are randomized sequentially using $\omega_j = (F_{s_j}(\mathtt{GC\_STATE}||\tau+1), F_{s_j}(\mathtt{GC\_DATA}||\tau+1), F_{s_j}(\mathtt{GC\_DIGEST}||\tau+1))$. This is same as $\mathsf{Transform}(\mathsf{Keys}^{\tau+1}, \{\omega_j\}_{j>i})$ inside $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$. By Fact 7.7, ω_j is the value used for $\mathsf{Transform}$ in $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$. □

Claim 7.9. *The above claim also holds for $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$ and $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$ when τ is the last circuit of a program for Q_i and $\tau+1$ is the first circuit for Q_{i+1}.*

Proof. When the client Q_i generates $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$, the keys hard-coded are $\mathsf{SampleKeys}(0^\star)$. Then, this circuit is re-randomized by Q_{i+1} resulting in keys $\mathsf{SampleKeys}(F_{s_{i+1}}(\mathtt{GC\_}\star||\tau+1))$ which same as the value used by Q_{i+1} to generate the step-circuit $\widetilde{\mathsf{C}}^{\mathsf{step}}_{\tau+1}$. □

Fact 7.10. *The first garbled step circuit $\widetilde{\mathsf{C}}^{\mathsf{step}}_1$ gets evaluated on $(x, 0, \mathsf{digest}_1)$.*

This follows from correctness of OT and consistency of re-rerandomization of OT and garbled circuits similar to Fact 7.5.

Now, we will prove a lemma about the execution of circuits generated by client Q_1. Then, we will prove a claim about the inputs on which circuit of Q_2 is executed. Finally, the correctness of execution programs of all clients would follow in a similar manner.

Lemma 7.11. *Consider the program P_1 and the database D_1 of the first client and the input x of the server. Consider the execution $P_1^{D_1}(x)$ execution in the clear as $(\mathsf{state}_\tau, \mathsf{rData}_\tau)$ as the values on which $\mathsf{C}^{\mathsf{step}}$ is executed. Also, let $(\hat{\mathsf{D}}_1^\tau, \mathsf{digest}_\tau)$ denote the $\mathsf{Hash}(\mathsf{crs}, D_1^\tau)$, where D_1^τ is the database at beginning of step τ. Then, while decryption, $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$ is executed on inputs $(\mathsf{state}_\tau, \mathsf{rData}_\tau, \mathsf{digest}_\tau)$. Moreover, the encoded database held by the server before step τ is $\hat{\mathsf{D}}_1^\tau$.*

Proof. We will prove this lemma by induction on τ. The base case follows from Fact 7.10. Assume that the lemma holds for $\tau = \rho$, then we prove that the lemma holds for $\rho + 1$ as follows: So it holds that $\widetilde{\mathsf{C}}_\rho^{\mathsf{step}}$ is executed on $(\mathsf{state}_\rho, \mathsf{rData}_\rho, \mathsf{digest}_\rho)$. Moreover, $(\hat{\mathsf{D}}_1^\rho, \mathsf{digest}_\rho)$ denote the $\mathsf{Hash}(\mathsf{crs}, D_1^\rho)$. Note that $\widetilde{\mathsf{C}}_\rho^{\mathsf{step}}$ correctly implements its code that includes one CPU step of P_1. Hence, $(\mathsf{state}', \mathsf{R/W}, L, \mathsf{wData}) = C_{\mathsf{CPU}}^P(\mathsf{state}_\rho, \mathsf{rData}_\rho)$. Also, by Claim 7.8, $\mathsf{nextKeys}^\dagger$ in $\widetilde{\mathsf{C}}_\rho^{\mathsf{step}}$ are correct input keys for $\widetilde{\mathsf{C}}_{\rho+1}^{\mathsf{step}}$. There are following two cases:

- $\mathsf{R/W} = \mathsf{read}$: In this case, database and the digest are unchanged. New CPU state and digest are output correctly. Moreover, the labels for bit read from the memory will be learnt via $\mathsf{Receive}$ of updatable laconic OT. Correctness of these labels follows from correctness of read of laconic OT.
- $\mathsf{R/W} = \mathsf{write}$: Similar to above, in this case new state and data keys are correctly output. Moreover, the digest keys w.r.t. the new updated digest are output via laconic OT. The correctness of these labels follows from correctness of laconic OT write function $\mathsf{ReceiveWrite}$. Finally, in this function, the encoded database is updated correctly.

□

Lemma 7.12. *Let $\widetilde{\mathsf{C}}_{t_1}^{\mathsf{step}}$ be the last circuit of client Q_1 or program P_1. Then, during decryption, $\widetilde{\mathsf{C}}_{t_1+1}^{\mathsf{step}}$ is executed on $(y_1, 0, \mathsf{digest}_2)$, where $y_1 = P_1^{D_1}(x)$ and digest_2 is the digest for D_2.*

Proof. At the time of homomorphic evaluation, in Step 1b, Figure 16, labels $\mathsf{dataKeys}_0^{t_1+1}$ and $\mathsf{digestKeys}_{\mathsf{digest}}^{t_1+1}$ are embedded in $\widetilde{\mathsf{C}}_{t_1+1}^{\mathsf{step}}$. Also, by Claim 7.9, in the final ciphertext ct, $\mathsf{nextKeys}^\dagger$ inside $\widetilde{\mathsf{C}}_{t_1}^{\mathsf{step}}$ are correct keys for $\widetilde{\mathsf{C}}_{t_1+1}^{\mathsf{step}}$. Hence, the lemma holds since $\widetilde{\mathsf{C}}_{t_1}^{\mathsf{step}}$ outputs $\mathsf{stateKeys}_{\mathsf{state}'}^\dagger$, where $\mathsf{state}' = y_1$.

□

Lemma 7.13. *Consider the program P_i and the database D_i of the client Q_i and the input x of the server. Consider the execution $P_i^{D_i}(y_{i-1})$ execution in the clear as $(\mathsf{state}_\tau, \mathsf{rData}_\tau)$ as the values on which $\mathsf{C}^{\mathsf{step}}$ is executed. Also, let $(\hat{\mathsf{D}}_i^\tau, \mathsf{digest}_\tau)$ denote the $\mathsf{Hash}(\mathsf{crs}, D_i^\tau)$, where D_i^τ is the database at beginning of step τ. Then, while decryption, $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$ is executed on inputs $(\mathsf{state}_\tau, \mathsf{rData}_\tau, \mathsf{digest}_\tau)$. Moreover, the encoded database held by the server before step τ is $\hat{\mathsf{D}}_i^\tau$.*

Proof. The lemma follows via induction on number of clients where the base case is proved in Lemma 7.11. The rest of the proof follows simply via induction similar to Lemma 7.11 where at the end of each program and beginning of a new program we prove Lemma 7.12. This proves both the properties 1 and 2. □

7.3.3 Extending to Multiple Executions

Recall that for correctness we also proved that the after one execution, the resulting garbled database $\widetilde{D} = (\hat{\mathsf{D}}, \mathsf{digest})$ corresponds to the output of $\mathsf{Hash}(\mathsf{crs}, D')$, where D' is the correct database resulting after the execution in the clear (See Property 2, Section 7.3.2).

Given this invariant after the first execution, the next execution happens identically as the first execution with minor differences. To run the algorithm Eval, the clients need the updated digest of their respective databases. The updated digests of all the clients taking part in an execution would be sent by the server to the first client on that execution path, and would be passed along with each ciphertext. Also, to ensure that no PRF output is used twice, each PRF invocation would take the session identifier sid as an additional input. With these changes, the second execution is identical to the first execution and hence, its correctness follows in a straight-forward manner.

Also, this does not affect the UMA-security because the simulator of the ideal world is given the databases as well as memory access pattern of the honest clients as input as well.

Moreover, note that this generalizes to the scenario when different subset of clients take part in different executions. Only the digests of the relevant client are passed around by the server.[14]

7.3.4 Security Proof

Server privacy follows receiver privacy of oblivious transfer. For ease of exposition, we prove client UMA privacy for the setting of a single honest client Q_i for a single execution. At the end of this section we will show that the proof can be extended for the case of multiple honest clients and multiple executions as well.

In the following, we prove that there exists a PPT simulator $\mathsf{ihopSim}$ such that, for any set of databases $\{D_j\}_{j\in[n]}$, any sequence of compatible programs $P_1, \cdots, P_n$ running time $t_1, \cdots, t_n$ and input x, the outputs of the following two experiments are computational indistinguishable:

Real experiment

- $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow \mathsf{KeyGen}(1^\lambda)$ for $\forall j \in [n]$.
- $\widetilde{D}_j = (\mathsf{digest}_j, \hat{\mathsf{D}}_j) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_j)$ for $\forall j \in [n]$.
- $(\mathsf{ct}_0, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(\mathsf{crs}, x)$.
- $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(\mathsf{crs}, j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for $\forall j \in [n]$.
- Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

Simulated experiment

- $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{ihopSim}(1^\lambda, i)$.
- $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow \mathsf{KeyGen}(1^\lambda; r_j)$ for $\forall j \in [n] \setminus \{i\}$. Here, r_j are uniform random coins.
- $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right) \leftarrow \mathsf{ihopSim}(\mathsf{crs}, x, \{\mathsf{pk}_j, \mathsf{sk}_j, D_j, t_j\}_{j\in[n]}, \{P_j, r_j\}_{j\in[n]\setminus\{i\}}, \mathsf{MemAccess}_i, y_i)$, where $y_i = {P_i}^{D_i}\left(\cdots\left({P_1}^{D_1}(x)\right)\cdots\right)$.
- Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

[14] It can also extended to the setting, when a client Q_i occurs multiple times in the chain of clients in an execution. To handle this setting, the digest of D_i is passed along all the programs between two instances of this client as additional state.

The above definition can be made semi-malicious by allowing the adversary to pick random coins r_j adversarially given the public key pk_i of honest client as follows: $\{r_j\}_{j\in[n]\setminus\{i\}} \leftarrow \mathcal{A}(1^\lambda, \mathsf{crs}, \mathsf{pk}_i)$ that will be used to define $(\mathsf{pk}_j, \mathsf{sk}_j)$ in Step 2. Our proof would also support this stronger setting as well.

Construction of $\mathsf{ihopSim}$: We describe the two phases of $\mathsf{ihopSim}$. In the first phase, $\mathsf{ihopSim}$ generates the keys of honest client Q_i as $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(1^\lambda)$.

In the second phase, $\mathsf{ihopSim}$ is described in Figure 22. At a high level, $\mathsf{ihopSim}$ generates everything honestly except ct_i. When generating ct_i, it simulates the step circuits one by one from the last to the first using the output y_i and memory access $\mathsf{MemAccess}_i$. In particular, since $\mathsf{ihopSim}$ takes D_i and $\mathsf{MemAccess}_i$ as input, it can compute D_i and digest_i before every step circuit, and use that to compute the output of every step circuit. Security follows from security of re-randomization of SFE, namely re-randomized garbled circuits are indistinguishable from freshly generated ones and that freshly generated garbled circuits are indistinguishable from simulated ones.

Now we give a series of hybrids such that the first hybrid outputs $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right)$ in the real execution, and the last hybrid is the output of $\mathsf{ihopSim}$. Notice that the only difference between the real and ideal experiments is ct_i, so all the hybrids generate everything in the same way except ct_i.

- $\hat{\mathsf{H}}_0$: Output in the real experiment.
- H_0: In this hybrid, replace $F_{s_i}(\cdot)$ with a truly random function F. In particular, when computing ct_i as in Figure 16, in steps 1a and 4, use the values generated by F; in step 3a embed labels corresponding to the values from F. The indistinguishability of this hybrid with $\hat{\mathsf{H}}_0$ follows from the pseudo-randomness of $F_{s_i}(\cdot)$ and privacy of oblivious transfer (s_i is hidden in pk_i).
- H_m ($m \in [T_i]$): Next we consider a sequence of hybrids $\mathsf{H}_1, \cdots, \mathsf{H}_{T_i}$. The description of H_m is in Figure 23. Notice that ct_i consists of T_i step circuits with corresponding PRF circuits. In hybrid H_m, the step circuits from 1 to m are simulated while the remaining step circuits ($m+1$ to T_i) are generated honestly. Given all the programs and secret keys, the intermediate outputs as well as input/output labels of every step circuit can all be computed. Given the correct output of circuit $\mathsf{C}^{\mathsf{step}}_m$, the step circuits from 1 to m can be simulated one by one from the m-th to the first similarly as in $\mathsf{ihopSim}$.

  To show H_m is indistinguishable from H_{m-1}, first notice that they are the same except $\left(\widetilde{\mathsf{C}}^{\mathsf{step}}_m, \{\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{m,j}, \mathsf{L}_{m,j}\}^n_{j=i+1}\right)$ in ct_i. Consider an intermediate hybrid $\hat{\mathsf{H}}_m$ which is the same as H_m except that in step 2f when $\tau = m$, follow the steps in Figure 24. In particular, when $\tau = m$, $\hat{\mathsf{H}}_m$ computes the output of $\mathsf{C}^{\mathsf{step}}_m$ and uses that output to simulate $\widetilde{\mathsf{C}}^{\mathsf{step}}_m$ by $\mathsf{CircSim}$ and OT_2. The output of $\widetilde{\mathsf{C}}^{\mathsf{step}}_m$ is the same for H_{m-1} and $\hat{\mathsf{H}}_m$. The indistinguishibility of $\hat{\mathsf{H}}_m$ and H_{m-1} follows from the security of garbled circuits directly when $m \in [T_{i-1}+1, T_i]$. When $m \in [T_{i-1}]$, it follows from the security of garbled circuits and re-randomization. More precisely, the re-randomized garbled circuit is indistinguishable from a freshly generated garbled circuit, which is indistinguishable from a simulated one. Notice that the random coins used in re-randomization for $\widetilde{\mathsf{C}}^{\mathsf{step}}_m$ is $F(\mathtt{GC}\_\star||m)$, which is not used anywhere else in H_{m-1}, so it can be treated as truly random coins.

$\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right) \leftarrow \mathsf{ihopSim}\left(\mathsf{crs}, x, \{\mathsf{pk}_j, \mathsf{sk}_j, D_j, t_j\}_{j\in[n]}, \{P_j\}_{j\in[n]\setminus\{i\}}, \mathsf{MemAccess}_i, y_i\right)$.

1. Compute $\widetilde{D}_j = (\mathsf{digest}_j, \hat{\mathsf{D}}_j) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_j)$ for $\forall j \in [n]$.
   Compute $(\mathsf{ct}_0, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$.
   Compute $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for every $j \in [i-1]$.
   Pick a random function F (in the following use random values for $F(\cdot)$).
2. Let $T_j := \sum_{k\in[j]} t_k$. Generate ct_i as follows:
   (a) Run the program ${P_{i-1}}^{D_{i-1}}\left(\cdots\left({P_1}^{D_1}(x)\right)\cdots\right)$ to obtain $(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$ for every CPU step $\tau \in [T_{i-1}]$. Obtain $(\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$ for $\tau \in [T_{i-1}+1, T_i]$ from $\mathsf{MemAccess}_i$.
   (b) $\left(\mathsf{stateKeys}^{T_i+1}, \mathsf{dataKeys}^{T_i+1}, \mathsf{digestKeys}^{T_i+1}\right) \leftarrow \mathsf{SampleKeys}(0^*)$.
   Compute $\mathsf{stateLabels}^{T_i+1}$ using y_i; compute $\mathsf{dataLabels}^{T_i+1}, \mathsf{digestLabels}^{T_i+1}$ using $(D_i, \mathsf{digest}_i, \mathsf{R/W}, L, \mathsf{wData})$ of the last CPU step.
   (c) For $\tau = T_i$ downto 1, do the following:
   $(\mathsf{R/W}, L, \mathsf{wData}) := (\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$.
   Define j s.t. $\tau \in [T_{j-1}+1, T_j]$.
   Let D be the database of Q_j before step τ.

   $\left(\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{digestLabels}^{\tau+1}\right)$
   $\leftarrow \mathsf{Transform}\left((\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{digestLabels}^{\tau+1}),\right.$
   $\left.\left\{F_{s_{i+1}}(\mathtt{GC\_}\star\,||\tau+1)\right\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}} || \cdots || \left\{F_{s_n}(\mathtt{GC\_}\star\,||\tau+1)\right\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}}\right)$.

   if $\mathsf{R/W} = \mathsf{read}$ **then**
   $\mathsf{e}_{\mathsf{data}} \leftarrow \ell\mathsf{OTSim}\left(\mathsf{crs}, D, L, \mathsf{dataLabels}^{\tau+1}\right)$.
   $X \leftarrow (\mathsf{stateLabels}^{\tau+1}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestLabels}^{\tau+1})$.
   else
   $\mathsf{e}_{\mathsf{digest}} \leftarrow \ell\mathsf{OTSimWrite}\left(\mathsf{crs}, D, L, \mathsf{wData}, \mathsf{digestLabels}^{\tau+1}\right)$.
   $X \leftarrow (\mathsf{stateLabels}^{\tau+1}, \mathsf{dataLabels}^{\tau+1}, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData})$

   $\left(\left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}\}_{j=i+1}^n\right\}, \mathsf{Labels}^\tau\right) \leftarrow \mathsf{CircSim}\left(1^\lambda, \mathcal{U}, (X, \mathsf{R/W}, L)\right)$ such that the output labels of $\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}$ are the same as input labels of $\mathsf{RdLabs}^{\tau,j}$ for $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$.
   Parse $\mathsf{Labels}^\tau = \left(\mathsf{stateLabels}^\tau, \mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau, \{\mathsf{PLabels}^{\tau,j}\}_{j\in[i+1,n]}\right)$.
   $\mathsf{L}_{\tau,j} \leftarrow \mathsf{OT}_2\left(\mathsf{pk}_j, (\mathsf{PLabels}^{\tau,j}, \mathsf{PLabels}^{\tau,j})\right)$ for every $j \in [i+1, n]$.

   if $\tau = T_{j-1}+1$ **then**
   Embed $\mathsf{stateLabels}^\tau$ and $\mathsf{digestLabels}^\tau$ in $\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}$.
   $(\mathsf{stateKeys}^\tau, \mathsf{dataKeys}^\tau, \mathsf{digestKeys}^\tau) \leftarrow \mathsf{Transform}\Big(\mathsf{SampleKeys}(0^*),$
   $\left\{F_{s_j}(\mathtt{GC\_}\star\,||\tau)\right\} || \cdots || \left\{F_{s_{i-1}}(\mathtt{GC\_}\star\,||\tau)\right\} || \left\{F(\mathtt{GC\_}\star\,||\tau)\right\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}}\Big)$.
   Compute $(\mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau)$ using $(D_{j-1}, \mathsf{digest}_{j-1}, \mathsf{R/W}, L, \mathsf{wData})$ at step $\tau - 1$.
   (d) $\mathsf{L}_0 \leftarrow \mathsf{OT}_2\left(\mathsf{ct}_0, (\mathsf{stateLabels}^1, \mathsf{stateLabels}^1)\right)$.
   (e) $\mathsf{ct}_i := \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}, \mathsf{L}_{\tau,j}\}_{j=i+1}^n\right\}_{\tau\in[T_i]}\right)$.
3. Compute $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for every $j \in [i+1, n]$.
4. Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

Figure 22: Simulator of multi-hop RAM

1. Compute $\widetilde{D}_j = (\mathsf{digest}_j, \hat{\mathsf{D}}_j) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_j)$ for $\forall j \in [n]$.
   Compute $(\mathsf{ct}_0, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$.
   Compute $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for every $j \in [i-1]$.
   Pick a random function F (in the following use random values for $F(\cdot)$).
2. Let $T_j := \sum_{k\in[j]} t_k$. Generate ct_i as follows:
   (a) Compute $\left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}, \mathsf{L}_{\tau,j}\}_{j=i+1}^n\right\}_{\tau\in[m+1,T_i]}$ honestly as in Figure 16.
   (b) Run the program $P_i{}^{D_i}\left(\cdots\left(P_1{}^{D_1}(x)\right)\cdots\right)$ to obtain $(\mathsf{state}^\tau, \mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$ for every $\tau \in [T_i]$.
   (c) Define j s.t. $m \in [T_{j-1}+1, T_j]$.
   (d) Set $\mathsf{stateKeys}^{m+1}, \mathsf{dataKeys}^{m+1}, \mathsf{digestKeys}^{m+1}$ as $\mathsf{SampleKeys}(F_{s_j}(\mathtt{GC\_} \star ||m+1))$ where $\star \in \{\mathtt{STATE}, \mathtt{DATA}, \mathtt{DIGEST}\}$, respectively.
   If $m = T_j$, then set $\mathsf{stateKeys}^{m+1}, \mathsf{dataKeys}^{m+1}, \mathsf{digestKeys}^{m+1}$ to $\mathsf{SampleKeys}(0^*)$.
   If $j < i$, then $\left(\mathsf{stateKeys}^{m+1}, \mathsf{dataKeys}^{m+1}, \mathsf{digestKeys}^{m+1}\right)$
   $$\leftarrow \mathsf{Transform}\Big(\left(\mathsf{stateKeys}^{m+1}, \mathsf{dataKeys}^{m+1}, \mathsf{digestKeys}^{m+1}\right),$$
   $$\left\{F_{s_{j+1}}(\mathtt{GC\_}\star||m+1)\right\}||\cdots||\left\{F_{s_{i-1}}(\mathtt{GC\_}\star||m+1)\right\}||\left\{F(\mathtt{GC\_}\star||m+1)\right\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}}\Big).$$
   (e) Compute $(\mathsf{stateLabels}^{m+1}, \mathsf{dataLabels}^{m+1}, \mathsf{digestLabels}^{m+1})$ using $(\mathsf{state}^m, \mathsf{R/W}^m, L^m, \mathsf{wData}^m)$ and (D_j, digest_j) at step m.
   (f) For $\tau = m$ downto 1, do the following:
   Follow the same steps as in Figure 22 step 2c.
   (g) $\mathsf{L}_0 \leftarrow \mathsf{OT}_2\left(\mathsf{ct}_0, (\mathsf{stateLabels}^1, \mathsf{stateLabels}^1)\right)$.
   (h) $\mathsf{ct}_i := \left(\mathsf{L}_0, \left\{\widetilde{\mathsf{C}}_\tau^{\mathsf{step}}, \{\widetilde{\mathsf{C}}_{\tau,j}^{\mathsf{PRF}}, \mathsf{L}_{\tau,j}\}_{j=i+1}^n\right\}_{\tau\in[T_i]}\right)$.
3. Compute $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for every $j \in [i+1, n]$.
4. Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

Figure 23: Hybrid H_m

$(\mathsf{R/W}, L, \mathsf{wData}) := (\mathsf{R/W}^\tau, L^\tau, \mathsf{wData}^\tau)$.
Define j s.t. $\tau \in [T_{j-1}+1, T_j]$.
Let (D, digest) be the database and digest of Q_j before step τ.
Let state' be the CPU state after step τ.

$\big(\mathsf{stateKeys}^{\tau+1}, \mathsf{dataKeys}^{\tau+1}, \mathsf{digestKeys}^{\tau+1}\big)$
$\leftarrow \mathsf{Transform}\big((\mathsf{stateKeys}^{\tau+1}, \mathsf{dataKeys}^{\tau+1}, \mathsf{digestKeys}^{\tau+1}),$
$\big\{F_{s_{i+1}}(\mathtt{GC\_}\star\,||\tau+1)\big\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}} || \cdots || \big\{F_{s_n}(\mathtt{GC\_}\star\,||\tau+1)\big\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}}\Big)$.

if $\mathsf{R/W} = \mathsf{read}$ **then**
$\mathsf{e}_{\mathsf{data}} \leftarrow \mathsf{Send}(\mathsf{crs}, \mathsf{digest}, L, \mathsf{dataKeys}^{\tau+1}; F(\mathtt{PSI}||\tau) \oplus \bigoplus_{j>i} F_{s_j}(\mathtt{LACONIC\_OT}||\tau))$.
$X \leftarrow (\mathsf{stateKeys}^{\tau+1}_{\mathsf{state}'}, \mathsf{e}_{\mathsf{data}}, \mathsf{digestKeys}^{\tau+1}_{\mathsf{digest}})$.
else
$\mathsf{e}_{\mathsf{digest}} \leftarrow \mathsf{SendWrite}\Big(\mathsf{crs}, \mathsf{digest}, L, \mathsf{wData}, \mathsf{digestKeys}^{\tau+1}; F(\mathtt{PSI}||\tau) \oplus \bigoplus_{j>i} F_{s_j}(\mathtt{LACONIC\_OT}||\tau)\Big)$.
$X \leftarrow (\mathsf{stateKeys}^{\tau+1}_{\mathsf{state}'}, \mathsf{dataKeys}^{\tau+1}_0, \mathsf{e}_{\mathsf{digest}}, \mathsf{wData})$.

$\Big(\Big\{\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau, \{\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,j}\}_{j=i+1}^n\Big\}, \mathsf{Labels}^\tau\Big) \leftarrow \mathsf{CircSim}\left(1^\lambda, \mathcal{U}, (X, \mathsf{R/W}, L)\right)$ such that the output labels of $\widetilde{\mathsf{C}}^{\mathsf{PRF}}_{\tau,j}$ are the same as input labels of $\mathsf{RdLabs}^{\tau,j}$ for $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$.
Parse $\mathsf{Labels}^\tau = \big(\mathsf{stateLabels}^\tau, \mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau, \{\mathsf{PLabels}^{\tau,j}\}_{j\in[i+1,n]}\big)$.
$\mathsf{L}_{\tau,j} \leftarrow \mathsf{OT}_2\left(\mathsf{pk}_j, (\mathsf{PLabels}^{\tau,j}, \mathsf{PLabels}^{\tau,j})\right)$ for every $j \in [i+1, n]$.

if $\tau = T_{j-1}+1$ **then**
Embed $\mathsf{stateLabels}^\tau$ and $\mathsf{digestLabels}^\tau$ in $\widetilde{\mathsf{C}}^{\mathsf{step}}_\tau$.
$(\mathsf{stateKeys}^\tau, \mathsf{dataKeys}^\tau, \mathsf{digestKeys}^\tau) \leftarrow \mathsf{Transform}\Big(\mathsf{SampleKeys}(0^*),$
$\big\{F_{s_j}(\mathtt{GC\_}\star\,||\tau)\big\} || \cdots || \big\{F_{s_{i-1}}(\mathtt{GC\_}\star\,||\tau)\big\} || \big\{F(\mathtt{GC\_}\star\,||\tau)\big\}_{\star\in\{\mathtt{STATE,DATA,DIGEST}\}}\Big)$.
Compute $(\mathsf{dataLabels}^\tau, \mathsf{digestLabels}^\tau)$ using $(D_{j-1}, \mathsf{digest}_{j-1}, \mathsf{R/W}, L, \mathsf{wData})$ at step $\tau - 1$.

Figure 24: Difference of H_m and $\hat{\mathsf{H}}_m$.

To switch from $\hat{\mathsf{H}}_m$ to H_m, we replace X in Figure 24 with simulated $\mathsf{e}_{\mathsf{data}}$ and $\mathsf{e}_{\mathsf{digest}}$ for $\mathsf{CircSim}$ and OT_2. The indistinguishability follows from sender privacy of updatable laconic OT and that Send and $\mathsf{SendWrite}$ both take random coins $F(\mathtt{PSI}||m)$.

- $\hat{\mathsf{H}}_{T_i}$: Output in the simulated experiment. This hybrid is the same as H_{T_i}.

Extension. The above proof can be naturally extended to provide security for multiple clients and many executions. For example in the case of two clients Q_{i_1} and Q_{i_2}, $\mathsf{ihopSim}$ first computes $\left(\mathsf{ct}_0, \{\widetilde{D}_j\}_{j\in[n]}, \{\mathsf{ct}_j\}_{j\in[i_1-1]}\right)$ honestly, then computes ct_{i_1} same as in Figure 22 step 2. It then computes $\{\mathsf{ct}_j\}_{j\in[i_1+1,i_2-1]}$ from ct_{i_1} by Eval, and computes ct_{i_2} same as in Figure 22 step 2.[15] Finally it computes $\{\mathsf{ct}_j\}_{j\in[i_2+1,n]}$ from ct_{i_2} by Eval. To show this is indistinguishable from the real execution, we consider the following hybrids:

- H_0: Output in the real experiment.
- H_1: First compute $\left(\mathsf{ct}_0, \{\widetilde{D}_j\}_{j\in[n]}, \{\mathsf{ct}_j\}_{j\in[i_2-1]}\right)$ honestly, and then compute ct_{i_2} same as in Figure 22 step 2. Finally it computes $\{\mathsf{ct}_j\}_{j\in[i_2+1,n]}$ from ct_{i_2} honestly by Eval.
- H_2: Output in the simulated experiment.

The above hybrids are indistinguishable because an honestly generated ct_{i_1} or ct_{i_2} is indistinguishable from a simulated one, as we have shown in the single-client case.

To simulate multiple executions, $\mathsf{ihopSim}$ can simply repeat the procedure for every execution. We note that there is no connection between executions except the digest, so they can be simulated separately given the initial digests of every execution. The hybrids go from the real experiment to the simulated experiment by replacing all the honestly generated ct's in one execution by simulated ones, one execution per hybrid.

7.3.5 UMA to Full Security for Multi-hop RAM Scheme

In this section we provide a fully secure multi-hop RAM scheme. We first review Oblivious RAM (ORAM), which was first introduced by Goldreich [Gol87, GO96] and Ostrovsky [Ost90, Ost92, GO96]. We then use ORAM as a compiler to encode the memory and program into a special format that does not reveal the access pattern or data contents during an execution.

Definition 7.14 (Oblivious RAM)**.** *An* Oblivious RAM *scheme consists of two procedures* $(\mathsf{OData}, \mathsf{OProg})$ *with syntax:*

- $(D^*, s^*) \leftarrow \mathsf{OData}(1^\lambda, D)$*: Given a security parameter* λ *and memory* $D \in \{0,1\}^M$ *as input,* OData *outputs the encoded memory* D^* *and encoding key* s^**.*
- $P^* \leftarrow \mathsf{OProg}(1^\lambda, 1^{\log M}, 1^t, P)$*: Given a security parameter* λ*, a memory size* M*, and a program* P *that runs in time* t*,* OProg *outputs an oblivious program* P^* *that can access* D^* *as RAM and takes two inputs* x *and* s^**.*

[15] Notice that different from Figure 22, the simulator here doesn't take P_{i_1} as input, but the simulation can still obtain $(\mathsf{R}/\mathsf{W}^\tau, L^\tau, \mathsf{wData}^\tau)$ for every step $\tau \in [T_{i_1-1}+1, T_{i_1}]$ given D_{i_1} and $\mathsf{MemAccess}_{i_1}$, and that is enough for the simulation.

Efficiency. *We require that the run-time of* OData *should be* $M \cdot \mathsf{polylog}(M) \cdot \mathsf{poly}(\lambda)$*, and the run-time of* OProg *should be* $t \cdot \mathsf{poly}(\lambda) \cdot \mathsf{polylog}(M)$*. Finally, the oblivious program* P^* *itself should run in time* $t' = t \cdot \mathsf{poly}(\lambda) \cdot \mathsf{polylog}(M)$*. Both the new memory size* $M' = |D^*|$ *and the running time* t' *should be efficiently computable from* M, t*, and* λ*.*

Correctness. *Let* $P_1, \ldots, P_\ell$ *be programs running in polynomial times* $t_1, \ldots, t_\ell$ *on memory* D *of size* M*. Let* $x_1, \ldots, x_\ell$ *be the inputs and* λ *be a security parameter. Then we require that:*

$$\Pr[(P_1^*(x_1, s^*), \ldots, P_\ell^*(x_\ell, s^*))^{D^*} = (P_1(x_1), \ldots, P_\ell(x_\ell))^D] = 1$$

where $(D^*, s^*) \leftarrow \mathsf{OData}(1^\lambda, D)$*,* $P_i^* \leftarrow \mathsf{OProg}(1^\lambda, 1^{\log M}, 1^t, P_i)$ *and* $(P_1^*(x_1, s^*), \ldots, P_\ell^*(x_\ell, s^*))^{D^*}$ *indicates running the ORAM programs on* D^* *sequentially.*

Security. *For security, we require that there exists a PPT simulator* S *such that for any sequence of programs* $P_1, \ldots, P_\ell$*, initial memory data* $D \in \{0,1\}^M$*, and inputs* $x_1, \ldots, x_\ell$ *we have that:*

$$(D^*, \mathsf{MemAccess}) \overset{\text{c}}{\approx} S(1^\lambda, 1^M, \{1^{t_i}, y_i\}_{i=1}^\ell)$$

where $(y_1, \ldots, y_\ell) = (P_1(x_1), \ldots, P_\ell(x_\ell))^D$*,* $(D^*, s^*) \leftarrow \mathsf{OData}(1^\lambda, D)$*, and* $\mathsf{MemAccess}$ *corresponds to the access pattern of the CPU-step circuits during the sequential execution of the oblivious programs* $(P_1^*(x_1, s^*), \ldots, P_\ell^*(x_\ell, s^*))^{D^*}$*.*

We prove the following theorem.

Theorem 7.15. *Assume there exists a UMA-secure multi-hop RAM scheme and an ORAM scheme. Then there exists a fully secure multi-hop RAM scheme. Moreover, we give a black-box construction of one given a UMA-secure multi-hop RAM and ORAM scheme.*

Proof. We first give the construction of the scheme itself and then provide a construction of an appropriate simulator to prove security. Let $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{EncData}, \mathsf{InpEnc}, \mathsf{Eval}, \mathsf{Dec})$ be a UMA-secure multi-hop RAM scheme and let $(\mathsf{OData}, \mathsf{OProg})$ be an ORAM scheme. We construct a new multi-hop RAM scheme $(\widehat{\mathsf{Setup}}, \widehat{\mathsf{KeyGen}}, \widehat{\mathsf{EncData}}, \widehat{\mathsf{InpEnc}}, \widehat{\mathsf{Eval}}, \widehat{\mathsf{Dec}})$ as follows:

- $\widehat{\mathsf{Setup}}(1^\lambda)$: Generate crs same as Setup.
- $\widehat{\mathsf{KeyGen}}(1^\lambda)$: Generate $(\mathsf{pk}, \mathsf{sk})$ same as KeyGen.
- $\widehat{\mathsf{EncData}}(\mathsf{crs}, D)$: Execute $(D^*, s^*) \leftarrow \mathsf{OData}(1^\lambda, D)$ followed by $\widetilde{D} \leftarrow \mathsf{EncData}(1^\lambda, D^*)$.
- $\widehat{\mathsf{InpEnc}}(x)$: Execute $(\mathsf{ct}, \text{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$.
- $\widehat{\mathsf{Eval}}\left(i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}, \mathsf{sk}, (P, t), \mathsf{digest}\right)$: Execute $(P^*, t^*) \leftarrow \mathsf{OProg}(1^\lambda, 1^{\log M}, 1^t, P)$ followed by $\mathsf{Eval}\left(i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}, \mathsf{sk}, (P^*[s^*], t^*), \mathsf{digest}\right)$, where P^* has s^* hard-coded inside it.
- $\widehat{\mathsf{Dec}}^{\widetilde{D}_1, \cdots, \widetilde{D}_n}(\text{x\_secret}, \mathsf{ct})$: Output $\mathsf{Dec}^{\widetilde{D}_1, \cdots, \widetilde{D}_n}(\text{x\_secret}, \mathsf{ct})$.

We prove that the construction above given by $(\widehat{\mathsf{Setup}}, \widehat{\mathsf{KeyGen}}, \widehat{\mathsf{EncData}}, \widehat{\mathsf{InpEnc}}, \widehat{\mathsf{Eval}}, \widehat{\mathsf{Dec}})$ is a fully secure multi-hop RAM scheme.

Correctness in a single execution. First we prove correctness for a single execution, and then we will generalize to multiple executions. In a single execution, our goal is to demonstrate that

$$\mathsf{Pr}\left[\widehat{\mathsf{Dec}}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}(\mathrm{x\_secret},\mathsf{ct}_n) = {P_n}^{D_n}\left(\cdots\left({P_1}^{D_1}(x)\right)\cdots\right)\right] = 1,$$

where $\widetilde{D}_i \leftarrow \widehat{\mathsf{EncData}}(1^\lambda, D_i)$, $(\mathsf{ct}_0, \mathrm{x\_secret}) \leftarrow \widehat{\mathsf{InpEnc}}(x)$, $\mathsf{ct}_i \leftarrow \widehat{\mathsf{Eval}}(i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}_{i-1}, \mathsf{sk}, P_i, t_i, \mathsf{digest}_i)$.

By definition, $\widehat{\mathsf{Dec}}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}(\mathrm{x\_secret},\mathsf{ct}_n) = \mathsf{Dec}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}(\mathrm{x\_secret},\mathsf{ct}_n)$. By the correctness of the UMA-secure multi-hop RAM scheme, we have that $\mathsf{Dec}^{\widetilde{D}_1,\cdots,\widetilde{D}_n}(\mathrm{x\_secret},\mathsf{ct}_n) = P_n^*[s_n^*]^{D_n^*}\left(\cdots\left(P_1^*[s_1^*]^{D_1^*}(x)\right)\cdots\right)$. Finally, by the correctness of the ORAM scheme, $P_n^*[s_n^*]^{D_n^*}\left(\cdots\left(P_1^*[s_1^*]^{D_1^*}(x)\right)\cdots\right) = {P_n}^{D_n}\left(\cdots\left({P_1}^{D_1}(x)\right)\cdots\right)$.

Correctness in multiple executions. To prove correctness in multiple executions, we need to show that

$$\mathsf{Pr}\left[\widehat{\mathsf{Dec}}^{\widetilde{D}_1^{(\mathsf{sid})},\cdots,\widetilde{D}_n^{(\mathsf{sid})}}\left(\mathrm{x\_secret}^{(\mathsf{sid})},\mathsf{ct}_n^{(\mathsf{sid})}\right) = {P_n^{(\mathsf{sid})}}^{D_n^{(\mathsf{sid})}}\left(\cdots\left({P_1^{(\mathsf{sid})}}^{D_1^{(\mathsf{sid})}}(x^{(\mathsf{sid})})\right)\cdots\right)\right] = 1,$$

where $\widetilde{D}_i^{(\mathsf{sid})}$ is the resulting garbled database after executing sid-1 homomorphic evaluations, $(\mathsf{ct}_0^{(\mathsf{sid})}, \mathrm{x\_secret}^{(\mathsf{sid})}) \leftarrow \widehat{\mathsf{InpEnc}}(x^{(\mathsf{sid})})$, $\mathsf{ct}_i^{(\mathsf{sid})} \leftarrow \widehat{\mathsf{Eval}}(i, \{\mathsf{pk}_j\}_{j=i+1}^n, \mathsf{ct}_{i-1}^{(\mathsf{sid})}, \mathsf{sk}, P_i^{(\mathsf{sid})}, t_i^{(\mathsf{sid})}, \mathsf{digest}_i^{(\mathsf{sid})})$.

Recall that for correctness of the UMA-secure multi-hop RAM scheme we proved that the after every execution, the resulting garbled database $\widetilde{D}_i^{(\mathsf{sid})}$ corresponds to the output of $\mathsf{Hash}(\mathsf{crs}, D_i^{(\mathsf{sid})})$, where $D_i^{(\mathsf{sid})}$ is the correct D_i^* resulting after previous $\mathsf{sid}-1$ executions in the clear (see Property 2, Section 7.3.2). By correctness of ORAM and the underlying UMA-secure multi-hop RAM scheme, we conclude the correctness in multiple executions.

Security for a single client in a single execution. Server privacy is follows by receiver privacy of oblivious transfer. Now we prove client privacy for a single honest client Q_i in a single execution. More precisely, we prove that there exists a PPT simulator $\mathsf{ihopSim}$ such that, for any set of databases $\{D_j\}_{j\in[n]}$, any sequence of compatible programs $P_1, \cdots, P_n$ running time $t_1, \cdots, t_n$ and input x, the outputs of the following two experiments are computational indistinguishable:

Real experiment

- $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow \mathsf{KeyGen}(1^\lambda)$ for $\forall j \in [n]$.
- $\widetilde{D}_j = (\mathsf{digest}_j, \hat{\mathsf{D}}_j) \leftarrow \mathsf{EncData}(\mathsf{crs}, D_j)$ for $\forall j \in [n]$.
- $(\mathsf{ct}_0, \mathrm{x\_secret}) \leftarrow \mathsf{InpEnc}(x)$.
- $\mathsf{ct}_j \leftarrow \mathsf{Eval}\left(j, \{\mathsf{pk}_k\}_{k=j+1}^n, \mathsf{ct}_{j-1}, \mathsf{sk}_j, (P_j, t_j), \mathsf{digest}_j\right)$ for $\forall j \in [n]$.
- Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

Simulated experiment

- $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{ihopSim}(1^\lambda, i)$.
- $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow \mathsf{KeyGen}(1^\lambda; r_j)$ for $\forall j \in [n] \setminus \{i\}$. Here, r_j are uniform random coins.

- $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right) \leftarrow \mathsf{ihopSim}\left(\mathsf{crs}, x, \{\mathsf{pk}_j, \mathsf{sk}_j, t_j\}_{j\in[n]}, \{D_j, P_j, r_j\}_{j\in[n]\setminus\{i\}}, 1^{M_i}, y_i\right)$, where $y_i = {P_i}^{D_i}\left(\cdots\left({P_1}^{D_1}(x)\right)\cdots\right)$.
- Output $\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}$.

We let OSim be the ORAM simulator, and USim be the simulator for the UMA-secure multi-hop RAM scheme. We describe the two phases of $\mathsf{ihopSim}$. In the first phase, $\mathsf{ihopSim}$ generates the keys of honest client Q_i as $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KeyGen}(1^\lambda)$. In the second phase, $\mathsf{ihopSim}$ proceeds as follows.

1. Compute $(D_i^*, \mathsf{MemAccess}_i) \leftarrow \mathsf{OSim}(1^\lambda, 1^{M_i}, 1^{t_i}, y_i)$.
2. Compute (D_j^*, s_j^*) from $\widehat{\mathsf{EncData}}$ and P_j^* from $\widehat{\mathsf{Eval}}$ for every $j \in [n] \setminus \{i\}$.
3. Compute $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right) \leftarrow \mathsf{USim}(\mathsf{crs}, x, \{\mathsf{pk}_j, \mathsf{sk}_j, t_j^*\}_{j\in[n]}, \{D_j^*, P_j^*[s_j^*], r_j\}_{j\in[n]\setminus\{i\}}, D_i^*, \mathsf{MemAccess}_i, y_i)$.
4. Output $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right)$.

We now prove the output of the simulator is computationally indistinguishable from the real distribution.

- H_0: Output of the real experiment.
- H_1: Compute (D_j^*, s_j^*) from $\widehat{\mathsf{EncData}}$ and P_j^* from $\widehat{\mathsf{Eval}}$ for every $j \in [n]\setminus\{i\}$. Use the honestly generated (D_i^*, s_i^*) from $\widehat{\mathsf{EncData}}$ and P_i^* from $\widehat{\mathsf{Eval}}$ to execute the program ${P_i^*[s_i^*]}^{D_i^*}\left(\cdots\left({P_1^*[s_1^*]}^{D_1^*}(x)\right)\cdots\right)$ and obtain y_i and a sequence of memory accesses $\mathsf{MemAccess}_i$. Run $\left(\mathsf{ct}_0, \{\widetilde{D}_j, \mathsf{ct}_j\}_{j\in[n]}\right) \leftarrow \mathsf{USim}(\mathsf{crs}, x, \{\mathsf{pk}_j, \mathsf{sk}_j, t_j^*\}_{j\in[n]}, \{D_j^*, P_j^*[s_j^*], r_j\}_{j\in[n]\setminus\{i\}}, D_i^*, \mathsf{MemAccess}_i, y_i)$ and output.
  Since $(D_i^*, \mathsf{MemAccess}_i)$ is the same as the real execution, the indistinguishability of this hybrid and H_0 follows from UMA-security of the underlying multi-hop RAM scheme.
- H_2: Output of the simulated experiment. The only thing that differs in H_1 and H_2 is how we generate D_i^* and $\mathsf{MemAccess}_i$. In H_1 they are generated honestly and in H_2 they are generated by OSim. $\mathsf{H}_1 \overset{\text{c}}{\approx} \mathsf{H}_2$ follows from the security of ORAM.

Security for multiple clients and multiple executions. Similar as in the proof of UMA security, the above proof can be naturally extended to provide security for multiple clients and many executions. For example in the case of two clients Q_{i_1} and Q_{i_2}, $\mathsf{ihopSim}$ first computes $\left(\mathsf{ct}_0, \{\widetilde{D}_j\}_{j\in[n]}, \{\mathsf{ct}_j\}_{j\in[i_1-1]}\right)$ honestly, then simulates ct_{i_1} same as above as if there were only one honest client Q_{i_1}. It then computes $\{\mathsf{ct}_j\}_{j\in[i_1+1,i_2-1]}$ from ct_{i_1} by $\widehat{\mathsf{Eval}}$, and simulates ct_{i_2} same as above as if there were only one honest client Q_{i_2}. Notice that when simulating ct_{i_2}, similar as in the UMA-secure scenario, $\mathsf{ihopSim}$ cannot generate $(D_{i_1}^*, s_{i_1}^*, P_{i_1}^*)$ honestly. Instead it will use the simulated $(D_{i_1}^*, \mathsf{MemAccess}_{i_1})$ generated from OSim, and that is enough for the simulation. Finally it computes $\{\mathsf{ct}_j\}_{j\in[i_2+1,n]}$ from ct_{i_2} by $\widehat{\mathsf{Eval}}$. To show this is indistinguishable from the real execution, we consider the following hybrids:

- H_0: Output in the real experiment.
- H_1: First compute $\left(\mathsf{ct}_0, \{\widetilde{D}_j\}_{j\in[n]}, \{\mathsf{ct}_j\}_{j\in[i_2-1]}\right)$ honestly, and then compute ct_{i_2} same as above as if there were only one honest client Q_{i_2}. Finally it computes $\{\mathsf{ct}_j\}_{j\in[i_2+1,n]}$ from ct_{i_2} honestly by $\widehat{\mathsf{Eval}}$.
- H_2: Output in the simulated experiment.

The above hybrids are indistinguishable because an honestly generated ct_{i_1} or ct_{i_2} is indistinguishable from a simulated one, as we have shown in the single-client case.

To simulate multiple executions, $\mathsf{ihopSim}$ should first use OSim to simulate $(D_i^*, \mathsf{MemAccess}_i)$ for every honest client Q_i in all executions, and then repeat the above procedure for every execution. In the hybrids, we start from the real execution and first replace the honestly generated ct's by simulated ones while using honestly generated $(D_i^*, \mathsf{MemAccess}_i)$, and this step follows from the UMA-security of the underlying multi-hop RAM scheme. Afterwards we replace the honestly generated $(D_i^*, \mathsf{MemAccess}_i)$ by the output of OSim, and this step follows from the security of ORAM supporting multiple executions. □

Acknowledgement

We thank the anonymous reviewers of CRYPTO 2017 for their helpful suggestions in improving this paper. We also thank Yuval Ishai for useful discussions.

References

[ADT11] Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. (If) size matters: Size-hiding private set intersection. In Dario Catalano, Nelly Fazio, Rosario Gennaro, and Antonio Nicolosi, editors, *PKC 2011: 14th International Conference on Theory and Practice of Public Key Cryptography*, volume 6571 of *Lecture Notes in Computer Science*, pages 156–173, Taormina, Italy, March 6–9, 2011. Springer, Heidelberg, Germany.

[AIKW13] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate or how to compress garbled circuits keys. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 166–184, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[AIR01] William Aiello, Yuval Ishai, and Omer Reingold. Priced oblivious transfer: How to sell digital goods. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 119–135, Innsbruck, Austria, May 6–10, 2001. Springer, Heidelberg, Germany.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference*

on Computer and Communications Security, pages 535–548, Berlin, Germany, November 4–8, 2013. ACM Press.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 326–349, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.

[Bea96] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *28th Annual ACM Symposium on Theory of Computing*, pages 479–488, Philadephia, PA, USA, May 22–24, 1996. ACM Press.

[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, Portland, OR, USA, June 14–17, 2015. ACM Press.

[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, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Heidelberg, Germany.

[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 12: 19th Conference on Computer and Communications Security*, pages 784–796, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

[BPMW16] Florian Bourse, Rafaël Del Pino, Michele Minelli, and Hoeteck Wee. FHE circuit privacy almost for free. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 62–89, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany.

[BSCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[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, Palm Springs, CA, USA, October 22–25, 2011. IEEE Computer Society Press.

[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, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany.

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for RAM programs. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM Symposium on Theory of Computing*, pages 429–437, Portland, OR, USA, June 14–17, 2015. ACM Press.

[CHK04] Ran Canetti, Shai Halevi, and Jonathan Katz. Chosen-ciphertext security from identity-based encryption. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 207–222, Interlaken, Switzerland, May 2–6, 2004. Springer, Heidelberg, Germany.

[COV15] Melissa Chase, Rafail Ostrovsky, and Ivan Visconti. Executable proofs, input-size hiding secure computation and a new ideal world. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 532–560, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

[CS98] Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[CS02] Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 45–64, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

[CV12] Melissa Chase and Ivan Visconti. Secure database commitments and universal arguments of quasi knowledge. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 236–254, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[DG17] Nico Döttling and Sanjam Garg. Identity-based encryption from the diffie hellman assumption. CRYPTO 2017 (to appear), 2017.

[DS16] Léo Ducas and Damien Stehlé. Sanitization of FHE ciphertexts. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 294–310, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[FLS90] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple non-interactive zero knowledge proofs based on a single random string (extended abstract). In *31st Annual Symposium*

on Foundations of Computer Science, pages 308–317, St. Louis, Missouri, October 22–24, 1990. IEEE Computer Society Press.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press.

[GGH13a] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 1–17, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[GGH+13b] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual Symposium on Foundations of Computer Science*, pages 40–49, Berkeley, CA, USA, October 26–29, 2013. IEEE Computer Society Press.

[GGMP16] Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty RAM computation in constant rounds. In *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I*, pages 491–520, 2016.

[GGSW13] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 467–476, Palo Alto, CA, USA, June 1–4, 2013. ACM Press.

[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, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *55th Annual Symposium on Foundations of Computer Science*, pages 404–413, Philadelphia, PA, USA, October 18–21, 2014. IEEE Computer Society Press.

[GHV10] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. i-Hop homomorphic encryption and rerandomizable Yao circuits. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 155–172, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12: 19th Conference on Computer and Communications Security*, pages 513–524, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 536–553, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[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, Berkeley, CA, USA, October 17–20, 2015. IEEE Computer Society Press.

[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, Portland, OR, USA, June 14–17, 2015. ACM Press.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[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, New York City, NY, USA, May 25–27, 1987. ACM Press.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Non-interactive zaps and new techniques for NIZK. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 97–111, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[HK12] Shai Halevi and Yael Tauman Kalai. Smooth projective hashing and two-message oblivious transfer. *Journal of Cryptology*, 25(1):158–193, January 2012.

[HW15] Pavel Hubacek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In Tim Roughgarden, editor, *ITCS 2015: 6th Innovations in Theoretical Computer Science*, pages 163–172, Rehovot, Israel, January 11–13, 2015. Association for Computing Machinery.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume

2729 of *Lecture Notes in Computer Science*, pages 145–161, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany.

[IKO+11] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 406–425, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

[IP07] Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In Salil P. Vadhan, editor, *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 575–594, Amsterdam, The Netherlands, February 21–24, 2007. Springer, Heidelberg, Germany.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Heidelberg, Germany.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *20th Annual ACM Symposium on Theory of Computing*, pages 20–31, Chicago, IL, USA, May 2–4, 1988. ACM Press.

[KK13] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 54–70, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM Symposium on Theory of Computing*, pages 419–428, Portland, OR, USA, June 14–17, 2015. ACM Press.

[LNO13] Yehuda Lindell, Kobbi Nissim, and Claudio Orlandi. Hiding the input-size in secure two-party computation. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology – ASIACRYPT 2013, Part II*, volume 8270 of *Lecture Notes in Computer Science*, pages 421–440, Bengalore, India, December 1–5, 2013. Springer, Heidelberg, Germany.

[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, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

[MRK03] Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *44th Annual Symposium on Foundations of Computer Science*, pages 80–91, Cambridge, MA, USA, October 11–14, 2003. IEEE Computer Society Press.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In S. Rao Kosaraju, editor, *12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 448–457, Washington, DC, USA, January 7–9, 2001. ACM-SIAM.

[OPP14] Rafail Ostrovsky, Anat Paskin-Cherniavsky, and Beni Paskin-Cherniavsky. Maliciously circuit-private FHE. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 536–553, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[OPWW15] Tatsuaki Okamoto, Krzysztof Pietrzak, Brent Waters, and Daniel Wichs. New realizations of somewhere statistically binding hashing and positional accumulators. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 121–145, Auckland, New Zealand, November 30 – December 3, 2015. Springer, Heidelberg, Germany.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th Annual ACM Symposium on Theory of Computing*, pages 294–303, El Paso, TX, USA, May 4–6, 1997. ACM Press.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd Annual ACM Symposium on Theory of Computing*, pages 514–523, Baltimore, MD, USA, May 14–16, 1990. ACM Press.

[Ost92] Rafail Ostrovsky. *Software Protection and Simulation On Oblivious RAMs*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1992.

[Rab81] Michael O. Rabin. How to exchange secrets with oblivious transfer, 1981.

[Vil12] Jorge Luis Villar. Optimal reductions of some decisional problems to the rank problem. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 80–97, Beijing, China, December 2–6, 2012. Springer, Heidelberg, Germany.

[WHC+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14: 21st Conference on Computer and Communications Security*, pages 191–202, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press.