Secure Computation with Sublinear Amortized Work

Dov Gordon* Jonathan Katz† Vladimir Kolesnikov‡ Tal Malkin*
Mariana Raykova* Yevgeniy Vahlis§

Abstract

Traditional approaches to secure computation begin by representing the function f being computed as a circuit. For any function f that depends on each of its inputs, this implies a protocol with complexity at least linear in the input size. In fact, linear running time is *inherent* for secure computation of non-trivial functions, since each party must "touch" every bit of their input lest information about other party's input be leaked. This seems to rule out many interesting applications of secure computation in scenarios where at least one of the inputs is huge and sublinear-time algorithms can be utilized in the insecure setting; *private database search* is a prime example.

We present an approach to secure two-party computation that yields sublinear-time protocols, in an amortized sense, for functions that can be computed in sublinear time on a random access machine (RAM). Furthermore, a party whose input is "small" is required to maintain only small state. We provide a generic protocol that achieves the claimed complexity, based on any oblivious RAM and any protocol for secure two-party computation. We then present an optimized version of this protocol, where generic secure two-party computation is used only for evaluating a small number of simple operations.

1 Introduction

Consider the natural task of searching over a sorted database of n elements. Using binary search, this can be done in time $O(\log n)$. Next consider a secure version of this problem where a client holds an item and wants to learn whether this item is present in a database held by a server, with neither party learning anything else. Applying standard protocols for secure computation to this task, we would find that they begin by expressing the computation as a (binary or arithmetic) circuit of size at least n, resulting in protocols of complexity $\Omega(n)$. Moreover, it is well known that this is inherent. Namely, in any secure protocol for this problem the server must "touch" every bit of its database; otherwise, the server learns some information about the client's input from the portions of its database that were never touched.

Tracing the sources of the inefficiency, one may notice two opportunities for improvement:

- Any circuit computing a non-trivial function f on inputs of length n must have size $\Omega(n)$. On the other hand, many interesting functions can be computed in *sublinear* time on a

*Columbia University. Email: {gordon,tal,mariana}@cs.columbia.edu

†Dept. of Computer Science, University of Maryland. Email: jkatz@cs.umd.edu. This work was supported by NSF award #1111599 and DARPA.

‡Alcatel-Lucent Bell Labs. Email: kolesnikov@research.bell-labs.com

§AT&T Security Research Center. Email: evahlis@att.com This work was done while at Columbia University.

random-access machine (RAM). Thus, it would be desirable to have protocols for generic secure computation that use RAMs — rather than circuits — as their starting point.

- The fact that linear work (or more) is inherent for secure computation of any non-trivial function f only applies when f is computed *once*. However, it does not rule out the possibility of doing better, in an amortized sense, when the parties compute the function *several* times.

Inspired by the above, we explore scenarios in which secure computation with *sublinear* amortized work is possible. We focus on a setting where a client and server repeatedly evaluate some function f on their respective inputs, maintaining state across these executions, with the server's (huge) input D assumed to be fixed throughout and the client's (small) input x changing each time f is evaluated. Our main result is as follows:

Theorem 1 (Informal)**.** *Suppose $f(x, D)$ can be computed using time t and space s in the RAM model of computation. Then there is a secure two-party protocol computing f in which the client and server run in amortized time $O(t) \cdot \mathsf{polylog}(s)$, the client uses space $O(\log(s))$, and the server uses space $O(s \cdot \mathsf{polylog}(s))$.*

We show a generic protocol achieving the above bounds based on any oblivious RAM (ORAM) construction and any secure two-party computation protocol. The resulting protocol demonstrates the feasibility of sublinear-complexity secure computation, and serves as a useful template for the more efficient solution given by our second, optimized construction. In our optimized protocol we use a specific ORAM construction, and design the protocol so that generic secure computation is utilized only for a small number of simple operations. The resulting protocol provides a much more efficient construction.

1.1 Technical Overview

Our starting point is the ORAM primitive [9], which allows a client (with small memory) to perform RAM computations using the (large) memory of a remote untrusted server. At a high level, the client stores encrypted memory cells on the server and then emulates a RAM computation of some function f by replacing each read/write access of the original RAM computation with a series of read/write accesses of the remote data, such that the actual access pattern of the client remains hidden from the server. Results of Goldreich and Ostrovsky [9], since improved by others (see Section 1.2), show that if f can be computed on a RAM in t steps and space s (see Section 2.1 for our formal model of RAM algorithms), then it can be computed on an ORAM in $t \cdot \mathsf{polylog}(s)$ steps while using $s \cdot \mathsf{polylog}(s)$ space at the server.

In our setting, ORAM suggests a candidate protocol for computing f with sublinear amortized overhead. Say the server holds input D and the client wants to compute $f(x_1, D), f(x_2, D), \ldots$ for a sequence of inputs $x_1, x_2, \ldots$. The client and server (interactively) pre-process D as required for the ORAM construction. This pre-processing step will take (at least) time linear in $|D|$, but will be amortized over several computations of f. In each computation, the client and server run the ORAM protocol until the client learns the output. If f can be evaluated in t steps on a RAM, then each such evaluation can be done in time $t \cdot \mathsf{polylog}(|D|)$.

The above protocol provides "one-sided security," in that it ensures privacy of the client's input against a semi-honest server. (Though, in fact, it was already shown by Goldreich and Ostrovsky [9] how malicious behavior by the server can be addressed.) However, it provides no security guarantees

for the server! We can address this by running each ORAM instruction inside a (standard) secure two-party computation protocol, with intermediate states being shared between the client and server. This is the basic idea behind our generic construction, as described in detail in Section 3.

In our second construction, we optimize the efficiency of the above by building on the *specific* ORAM construction of Goldreich and Ostrovsky [9] and aiming to minimize our reliance on generic secure computation. In particular, we make sure that generic secure computation is applied only to very small circuits. To reduce our reliance on generic secure computation, we design a protocol for obliviously evaluation of a pseudorandom function (PRF) where both the key and the input/output are shared. With careful attention to detail, and several important changes to the underlying ORAM protocol, we end up with a much more efficient protocol. We describe this in detail in Section 4.

In the course of proving security of our protocol, we identified a security issue with some previous ORAM constructions; our observations impact the security of the Pinkas-Reinman protocol [26] as well as the analysis (and the security for some parameter settings) of the Goldreich-Ostrovsky protocol [9].[1] See Section 4.3 for further discussion.

1.2 Related Work

Some previous works achieve amortized sublinear computation for *specific* problems: e.g., PIR [2, 15], or certain approximation problems [5, 13, 6, 18, 21, 32]. In contrast, we address secure computation of *general* functionalities, achieving (amortized) sublinear-time secure computation whenever sublinear-time algorithms are possible in a RAM model of computation.

As discussed in the introduction, most existing approaches for generic secure computation of a function f begin by representing f as a boolean or arithmetic circuit. Models of computation other than circuits have been used previously as the starting point for secure protocols, including branching programs [24, 16, 14, 7], ordered binary decision diagrams [19], and circuits with lookup tables [24, 7]. The work of [24, 7], in particular, uses a branching-program representation to construct protocols with sublinear *communication* complexity for certain functions. However, they do not achieve sublinear computational complexity. (In fact, in some cases their protocols require exponential work.)

Damgård et al. [4] also observe that ORAM can be used for secure computation. In their approach, which they only briefly sketch, players secret-share the entire (super-linear) state of the ORAM and execute the ORAM program on their joint state; thus, applying their approach to our setting would require the client to use an amount of memory equal to that of the server. Most importantly, they make no attempt to optimize the concrete efficiency of their protocol, which is a major contribution of our work.

In our efficient solution we build on the Goldreich-Ostrovsky ORAM protocol [9]. The ORAM constructions of Damgård et al. [4] and of Ajtai [1] focused on removing computational assumptions, but yield less efficient solutions. The works of Williams and Sion [30] and Williams et al. [31] both require significant storage at the client, which we consider unacceptable in the applications we envision. We had initially attempted to build on the Pinkas-Reinman ORAM [26] (which is more efficient than that of Goldreich-Ostrovsky), but in the course of our work we discovered several flaws in their protocol, one of which is discussed in Section 4.3. The flaws in their protocol were independently discovered by other researchers [11, 20], and (subsequent to our work) Pinkas and

[1] Independent of (but prior to) our work, similar flaws have been pointed out by others [11, 20].

Reinman have developed a fix for their scheme [27]. Subsequent to our work, several improved constructions of ORAM have appeared [11, 28, 20, 29, 12] and we have not investigated whether these would lead to more efficient protocols for our purposes.

2 Preliminaries

2.1 Random Access Machines

In this work, we focus on RAM programs for computing a function $f(x, D)$, where x is a "small" input that can be read in its entirety and D is a larger array that is accessed via a sequence of read and write instructions. Any such instruction $I \in (\{\mathsf{read}, \mathsf{write}\} \times \mathbb{N} \times \{0,1\}^*)$ takes the form (write, v, d) ("write data element d in location/address v") or $(\mathsf{read}, v, \perp)$ ("read the data element stored at location v"). We also assume a designated "stop" instruction of the form (stop, z) that indicates termination of the RAM protocol with output z.

Formally, a RAM program is defined by a "next instruction" function Π which, given its current state and a value d (that will always be equal to the last-read element), outputs the next instruction and an updated state. Thus if D is an array of n entries, each ℓ bits long, we can view execution of a RAM program as follows:

- Set $\mathsf{state}_\Pi = (1^{\log n}, 1^\ell, \mathsf{start}, x)$ and $d = 0^\ell$. Then until termination do:
  1. Compute $(I, \mathsf{state}'_\Pi) = \Pi(\mathsf{state}_\Pi, d)$. Set $\mathsf{state}_\Pi = \mathsf{state}'_\Pi$.
  2. If $I = (\mathsf{stop}, z)$ then terminate with output z.
  3. If $I = (\mathsf{write}, v, d')$ then set $D[v] = d'$.
  4. If $I = (\mathsf{read}, v, \perp)$ then set $d = D[v]$.

(We stress that the contents of D change during the course of the execution.) To make things non-trivial, we require that the size of state_Π, and the space required to compute Π, is polynomial in $\log n, \ell$, and $|x|$. (Thus, if we view a client running Π and issuing instructions to a server storing D, the space used by the client is small.)

We allow the possibility for D to grow beyond n entries, so the RAM program may issue write (and then read) instructions for indices greater than n. The space complexity of a RAM program on inputs x, D is the maximum number of entries used by D during the course of the execution. The time complexity of a RAM program on the same inputs is the number of instructions issued in the execution as described above. For our application, we do not want the running time of a RAM program to reveal anything about the inputs. Thus, we will assume that any RAM program has associated with it a polynomial t such that the running time on x, D is exactly $t(n, \ell, |x|)$.

2.2 Oblivious RAM

We view an oblivious-RAM (ORAM) construction as a mechanism that simulates read/write access to an underlying (virtual) array D via accesses to some (real) array $\tilde{D}$; "obliviousness" means that no information about the virtual accesses to D is leaked by observation of the real accesses to $\tilde{D}$. An ORAM construction can be used to compile any RAM program into an oblivious version of that program.

An ORAM construction consists of two algorithms OI and OE for initialization and execution, respectively. OI initializes some state $\mathsf{state}_{\mathsf{oram}}$ that is used (and updated by) OE. The second

algorithm, OE, is used to compile a single read/write instruction I (on the virtual array D) into a sequence of read/write instructions $\tilde{I}_1, \tilde{I}_2, \ldots$ to be executed on (the real array) $\tilde{D}$. The compilation of an instruction I into $\tilde{I}_1, \tilde{I}_2, \ldots$, can be adaptive; i.e., instruction $\tilde{I}_j$ may depend on the values read in some prior instructions. To capture this, we define an iterative procedure called $\mathsf{doInstruction}$ that makes repeated use of OE. Given a read/write instruction I, we define $\mathsf{doInstruction}(\mathsf{state}_{\mathsf{oram}}, I)$ as follows:

- Set $d = 0^\ell$. Then until termination do:
  1. Compute $(\tilde{I}, \mathsf{state}'_{\mathsf{oram}}) \leftarrow \mathsf{OE}(\mathsf{state}_{\mathsf{oram}}, I, d)$, and set $\mathsf{state}_{\mathsf{oram}} = \mathsf{state}'_{\mathsf{oram}}$.
  2. If $\tilde{I} = (\mathsf{done}, z)$ then terminate with output z.
  3. If $\tilde{I} = (\mathsf{write}, v, d')$ then set $\tilde{D}[v] = d'$.
  4. If $\tilde{I} = (\mathsf{read}, v, \perp)$ then set $d = \tilde{D}[v]$.

If I was a read instruction with $I = (\mathsf{read}, v, \perp)$, then the final output z should be the value "written" at $D[v]$. (See below, when we define correctness.)

Correctness. We define correctness of an ORAM construction in the natural way. Let $I_1, \ldots, I_k$ be any sequence of instructions with $I_k = (\mathsf{read}, v, \perp)$, and $I_j = (\mathsf{write}, v, d)$ the last instruction that writes to address v. If we start with $\tilde{D}$ initialized to empty and then run $\mathsf{state}_{\mathsf{oram}} \leftarrow \mathsf{OI}(1^\kappa)$ followed by $\mathsf{doInstruction}(I_1), \ldots, \mathsf{doInstruction}(I_1)$, then the final output will be equal to d with all but negligible probability.

Security. Intuitively, the security requirement is that for any two equal-length sequences of RAM instructions, the (real) access patterns generated by those instructions will be indistinguishable. We will use the standard definition from the literature, which assumes the two instruction sequences are chosen in advance.[2] Formally, let $\mathcal{ORAM} = \langle \mathsf{OI}, \mathsf{OE} \rangle$ be an ORAM construction and consider the following experiment:

Experiment $\mathsf{ExpAPH}_{\mathcal{ORAM},\mathsf{Adv}}(\kappa, b)$:

1. The adversary Adv outputs two sequences of queries $(\mathbf{I}^0, \mathbf{I}^1)$, where $\mathbf{I}^0 = \{I_1^0, \ldots, I_k^0\}$ and $\mathbf{I}^1 = \{I_1^1, \ldots, I_k^1\}$ for arbitrary k.
2. Run $\mathsf{state}_{\mathsf{oram}} \leftarrow \mathsf{OI}(1^\kappa)$; initialize $\tilde{D}$ to empty; and then execute $\mathsf{doInstruction}(\mathsf{state}_{\mathsf{oram}}, I_1^b)$, $\ldots$, $\mathsf{doInstruction}(\mathsf{state}_{\mathsf{oram}}, I_k^b)$ (note that $\mathsf{state}_{\mathsf{oram}}$ is updated each time $\mathsf{doInstruction}$ is run). The adversary is allowed to observe $\tilde{D}$ the entire time.
3. Finally, the adversary outputs a guess $b' \in \{0, 1\}$. The experiment evaluates to 1 iff $b' = b$.

Definition 1. *An ORAM construction $\mathcal{ORAM} = \langle \mathsf{OI}, \mathsf{OE} \rangle$ is* access-pattern hiding *against honest-but-curious adversaries if for every* PPT *adversary* Adv *the following probability, taken over the randomness of the experiment and $b \in_R \{0, 1\}$, is negligible:*

$$\left| \Pr\left[\mathsf{ExpAPH}_{\mathcal{ORAM},\mathsf{Adv}}(1^\kappa, b) = 1\right] - \frac{1}{2} \right|.$$

[2] It appears that existing ORAM constructions are secure even if the adversary is allowed to adaptively choose the next instruction after observing the access pattern on $\tilde{D}$ caused by the previous instruction. Since this has not been claimed by any ORAM construction in the literature, we do not define it.

2.3 Secure Computation

We focus on the setting where a server holds a (large) database D and a client wants to repeatedly compute $f(x, D)$ for different inputs x; moreover, f may also change the contents of D itself. We allow the client to keep (short) state between executions, and the server will keep state that reflects the (updated) contents of D.

For simplicity, we focus only on the two-party (client/server) setting in the semi-honest model but it is clear that our definitions can be extended to the multi-party case with malicious adversaries.

Definition of security. We use a standard simulation-based definition of secure computation [10], comparing a real execution to that of an ideal (reactive) functionality F. In the ideal execution, the functionality maintains the updated state of D on behalf of the server. We also allow F to take a description of f as input (which allows us to consider a single ideal functionality).

The real-world execution proceeds as follows. An environment $\mathcal{Z}$ initially gives the server a database $D = D^{(1)}$, and the client and server then run protocol Π_f (with the client using input init and the server using input D) that ends with the client and server each storing some state that they will maintain (and update) throughout the subsequent execution. In the ith iteration ($i = 1, \ldots$), the environment gives x_i to the client; the client and server then run protocol Π_f (with the client using its state and input x_i, and the server using its state) with the client receiving output out_i. The client sends out_i to $\mathcal{Z}$, thus allowing adaptivity in $\mathcal{Z}$'s next input selection x_{i+1}. At some point, $\mathcal{Z}$ terminates execution by sending a special end message to the players. At this time, an honest player simply terminates execution; a corrupted player sends its entire view to $\mathcal{Z}$.

For a given environment $\mathcal{Z}$ and some fixed value κ for the security parameter, we let $\text{REAL}_{\Pi_f,\mathcal{Z}}(\kappa)$ be the random variable denoting the output of $\mathcal{Z}$ following the specified execution in the real world.

In the ideal world, we let F be a trusted functionality that maintains state throughout the execution. An environment $\mathcal{Z}$ initially gives the server a database $D = D^{(1)}$, which the server in turn sends to F. In the ith iteration ($i = 1, \ldots$), the environment gives x_i to the client who sends this value to F. The trusted functionality then computes

$$(\mathsf{out}_i, D^{(i+1)}) \leftarrow f(x_i, D^{(i)}),$$

and sends out_i to the client. (Note the server does not learn anything from the execution, neither about out_i nor about the updated contents of D.) The client ends out_i to $\mathcal{Z}$. At some point, $\mathcal{Z}$ terminates execution by sending a special end message to the players. The honest player simply terminates execution; the corrupted player may send an arbitrary function of its entire view to $\mathcal{Z}$.

For a given environment $\mathcal{Z}$, some fixed value κ for the security parameter, and some algorithm $\mathcal{S}$ being run by the corrupted party, we let $\text{IDEAL}_{F,\mathcal{S},\mathcal{Z}}(\kappa)$ be the random variable denoting the output of $\mathcal{Z}$ following the specified execution.

Definition 2. *We say that* protocol Π_f securely computes f *if there exists a probabilistic polynomial-time ideal-world adversary $\mathcal{S}$ (run by the corrupted player) such that for all non-uniform, polynomial-time environments $\mathcal{Z}$ there exists a negligible function* negl *such that*

$$\left|\Pr\left[\text{REAL}_{\Pi_f,\mathcal{Z}}(\kappa) = 1\right] - \Pr\left[\text{IDEAL}_{F,\mathcal{S},\mathcal{Z}}(\kappa) = 1\right]\right| \leq \mathsf{negl}(\kappa).$$

Remark: "adaptivity" in the choice of the $\{x_i\}$. In the "standard" ideal-world definition of reactive computation, a corrupted player (either client or server) would give its entire view to the environment each time the functionality F is accessed. Here, however, we allow the players to give

Secure initialization protocol

Input: The server has input D of length n, and the client does not use its input in this stage.

Protocol:

1. The participants run a secure computation of $\mathsf{OI}(1^\kappa)$, which results in each party receiving a secret share of the initial ORAM state. We denote this by $[\mathsf{state}_{\mathsf{oram}}]$.
2. For $i = 1, \ldots, n$ do
   (a) The server creates instruction $I = (\mathsf{write}, v, D[v]))$ and secret shares it with the client. We denote the resulting sharing by $[I]$.
   (b) The parties execute $([\mathsf{state}'_{\mathsf{oram}}], [\perp]) \leftarrow \mathsf{doInstruction}([\mathsf{state}_{\mathsf{oram}}], [I])$ (see Figure 3), and set $[\mathsf{state}_{\mathsf{oram}}] \leftarrow [\mathsf{state}'_{\mathsf{oram}}]$.

Figure 1: Secure initialization protocol π_{Init}.

its view to the environment only at the end of the entire execution. This seems to be reasonable in the semi-honest setting we consider, where a subsequent input x_{i+1} should have no dependence on the view of the ith protocol execution. (On the other hand, we do allow x_{i+1} to depend on $\mathsf{out}_1, \ldots, \mathsf{out}_i$, a dependence that is realistic.) In fact, our protocols satisfy the stronger notion (where a corrupted server gives its view to $\mathcal{Z}$ after each execution of the protocol in the real world, and gives an arbitrary function of its view to $\mathcal{Z}$ after each iteration in the ideal world) as long as the underlying ORAM construction they use satisfies the adaptive notion of security discussed in footnote 2. (To the best of our knowledge, this property has not been considered in any prior work on ORAM. Nevertheless, we conjecture that all known constructions are secure even under adaptive choice of instructions.)

3 Generic Construction

In this section we present our generic solution to the amortized sublinear secure computation problem. The construction is based in a black-box manner on any ORAM scheme and any secure two-party computation protocol. While our second protocol, which we present in Section 4, is substantially more efficient than any specific instantiation of the protocol in this section, this generic protocol is conceptually simple and clean, demonstrates theoretical feasibility, and provides a good overview of our overall approach.

Our first observation is that the server can store his *own* data in his *own* ORAM structure: the security definition of ORAM in Section 2 guarantees security against a semi-honest server even when he knows the data content completely. This allows us to give the client access to the server's data without violating client privacy, and without requiring him to store a secret-sharing of the entire database. We now only need to ensure that the client does not learn any information either. Our second observation is that this can be achieved, at a cost *independent* of the size of D, by always secret-sharing the client's state with the server (this keeps the client oblivious), and by facilitating the ORAM operations using standard secure computation techniques on their joint state. More specifically, we will use MPC to compute each next-instruction function in RAM, and then further to compile each RAM instruction into a sequence of ORAM instructions. ORAM instructions are then reconstructed and executed by the server (they can safely be shown to the server), and the result of the RAM instructions is secret-shared with the client to form part of the updated client's

Secure evaluation protocol π_f

Inputs: The server has array $\tilde{D}$ and the client has input $1^{\log n}, 1^\ell$, and x. Also the server and the client have secret shares of an ORAM state, denoted $[\mathsf{state}_{\mathsf{oram}}]$.
Shared input: A RAM program defined by the "next-instruction function" Π.

Protocol:

1. The client sets $\mathsf{state}_\Pi = (1^{\log n}, 1^\ell, \mathsf{start}, x)$ and $d = 0^\ell$ and secret shares both values with the server; we denote the shared values by $[\mathsf{state}_\Pi]$ and $[d]$, respectively.
2. Do:
   (a) The parties securely compute $([I], [\mathsf{state}'_\Pi]) \leftarrow \Pi([\mathsf{state}_\Pi], [d])$, and set $[\mathsf{state}_\Pi] = [\mathsf{state}'_\Pi]$.
   (b) The parties perform a secure computation to check whether $\mathsf{state}_\Pi = (\mathsf{stop}, z)$. If so, break.
   (c) The parties execute $([\mathsf{state}'_{\mathsf{oram}}], [d']) \leftarrow \mathsf{doInstruction}([\mathsf{state}_{\mathsf{oram}}], [I])$. They set $[\mathsf{state}_{\mathsf{oram}}] = [\mathsf{state}'_{\mathsf{oram}}]$ and $[d] = [d']$.
3. The server sends its share of $[\mathsf{state}_\Pi]$ and $[d]$ to the client, who recovers the output z.

Output: The client outputs z.

Figure 2: Secure evaluation of a RAM program.

state. The players then use the updated state to continue with the evaluation of the next RAM instruction.

In more technical detail, let f, encoded as a RAM next-instruction function Π, be the computed function. Notation-wise, for value v, let $[v]$ denote a bitwise secret-sharing of v between the two

The doInstruction subroutine

Inputs: The server has array $\tilde{D}$, and the server and the client have secret shares of an ORAM state (denoted $[\mathsf{state}_{\mathsf{oram}}]$) and a RAM instruction (denoted $[I]$).

1. The server sets $d = 0^\ell$ and secret shares this value with the client; we denote the shared value by $[d]$.
2. Do:
   (a) The parties securely compute $([\hat{I}], [\mathsf{state}'_{\mathsf{oram}}]) \leftarrow \mathsf{OE}([\mathsf{state}_{\mathsf{oram}}], [I], [d])$, and set $[\mathsf{state}_{\mathsf{oram}}] = [\mathsf{state}'_{\mathsf{oram}}]$.
   (b) The parties perform a secure computation to check if $[\hat{I}] = (\mathsf{done}, z)$. If so, set $[d] = [z]$ and break.
   (c) The client sends its share of $[\hat{I}]$ to the server, who reconstructs $[\hat{I}]$. Then:
      i. If $\hat{I} = (\mathsf{write}, v, d')$ then the server sets $\tilde{D}[v] = d'$ and sets $d = d'$.
      ii. If $\hat{I} = (\mathsf{read}, v, \perp)$ then the server sets $d = \tilde{D}[v]$.
   (d) The server secret shares d with the client.

Output: Each player outputs his share of $\mathsf{state}_{\mathsf{oram}}$ and his share of $[d]$.

Figure 3: Subroutine for executing one RAM instruction.

parties. Our secure ORAM protocol proceeds as follows:

1. The parties run a secure computation of OI (Figure 1). This initializes the ORAM structure, and securely populates it with Server's data D.
2. The parties securely generate and evaluate the RAM program (Figure 2). That is, the following is repeated until the RAM protocol terminates:
   (a) The server and the client use MPC to evaluate Π and obtain shares of the next instruction I.
   (b) I is then compiled, through repeated secure computations of OE, into a sequence of sub-queries, $\hat{I}_1, \ldots \hat{I}_\ell$, where $\ell = O(\log n)$.
   (c) After the server executes each of the sub-queries, instruction I is complete. (In case of a read instruction, the resulting data item is secret shared between the server and the client.)

Again, we stress that, although we do use generic MPC, it is independent of the size of D, and depends only on the RAM representation of f (i.e., Π), and on the complexity of the OE function. Of course, using generic MPC creates significant overhead. In Section 4 we present several tailored MPC protocols for computing the ORAM steps, which greatly improve the performance of our approach.

3.1 Proof of Security

We now prove that the construction presented in the previous section is a secure MPC protocol according to Definition 2.

At the very high level, security against the client holds because he only manipulates the data protected by secret sharing and MPC; the server additionally sees plaintext ORAM instructions – but they do not reveal anything by the ORAM guarantee. (ORAM security [9] is proven in the non-adaptive setting only. However, as we will show, our security simulation goes through, since the adaptive input and function selection by $\mathcal{Z}$ does not depend on protocol message view, and hence the simulators can query the ORAM functions *after* $\mathcal{Z}$ had completed the adaptive selection.)

We start with the descriptions of the Client simulator S_{cl}, who interacts with Z. In i-th computation, S_{cl} receives x_i and $y_i = f_(x_i, D^{(i-1)})$, stores them, and postpones it simulation until he receives the special end symbol from $\mathcal{Z}$.

At this point, S_{cl} outputs entire simulation, as follows:

Pre-processing. S_{cl} simulates pre-processing by generating an appropriate number of random ORAM state shares:

1. S_{cl} runs the $\mathsf{OI}(1^\kappa)$ functionality to obtain an initial state for the ORAM, and generates a uniformly random share $[\mathsf{state}_{\mathsf{oram}}]_c$ for the client.
2. Let $I_1, \ldots, I_{|D^{(0)}|}$ be instructions of the form $(\mathsf{write}, v, \bar{0})$ for $1 \leq v \leq |D^{(0)}|$. S_{cl} sequentially applies OE to $I_1, \ldots, I_{|D^{(0)}|}$, along with the current ORAM state. After each instruction is submitted, the OE functionality returns an updated state, and the simulator generates a uniformly random share $[\mathsf{state}_{\mathsf{oram}}]'_c$ of the updated state for the client.

Computation. For each RAM f to be evaluated, S_{cl} will simulate its execution evaluating the same number of instructions of the form $(\mathsf{read}, 0, \perp)$ using OE. Denote by $|f|$ the execution length of RAM f. Then, for each functionality f:

1. S_{cl} starts with a previously generated share $[\mathsf{state}_{\mathsf{oram}}]_c$ of the ORAM state that was generated during the pre-processing, or during the last computation.
2. Let $I_1, \ldots, I_{|f|}$ be instructions of the form $(\mathsf{read}, 0, \perp)$. As in the pre-processing phase, S_{cl} sequentially runs OE on $I_1, \ldots, I_{|f|}$, along with the current ORAM state. After each instruction is evaluated, OE returns an updated state, and S_{cl} generates a new uniformly random state share $[\mathsf{state}_{\mathsf{oram}}]'_c$.
3. The output reconstruction is simulated by opening to y_i the secret sharing of the output.

The server simulator S_{serv} proceeds similarly to S_{cl}. The notable difference is that the generated view additionally contains the instructions issued by OE. Specifically, during pre-processing, OE is used to evaluate instructions of the form $I_j = (\mathsf{write}, v, \bar{0})$ for $1 \leq v \leq |D^{(i-1)}|$. For each such instruction, OE generates a sequence of subqueries $\hat{\mathbf{I}}_j$, which are included in the generated view. Similarly, during the computation of each functionality f, each instruction is converted by OE into a sequence of subqueries. These subqueries are included in the generated view (in addition to the state shares).

It is not hard to see that these simulators produce views indistinguishable from real execution. The reduction to the (non-adaptive) security of ORAM is straightforward, given our prior observation that the simulators produce their output only after the entire sequence of x_i is specified by $\mathcal{Z}$ (and hence the adaptively chosen sequence of x_i can be fed non-adaptively into the ORAM security experiment).

This leads to the following.

Theorem 2. *Let $\mathcal{ORAM}$ be access-pattern hiding, as defined by Definition 1, and let the underlying employed MPC be secure according to standard definitions. Then, our generic construction (π_f) described above is a secure protocol (according to Definition 2) for computing f, in the presence of honest-but-curious adversaries. Furthermore, if f can be computed in time t and space s by a RAM machine, then π_f runs in amortized time $t \cdot \mathsf{polylog}(s)$, the client uses space $\log(s)$, and the server uses space $s \cdot \mathsf{polylog}(s)$.*

4 An Optimized Protocol

In Section 3 we showed that any Oblivious RAM protocol can be combined with any secure two-party computation scheme to obtain a secure computation scheme with sublinear amortized complexity. This generic solution may be appropriate in many situations, however, current instantiations of the ORAM primitive require us to evaluate complex functions, such as pseudorandom function (PRF), using a secure two-party computation protocol. For example, the ORAM construction of Goldreich and Ostrovsky [9] requires many encryptions, decryptions and executions of a PRF. In spite of the fact that recent advances in the latter provide very efficient solutions for secure joint evaluation of PRFs [8], such secure evaluation is orders of magnitude slower than simply evaluating a PRF locally.

In this section we present a far more efficient secure computation system with sublinear amortized input access. Specifically, we construct, and prove secure, a new secure computation scheme

that borrows ideas from ORAM protocols (specifically [9]) and MPC protocols, in order to provide extreme efficiency. Our resulting protocol uses only a handful of garbled circuits that contain nothing more than a few multiplications, if statements, and XOR operations, and in particular does not require evaluation of PRFs inside MPC. All other computation is done locally by the parties.

4.1 Technical Overview

Our starting point is the construction of Goldreich and Ostrovsky [9], which we use to store the server data. We begin with an overview of their ORAM protocol.

An overview of the Goldreich-Ostrovsky (GO) construction. In the GO construction, every pair (v_i, d_i), where d_i is a data item stored at index v_i in the original RAM protocol, is encrypted under a private key held by the client. (Recall that their protocol – and the ORAM model – does not offer privacy from the client.) Each of the N pairs are stored together in a data structure that has the following properties:

- It consists of $L = \log N$ levels for data of size N, though it will grow to size $\log t$ if there are t read and write operations. Level i contains 2^i "buckets", each of which can hold up to m data elements, where $m = \max(i, \log \kappa)$ and κ is the security parameter. The extra allocation in each bucket will be filled with "dummy" items, which we will explain below. All items stored in these buckets are encrypted with a key that is held by the client.
- Each level $i > 0$ has a hash function associated with it, h^i, which is chosen by the client. If an element (v, d) is stored at level i, it will be stored in the bucket having index $h^i(v)$.

The execution of a read operation and the execution of a write operation have identical structure, in order to prevent the server from distinguishing one from the other. The client begins by scanning the entire bucket at level $i = 0$, looking for the element of interest. Specifically, he decrypts each item in the bucket, one at a time, comparing it to the target value v. He then scans exactly one bucket at each level $i > 0$: if v was not yet found, the client scans the bucket with index $h^i(v)$, and if v was already found, he simply scans a random bucket. Finally after scanning a bucket at each level, (v, d) is written to the top level (regardless of whether this is a read or a write operation).

As mentioned above, whenever an item is being read or written, it is placed in the top level of the data structure. After m operations, this level will fill up. These items are then moved down a level, and shuffled in with the items below. Similarly, after 2^i read and write operations, the items at level i are moved down and shuffled in with the items at level $i + 1$. It follows that after every 2^i read and write operations, level $i + 1$ becomes half full, and after another 2^i operations, it becomes full and is immediately emptied. Every time level i is moved down to level $i + 1$, a fresh hash function is chosen for level $i + 1$, and the items are re-inserted in that level using the new function. This process of merging two levels is quite complex, and we describe it in the course of describing our own protocol below. We refer the interested reader to [9] for a proof that this is a secure ORAM construction.

Extending the protocol to efficient secure computation: In our setting, we are further restricted in that the content of the RAM and the access pattern must remain unseen by the client as well as the server (with the exception of the output that is revealed at the very end of the computation). We have to overcome several challenges:

1) In the protocol of Goldreich and Ostrovsky [9], while the client is reading or writing item (v, d), he has to compute $h^i(v)$ up to d times. In practice, these hash functions would be implemented

by a pseudo-random function (PRF), since we require h^i to be i-wise independent. In our setting, since the client should not learn the value of v, the naive way of implementing their protocol is to compute the PRF inside a garbled circuit. The resulting protocol would be extremely inefficient. Instead, we introduce a new primitive that we call a *shared-oblivious-PRF* (soPRF). This is a PRF in which the input and secret key are each shared between two parties, and the (single) recipient of the pseudo-random output can be designated at the start of the execution.[3] Our construction builds upon the oblivious PRF described by Freedman et al. [8].

2) While scanning each bucket to look for v, the client has to decrypt every ciphertext to see whether he has found a match. He also has to re-encrypt the value after reading it. We need to use an encryption scheme that can be efficiently computed inside a secure computation. We use $\mathsf{Enc}(m;r) = (F_K(r) \oplus m, r)$, and to ensure that encryption and decryption can be efficiently computed inside a garbled circuit, we have the client compute $F_K(r)$ outside the secure computation. All that has to be done inside the garbled circuit is boolean XOR. However, this requires care, since the value r may reveal something about the access pattern to the client. For example, we perform several oblivious sorts on the data during the shuffle protocol. In doing this, we repeatedly decrypt two items, decide whether to swap them, and then re-encrypt them. Suppose the result of one of these operations is $(\mathsf{Enc}(m_0;r_0), \mathsf{Enc}(m_1;r_1))$ if they are not swapped, and $(\mathsf{Enc}(m_1;r_1), \mathsf{Enc}(m_0;r_0))$ if they are. Since we allow the client to choose the randomness used in re-encryption, he can easily determine whether the values were swapped if (when) he sees these ciphertexts again at a later time! The solution is to make certain that the position of the randomness is independent of the outcome of the swap: we use $(\mathsf{Enc}(m_0;r_0), \mathsf{Enc}(m_1;r_1))$ if they are not swapped, and $(\mathsf{Enc}(m_1;r_0), \mathsf{Enc}(m_0;r_1))$ if they are.

3) In the original protocol, the client randomly reassigns elements to buckets whenever two levels are shuffled together (i.e., by choosing a new hash function, and re-hashing all the values). This is a crucial step for providing privacy, but in our setting we cannot entrust this task to the client, since we must protect their locations from him as well. Actually, this issue is subtly tied to the fact that the client knows the encryption randomness. Because he is given the randomness used for decryption during a lookup, he can easily determine the bucket index of the lookup as well. This is not in itself a problem: recall that the server also learns the bucket index during lookup, even in the original ORAM protocol. However, it requires us to hide bucket assignments from the client during the shuffling. If we did not reveal the bucket index during lookup, we might have hoped to reveal more to the client during the shuffle. Instead, we use a shared-oblivious-PRF during our shuffle protocol in order to hide the bucket assignments from *both* parties.

4.2 Our Construction

Shared-Oblivious PRF: As we mentioned above, our protocol makes use of a new primitive that we call a *shared-oblivious-PRF* (soPRF). Our soPRF construction is based on the oblivious PRF of Freedman et al. [8] (see Figure 6). Our particular construction is a function

$$\mathsf{soPRF} : \mathbb{Z}_p^{O(logN)} \times \mathbb{Z}_p^{O(\log N)} \times \{0,1\}^{\log N} \times \{0,1\}^{\log N} \to \mathbb{G} \times \mathbb{G},$$

where $\mathbb{G}$ is a group of prime order $p = O(2^\kappa)$ for which the DDH assumption is expected to hold. The output are secret shares α and β such that α^β is pseudorandom. For our purposes, this is

[3] So far we have only hinted at why the input to the soPRF needs to be shared, and we have said nothing about why the secret key would need to be shared. We explain this when we give the details of the protocol.

insufficient, because subsequent use of these shares in a Yao garbled circuit will be quite impractical; even a single exponentiation is more costly than computing AES. We therefore provide a second, efficient protocol that takes shares of this form, and outputs new *multiplicative* shares: $\alpha' \cdot \beta' = \alpha^{\beta}$. This appears in Figure 7. When we use the soPRF in our protocol, we leave the re-sharing protocol implicit. In our scheme, the input to the soPRF are secret shares of a virtual address, and the output are shares of a pseudorandom value, which is used for determining the location of the virtual address.

Encryption and Decryption: In the protocol that follows, all elements, (v, d), are encrypted using semantically secure, symmetric key encryption; the key K for a (standard) PRF is stored by the client and never changes. We will frequently perform secure computations that involve decrypting a ciphertext, performing an operation, and the re-encrypting the resulting value. As described above, before performing any such computation, the server first sends the random values r and r' to the client, where r is the randomness currently being used in the relevant ciphertext, and r' is chosen randomly for the re-encryption. The client computes $F_K(r)$ and $F_K(r')$ locally, and uses both values as input to the secure computation. Then, inside the garbled circuit, both decryption and re-encryption can be achieved with simple XOR operations. The server stores r', and sends it again to the client the next time the same value is needed in a secure computation. (This saves the client from having to store the randomness.) Below, when we describe the protocol, we leave this step implicit.

Our construction follows along the lines of the general protocol we described in Section 3. The two difficulties are to build an efficient implementation of the doInstruction function, and to make the shuffle protocol from [9] secure against the client while maintaining it's efficiency. We focus on those two tasks here, and do not repeat the remainder of the protocol. We find the prose that follows easier to read than would be precise pseudo-code. Although we reference the functionalities that will be implemented using Yao, and we give precise pseudo-code for each of those functionalities in Appendix C, we expect the reader will find those descriptions helpful mainly to verify the simplicity of the garbled circuits that we rely upon. We note that in the proof of security (Section 5), the interested reader will find a more precise listing of the messages sent to and from each party.

Preprocessing: The players insert the server's data into the ORAM using a sequence of write instructions. We describe the process for a write instruction next.

Read/Write: We assume the players each hold a secret share of the instruction being performed, which includes the virtual address, $v \in [N]$, being sought.

1. **The players scan the top layer looking for data item d stored at address v.** They do this by repeatedly performing a secure computation in which they decrypt an element, compare its virtual address to v, and then re-encrypt it before storing it back. However, the computation should not reveal to either player whether or not v was found, nor what the value of d is. Therefore, the output of the secure computation includes a secret sharing of a state variable that indicates whether v was found, along with shares of d in case v is found. Both players remain unaware of the values of these variables. They scan the entire level, even if v is found mid-way through the scan. The secure computation for this step is found in Figure 8.

2. **The players scan exactly one bucket at each level.** The index of the bucket scanned at level i is chosen as follows:

(a) First, the players engage in a secure computation in which the output is:

- a secret sharing of v if v was not yet found, and
- a secret sharing of the string "$\mathsf{dummy} \circ t$" if v was already found.

Here t is a counter stored by the server, and incremented after every read or write operation. The secure computation for this step is described in Figure 9.

(b) They then compute the soPRF on their shares of v (or on their shares of the dummy address). The key for the soPRF will have been created and shared when elements were last inserted into this level. This is described in the shuffle protocol below. The client sends his share of the soPRF output to the server.

(c) The server maps the output to the integers using the universal hash function h^i associated with level i. He then fetches the corresponding bucket from memory. The players scan this bucket searching for (v, d), precisely the way they scanned the top layer. They do this even if v was already found. If (v, d) is found in this bucket, they store the value in their state (again, unaware that they have done so), and instead of re-encrypting (v, d) and storing it back in the same location, they replace it with an encryption of an empty item. We use the same secure computation as above, described in Figure 8.

3. **The players write the element back to the top layer.** They do this by scanning the layer as before, using a sequence of secure computations to decrypt, compare, and re-encrypt. If they come across the previous version of (v, d), they overwrite it with the (possibly) newer value. (This will happen when (v, d) was first found in the top layer.) If they come across an empty spot in the layer, they simply store the newly encrypted value there. Either way, they continue the scan until they have re-encrypted the entire level. The secure computation for this step is described in Figure 10.

Shuffling. As in the work of Goldreich and Ostrovsky [9], we must occasionally merge a level with the one below, shuffle together the items, and reinsert them into the data structure. As in their protocol, we merge level i with level $i + 1$ after 2^i read or write instructions. We let $n = 2^i$ denote the maximum number of elements in level i. Recall that there are also n buckets at level i, each of size m. We note that there are at most n elements in level $i + 1$ at the time of the shuffle; the capacity is $2n$, but as soon as it fills, they are all moved down to level $i + 2$. Although there are at most $2n$ items in these two levels, there is enough space allocated for $3nm$ words. The remaining spaces are filled with encryptions of empty elements, of the form ("empty, empty"), which help hide how many real elements are currently contained in the level. The goal in the shuffle is to exactly fill every bucket, while ensuring that neither player learns anything about how the real items are distributed.

1. **The players choose new keys and setup a buffer.** They each choose a shared key for the soPRF. Each player will store their share of the key until the next time these two levels are merged; the same key will be used while reading and writing elements (as described above). The server also chooses and stores a universal hash function h^{i+1}, which will be used to map the output of the soPRF to a bucket index in $\{1, \ldots, 2n\}$. He stores this along with his share of the soPRF key. Finally, the server creates a buffer big enough to hold $3nm$ data elements. He places all nm elements from level i and all $2nm$ elements from level $i + 1$ in this buffer.

2. **The players assign the real items to buckets.** There are $2n$ (or fewer) items to be put in the $2n$ empty buckets of level $i+1$, and each bucket is of size m. (The remaining $2nm-2n$ empty spaces of the will later be filled with (encrypted) empty items.)

   (a) They begin this process by (obliviously) sorting the elements, giving priority to real items. This is done by jointly implementing an oblivious sort over the virtual addresses. For each comparison of the sort, the players compute a secure computation that recovers the value of the addresses, compares them, chooses whether to swap them, and finally re-encrypts both the address and the data element. The secure computation to be performed is described in Figure 12.

   (b) The players then do the following for each of the first $2n$ elements in the buffer:

   - They perform a secure computation of the functionality GetHashInput (Figure 9), which outputs a secret sharing of v. For empty items (if there are any), the secure computation simply outputs a random string $r \in \{0,1\}^\kappa$.
   - They compute the soPRF on this value.
   - They perform another secure computation in which both players use the shares they received from the soPRF as input, and the server uses as additional input the description of the universal hash function h^{i+1}. Inside the secure computation, the output shares of the soPRF are reconstructed and then mapped to a bucket index using the hash function h^{i+1}. The bucket index is encrypted and output to the server, who stores it with the element (still kept in the buffer). In Figure 11 we describe the hash function of Mansour et al. [22], which is very simple to compute inside a Yao circuit.

3. **The players assign the empty items to buckets.**

   (a) They scan the last $2nm$ elements in the buffer, which are all guaranteed to be empty (since $2n < mn$). The client encrypts a bucket index for each one: the first m items are mapped to bucket 1, the next m to bucket 2, and so on until exactly m of these empty elements have been assigned to each of the $2n$ buckets.

   (b) They perform another oblivious sort (again using a sorting network), this time sorting by bucket index, with priority given to real items. They then scan the items, using a secure computation to decrypt, increase the count for the current bucket and re-encrypt. If the counter has exceeded m elements for the current bucket, then the element's index is replaced with the symbol $\perp$ before re-encryption. The count is kept private (again by use of encryption). We note that the probability of removing a real element here is negligible, since the probability that more than m real items fall into one bucket is negligible. (See Lemma 3.) In case this does occur, we let the output of this secure computation be a special abort symbol, and the players abort the protocol. The necessary secure computation is described by the functionality in Figure 13.

   (c) Finally, they perform one more oblivious sort on the bucket index, treating $\perp$ as the largest index. In the end, the buffer contains m items per bucket, ordered by bucket index. The server simply copies these directly back into level $i+1$ in their current order, ignoring the leftover items labeled with $\perp$. They again use a secure computation for the functionality described in Figure 12.

4.3 Discussion: Bucket Size

As we will see below, the size of each bucket will play an important role in the proof of security. In particular, we claimed above in Step 3b that if we map n items to n buckets, the probability of overflowing a bucket of size m is negligible in the security parameter. Suppose this were not the case: that we instead use a smaller bucket size, and that we simply sample a new hash function if our elements overflow a bucket during insertion. This admits the following attack: consider a server that is trying to distinguish between two different search sequences by the client, $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n)$. Assume further that he knows all elements in search pattern X are found in level i, while all elements in Y are found at (say) level $i+1$. We note that security must hold even in such a situation. With non-negligible probability, while the client is searching for the elements of Y, he will query the same bucket at level i at least $m+1$ times. On the other hand, if his search was for items in list X, this could not happen, because, by our assumption, the hash function assigned to level i was chosen to map no more than m items to any one bucket. Put another way, we can ensure that the hash function for level i never overflows when mapping elements in level i, but we cannot ensure that it doesn't overflow when mapping elements that are *not* in level i. Therefore, we choose bucket sizes that are large enough to guarantee a negligible probability of overflow for any 2^i elements.

This issue affects security of previous ORAM constructions as well, including [26], and [9] when the buckets are small ([9] say that any bucket size will do). Indeed, [26] suffers from this issue (and a related one that stems from its use of cuckoo hashing) in a way that we were not able to repair (unless we increase their $\log^2$ overhead to $\log^3$).[4] However, for the [9] protocol, security *is* maintained for the parameters they suggest as most practical (logarithmic bucket size).

5 Proof Sketch

The security of the protocol that we presented in the previous section is captured by the following theorem (the performance is analyzed in detail in Section 6).

Theorem 3. *Assuming that the Decisional Diffie-Hellman (DDH) assumption holds, the protocol described in Section 4.2 (Π_F) is a secure protocol for computing F in the presence of honest-but-curious adversaries. Furthermore, if F can be computed in time t and space s by a RAM machine, then Π_F runs in amortized time $t \cdot \mathsf{polylog}(s)$, the client uses space $\log(s)$, and the server uses space $s \cdot \mathsf{polylog}(s)$.*

Proof. To prove security of the protocol against honest-but-curious players, we analyze our protocol in the *hybrid model*, in which we replace all secure computations used during the read and write operations with ideal executions of their corresponding functionalities. It follows from a well known result of Canetti [3] that if the resulting protocol is secure in this hybrid world, then the protocol remains secure in the real world as well.

We start with the following lemma which will be an important part of our proof.

Lemma 1. *In an honest-but-curious execution of protocol Π_F (as described in Section 4), for all soPRF keys k and for all inputs v, the probability that the players compute $\mathsf{soPRF}(k, v)$ in Step 2b more than one time is less than $\mathsf{negl}(\kappa)$, where negl is some negligible function.*

[4] As mentioned in Section 1.2, these insecurities were independently discovered by others, and potential fixes as well as alternative schemes have since been suggested.

Proof. We consider two types of inputs: $v \in [N]$, and dummy inputs of the form $\mathsf{dummy} \circ t$. For inputs of the latter form, note that this particular input can only be used in the tth operation, since t is incremented with every operation. The only way $\mathsf{soPRF}(k, \mathsf{dummy} \circ t)$ can be computed more than once, therefore, is if the same key k is assigned to two different levels at the same time. Since the number of levels is bound by some polynomial (in κ), and there are an exponential number of keys, this is negligibly likely to occur. Consider a pair (k, v) where v is of the first form. In this case, the proof follows from three properties of the protocol: a) elements are moved to the top layer once they are found. b) Whenever we have found an element at some level i, we query $\mathsf{dummy} \circ t$ at all levels $j > i$, where t is uniquely chosen in each operation. c) Whenever an item is moved down to a lower level, a new soPRF key is assigned to the lower level. If we assume that keys are chosen without replacement (i.e. that we never choose the same key more than once), then the lemma clearly follows from these three properties. Since the total number of keys chosen is bounded by some polynomial in κ, the probability that the same key is chosen more than once is negligible. ∎

Lemma 2. *For every non-uniform, polynomial time adversary $\mathcal{A}$ corrupting the server in a hybrid-world execution of the secure computation described in Section 4.2, there exists a non-uniform, polynomial time adversary $\mathcal{S}$ corrupting the server in the ideal world execution F, and a negligible function $neg(\cdot)$, such that for all $\kappa \in \mathbb{N}$:*

$$|\Pr[\mathsf{Real}(1^\kappa, \Pi, \mathcal{A}) = 1] - \Pr[\mathsf{Ideal}(1^\kappa, F, \mathcal{S}) = 1]| \leq neg(\kappa)$$

Proof. We begin by describing the simulation of the shuffle protocol. We will then describe the simulation of a single read/write operation. In the end we will put these together to argue that sequences of read/write operations and shuffles remain secure.

In simulating the shuffle protocol, recall that $\mathcal{S}$ is simulating the hybrid world in which the players have access to ideal executions of the functionalities described in Figures 12, 9, 6, 7, 11, and 13. We outline the hybrid world protocol in Figure 4.

We note that in this hybrid world, almost the entire shuffle protocol proceeds through a sequence of ideal function calls; the players rarely interact outside of these ideal executions. With a few exceptions (described below), the simulator only needs to simulate the output of each of these ideal functionalities. Furthermore, the output from these ideal functionalities is always either a ciphertext, or a random secret sharing. The simulation is therefore quite straightforward: the output of each functionality is simulated with a random string. When the output is supposed to be a ciphertext, the indistinguishability of the simulation follows from the security of the PRF used in the encryption scheme. When the output is supposed to be a secret share, the simulation is distributed identically to the output of the hybrid world. As we will see, the messages that are sent from the client (i.e. that are not communicated through an ideal function call) are always either the random string used in some ciphertext, or a complete ciphertext. Both can again be simulated with random strings.

Simulating oblivious sort: Since oblivious sort is performed three different times, we describe the simulation separately. We note that each oblivious sort in the hybrid world proceeds through a sequence of calls to the ideal instance of the swap functionality (Figure 12), each preceded by the exchange of four encryption and decryption strings. (In Figure 4, we have only depicted a single call in order to save space.) As described above, we simulate (r'_1, r'_2, r'_3, r'_4), which are sent

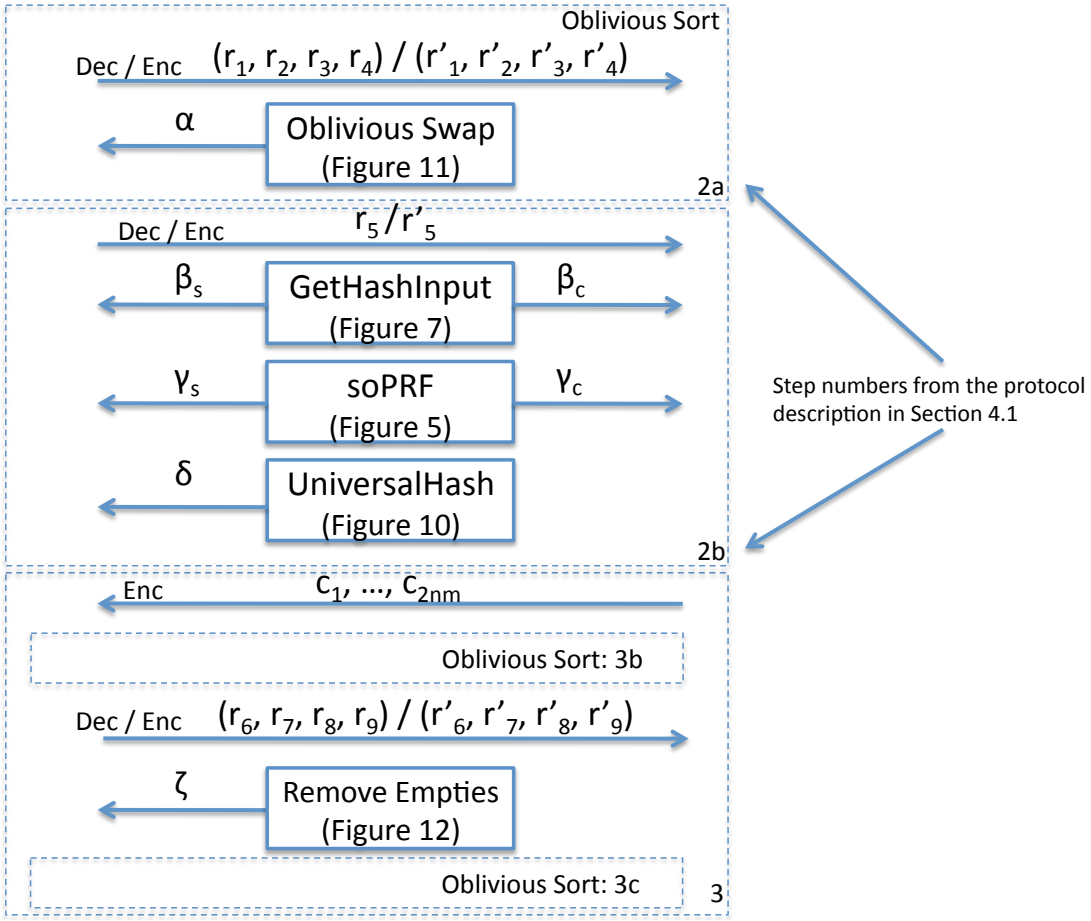

Figure 4: The shuffle protocol in the hybrid world.

from the client before each ideal function call, with random strings. We let α denote the output of the Oblivious Swap functionality; α is a group of four ciphertexts, which are also simulated with random strings. The security of the PRF used in the encryption scheme guarantees that the simulated ciphertexts are indistinguishable from the actual output of the trusted swap functionality.

Step 2a only contains an oblivious sort. In Step 2b, the players make three calls to ideal functionalities (again, engaging in no other communication). To simulate β_s, the output of the ideal functionality of Figure 9, and γ_s, the output of the ideal soPRF, the simulator simply outputs a random string. In the hybrid world, β_s and γ_s are random secret shares, so the simulated output is identically distributed to the hybrid world output. (We note that even if the input to the soPRF is identical in two sequential executions, the *output* is still a fresh pair of secret shares. Therefore, the simulator does not need to do any "book keeping" regarding previous inputs to the soPRF.) The random value r'_5 is used in constructing the ciphertext δ, and can be simulated with a random string. Finally, the last ideal call in this step is to an ideal functionality that takes the output of the soPRF, along with the description of a hash function, and outputs an encryption, δ of the bucket index that results from applying h^i to the output of the PRF (Figure 11). Since the output

is a ciphertext, the simulation simply proceeds as above, outputting a random string.

Step 3a proceeds without any ideal function calls. Here, the client sends exactly $2nm$ ciphertexts to the server, each an encryption of a random bucket index, which is to be assigned to an empty element. The simulator simply sends random strings, which are again indistinguishable from the messages sent by the client in the hybrid world, due to the security of the PRF used in encryption. In Step 3b, the simulator has to simulate the output of the ideal function described in Figure 13. In the hybrid world, when fewer than m real items are mapped to each bucket, the output of this ideal function call is a pair of ciphertexts, which can be simulated as usual. However, the simulator will not attempt to simulate the bad event in which more than m real items are mapped to a particular bucket. Instead, we rely on Lemma 3, which proves that this is negligibly likely to occur, and we allow our simulation to fail with his negligible probability. The final step of the shuffle contains another oblivious sort, which is handled as described above.

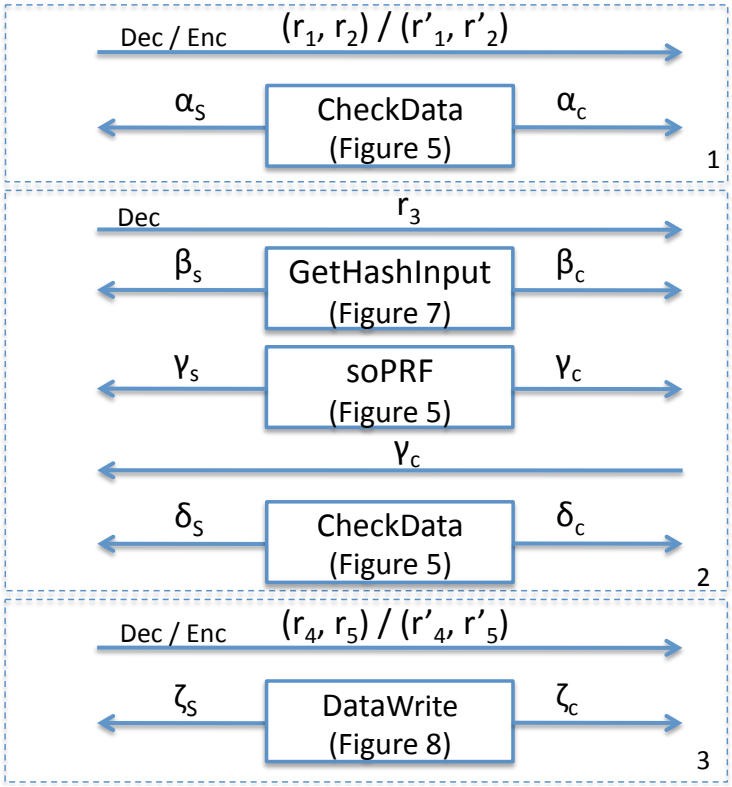

Figure 5: The read and write protocol in the hybrid world.

Simulating read/write: We proceed now to describe the simulation of a single read or write operation. With one important exception, the simulation of the server view during read and write executions is very similar to the simulation used in the shuffle protocol. The only message sent directly from the client to the server is the client's secret share of the soPRF output, sent in Step 2b (message γ_c in Figure 5). The rest of the protocol proceeds through the execution of ideal function calls, and, as before, simulating the output of these functions is straightforward. This is because the outputs of these computations are all either random secret shares, or ciphertexts, both of which can be simulated with random strings. The output from CheckData (messages α_S and δ_S) contain

two secret shares which are uniformly distributed, and one ciphertext. β_S is a secret sharing, and ζ_S contains one secret sharing and one ciphertext.

The most difficult part of the security proof is to simulate the secret share of the soPRF that is sent from the client to the server (γ_c) in Step 2b. In the shuffle protocol, the players keep their output from the soPRF private, using them as input to another secure computation. This simplified the simulation, since it allowed us to simulate just one share of the pseudorandom value, which is distributed uniformly. Here, since the client sends his share in the clear, we have to simulate the reconstructed output of the PRF, and not just a secret sharing of that output.[5]

To simulate the client's share of the soPRF output, we simply send a random group element. We stress that the simulator does *not* try to simulate a random function, because he has no idea what the input to the soPRF is; in particular, if the same input were used twice, he is very unlikely to output the same value both times. To claim that this simulation is indistinguishable from the real execution, therefore, we rely on Lemma 1, where we demonstrated that the players never reuse the same input to the soPRF in between two shuffles. (Recall that when a level is shuffled, the players refresh the key for the soPRF, so at that point it is irrelevant if they reuse an input that was used prior to the shuffle.) If they did reuse the same input, the simulation of the soPRF would be easily distinguished from the real execution.

As mentioned in the discussion of Section 4, another key lemma in the proof demonstrates that the buckets are large enough that they are negligibly likely to overflow during the shuffle. If this were not the case, the simulation of the soPRF *would* be distinguishable from the hybrid-world execution of the soPRF. Note that in the hybrid world protocol, we abort if $m+1$ inputs from the same level collide under the choice of soPRF key (See Figure 13). Therefore, the output of the soPRF in the real execution is pseudorandom, *conditioned on the fact that buckets do not overflow*. In contrast, the simulated output of the soPRF is truly random. Put more formally, for any N inputs, $v_1, \ldots, v_N$, let good_k denote the set of soPRF keys that map no more than m elements in $v_1, \ldots, v_N$ to the same value. Then we require that the following two distributions are indistinguishable:

$$\{\mathsf{soPRF}(k, v_1), \ldots, \mathsf{soPRF}(k, v_N) \mid k \xleftarrow{R} \mathcal{K}\} \stackrel{c}{=} \{\mathsf{soPRF}(k, v_1), \ldots, \mathsf{soPRF}(k, v_N) \mid k \xleftarrow{R} \mathsf{good}_k\}$$

It follows that these distributions are indistinguishable if the probability of overflow is negligible (i.e. if the set good_k contains most of the keyspace). We prove now that the buckets are unlikely to overflow if the soPRF is replaced with a random function. It follows from a hybrid argument that the simulation of message γ_c is indistinguishable from the hybrid-world soPRF.

Lemma 3 ([23])**.** *Let Y be a Poisson random variable with parameter (and mean) μ. If $y > \mu$ then* $\Pr[Y \geq y] \leq \frac{e^{-\mu}(e\mu)^y}{y^y}$.

Lemma 4 ([23])**.** *Let $X_i^{(m)}$, $1 \leq i \leq n$, be a random variable representing the number of balls in the ith bin when m balls are randomly thrown into n bins. Let $Y_1^{(m)}, \ldots, Y_n^{(m)}$ be Poisson random variables with mean m/n. Then, any event that takes place with probability p in the Poisson case, takes place with probability at most $pe\sqrt{m}$ in the exact case.*

[5] We note that we *could* change the protocol to match the shuffling protocol, having the players use their output from the soPRF in a secure computation of the functionality depicted in Figure 11. However, it is much more efficient to send the value in the clear. The reason we don't do the same thing when shuffling is that the protocol becomes insecure if the server sees the reconstructed output of the soPRF both during shuffling *and* during lookup.

Corollary 1. *Suppose that n balls are thrown into n bins. Then, the probability that in the end there is a bin that contains more than $z = \max(\log(n), \log(\kappa))$ balls is at most $\frac{n^{1/2} \max^{\frac{1}{\ln(2)}}(n,\kappa)}{z^z}$.*

Proof. Let $z = \max(\log(n), \log(\kappa))$. When n balls are thrown into n bins, the expected number of balls in each bin is 1. Now consider Poisson variables $Y_1, \ldots, Y_n$ with parameter $\mu = 1$. From Lemmas 3 and 4 we obtain that for all $1 \leq i \leq n$:

$$\Pr[Y_i \geq z] \leq \frac{e^{z-1}}{z^z} \implies \Pr[X_i \geq z] \leq \frac{e^z n^{1/2}}{z^z}$$

$$\frac{n^{1/2} \max^{\frac{1}{\ln(2)}}(n,\kappa)}{z^z}$$

■

We now put together the pieces and consider the full simulation of the server. Since we are in a semi-honest setting, the simulator can begin by submitting the server's input to the trusted party. The output of the honest player is always correct, and is distributed identically to the hybrid world execution. (We leave the correctness of the hybrid world protocol for the reader to verify.) Since the server receives no output, it remains only to argue that the complete view of the server in the ideal world is indistinguishable from that in the hybrid world. We have already argued above that the simulation of a *single* shuffle, or a single read write execution, is indistinguishable from the hybrid-world execution of the same protocol. We now need to argue that a sequence of such simulations remains indistinguishable. Again, the only subtlety arises with the soPRF output; all other messages are clearly independent of one another, even across multiple shuffles, reads and writes. Given Corollary 1 and Lemma 1, it follows that the output of the soPRF in the hybrid world is indistinguishable from a random sequence of group elements, each chosen independently. This is precisely how we have simulated the soPRF, so this concludes the proof of Lemma 2.

□

Lemma 5. *For every non-uniform, polynomial time adversary $\mathcal{A}$ corrupting the client in a hybrid-world execution of the secure computation described in Section 4.2, there exists a non-uniform, polynomial time adversary $\mathcal{S}$ corrupting the client in the ideal world execution F, and a negligible function $neg(\cdot)$, such that for all $\kappa \in \mathbb{N}$:*

$$|\Pr[\mathsf{Real}(1^\kappa, \Pi, \mathcal{A}) = 1] - \Pr[\mathsf{Ideal}(1^\kappa, F, \mathcal{S}) = 1]| \leq neg(\kappa)$$

Proof. Simulating encryption and decryption: In many of the ideal function calls, the server provides input $F_K(r) \oplus v$, while the client provides $F_K(r)$ for decryption, and $F_K(r')$ for re-encryption (see the discussion in Section 4.2). In every such case, the server sends r, r' to the client before they call the ideal function. We need to describe how to simulate these random strings sent by the server. The simulator has to do some book-keeping to be sure the appropriate random strings are sent. (Recall, the client saw the random string needed for decryption at some prior time when it was used for encryption, so the randomness sent by the simulator must remain consistent with those values.) Specifically, the simulator keeps an array of size N to store random strings. He keeps track of the randomness currently being used to encrypt each item at each location in the ORAM structure. We stress that this book-keeping succeeds only because our functionalities all maintain

the ordering of the randomness provided by the client. For example, in Figure 12, note that regardless of whether the items are swapped, the ordering of the randomness provided by the client remains fixed. If this were not the case, not only would the simulation fail (because it could not know what order to place the random strings in), but the client would easily learn something about the outcome of the oblivious sort by observing the final ordering of the randomness he provided.

In the hybrid world protocol, the only messages the client receives from the server are the random strings that we have just finished discussing (see Figure 4). Everything else is done through the execution of ideal functionalities. In the shuffle protocol, the simulation of these functionalities is actually a bit simpler than it was in the case of the server, because the client does not receive any output from the swap functionality used in oblivious sort (Figure 12), from the functionality that computes an encryption of the bucket index (Figure 11), or from the functionality that skims empty items from over-full buckets (Figure 13). The outputs that the client receives from the remaining ideal functionalities are random secret shares of various values: β_c from GetHashInput, and γ_c from the soPRF. The simulator simply outputs random strings to simulate each of these ideal function calls. In the read/write protocol, the same is true: the reader can verify that client output from all functionalities are secert-shares of state variables (i.e. messages α_c, β_c, γ_c, δ_c and ζ_c). All can be easily simulated with random secret shares.

To put these pieces together, when the simulator receives x_i from the environment, he immediately submits them to the trusted functionality and stores the output. He simulates the view of the client through the appropriate number of read/write executions and shuffles, until the RAM protocol terminates.[6] He then sends a second secret share of the output to the client, allowing him to reconstruct the correct output value, sends the output value to the environment, and waits for the next input to arrive from the environment. We note that the server has no output, so we do not need to worry about the joint distribution over the client's view and the server's output. When we consider the simulation of a sequence of read and write executions, we have to describe how the simulator chooses which random values to send to the client (i.e. which buckets should be read). We claim that choosing a random bucket at each level suffices. This follows from two facts: a) the client's view during the shuffle protocol reveals no information about how items have been mapped to buckets, and b) as proven in Lemma 1, with overwhelming probability, the same input is never used twice in the soPRF. ■

This completes the proof of Theorem 3. ■

6 Performance

We evaluate the performance of our protocol and compare it with the state-of-the-art solutions. First, recall our notation: N is the number of records in the database (each of length m), k is the security parameter for underlying DDH groups ($k \approx 256$). While we use big-O notation in our analysis, we stress that we do not hide any large constants; in fact we keep the constants of the higher-order terms.

Each ORAM read/write has complexity $O(4m \log^2 N + m \log N + 3 \log^3 N)$. Additionally, amortized per read/write shuffling has complexity $O((\log \log N)^3 + k^{\log_2 3} \log \log N)$. The amortized shuffling terms are of low order and may be dropped in further analysis, possibly except for the

[6] As mentioned in Section 2.1, we assume that the runtime of the RAM protocol is independent of the inputs.

term $k^{\log_2 3} \log\log N$, (here $k^{\log_2 3} \approx k^{1.6}$ is the size of boolean circuit implementing Karatsuba multiplication [17]).

While each step of ORAM is relatively costly, our solution is orders of magnitude faster than existing solutions (which are all linear in input size) for important functions such as binary search on large DB (or its derivative, location-based service provision). To illustrate the costs relationship for today's medium-size DB of 10^7 records, each of size 10^5, our solution performs $\approx 5 \cdot 10^9$ basic operations (comparable to Yao-gate evaluations), vs standard solution's cost of $\approx 10^{12}$ of same operations. In general, our approach will likely be advantageous when server's input is large, and ORAM program length is short (it is logarithmic for search).

Most importantly, as DB sizes grow with time, the performance advantage of our approach will increase (as much as exponentially, for binary search).

References

[1] M. Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In *Proceedings of the 42nd ACM symposium on Theory of computing*, pages 181–190. ACM, 2010.

[2] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. In *CRYPTO00*, pages 56–74, 2000.

[3] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[4] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious ram without random oracles. In *TCC*, 2011.

[5] J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. J. Strauss, and R. N. Wright. Secure multiparty computation of approximations. In *ICALP*, pages 927–938, 2001.

[6] J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. J. Strauss, and R. N. Wright. Secure multiparty computation of approximations. In *ACM Transactions on Algorithms*, 2006.

[7] Matthew K. Franklin, Mark Gondree, and Payman Mohassel. Multi-party indirect indexing and applications. In *ASIACRYPT*, pages 283–297, 2007.

[8] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography Conference (TCC 05*, pages 303–324, 2005.

[9] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):473, 1996.

[10] Oded Goldreich. *Foundations of Cryptography. Volume I: Basic Tools.* Cambridge University Press, Cambridge, England, 2001.

[11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP (2)*, pages 576–587, 2011.

[12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. *CoRR*, abs/1105.4125, 2011.

[13] Piotr Indyk and David P. Woodruff. Polylogarithmic private approximations and efficient matching. In *TCC*, pages 245–264, 2006.

[14] Yuval Ishai and Eyal Kushilevitz. Perfect constant-round secure computation via perfect randomizing polynomials. In *In Proc. 29th ICALP*, pages 244–256, 2002.

[15] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC*, pages 262–271, 2004.

[16] Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 575–594. Springer, 2007.

[17] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the SSSR Academy of Sciences*, 145:293–294, 1962.

[18] Joe Kilian, André Madeira, Martin J. Strauss, and Xuan Zheng. Fast private norm estimation and heavy hitters. In *TCC*, pages 176–193, 2008.

[19] Louis Kruger, Somesh Jha, Eu-Jin Goh, and Dan Boneh. Secure function evaluation with ordered binary decision diagrams. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 410–420, 2006.

[20] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. Cryptology ePrint Archive, Report 2011/327, 2011. `http://eprint.iacr.org/`.

[21] André Madeira and S. Muthukrishnan. Functionally private approximations of negligibly-biased estimators. In *FSTTCS*, pages 323–334, 2009.

[22] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. *Theor. Comput. Sci.*, 107:121–133, January 1993.

[23] Michael Mitzenmacher and Eli Upfal. *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[24] Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. In *STOC*, pages 590–599, 2001.

[25] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *J. ACM*, 51:231–262, March 2004.

[26] B. Pinkas and T. Reinman. Oblivious RAM Revisited. *Advances in Cryptology–CRYPTO 2010*, pages 502–519, 2010.

[27] Benny Pinkas. Personal Communication, 2011.

[28] Elaine Shi, Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram $O((logN)^3)$ with worst-case cost. Cryptology ePrint Archive, Report 2011/407, 2011. `http://eprint.iacr.org/`.

[29] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. *CoRR*, abs/1106.3652, 2011.

[30] Peter Williams and Radu Sion. Usable pir. In *NDSS*, 2008.

[31] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 08: 15th Conference on Computer and Communications Security*, pages 139–148. ACM Press, October 2008.

[32] David P. Woodruff. Near-optimal private approximation protocols via a black box transformation. In *STOC*, pages 735–744, 2011.

A Shared Oblivious PRF

Definition 3. *Let F be some pseudorandom function (PRF) with key-space $\mathcal{K}$ and input domain X. Let $[k]$ denote a 2-out-of-2 secret of $k \in \mathcal{K}$, and let $[x]$ be the same for input $x \in X$. We say that the function* soPRF *is a* shared oblivious pseudorandom function (soPRF) built on F *if for any $k \in \mathcal{K}$, and any $x \in X$, $\mathsf{soPRF}([x],[k])$ outputs a secret sharing of $F(k;x)$.*

We note that we can trivially build an soPRF from any PRF by using Yao's protocol. However, the goal is to give more efficient construction. Our construction of an soPRF is built on the Naor-Reingold PRF [25]. We review this PRF here. Let $\mathbb{G}$ be some prime order group for which the DDH assumption is expected to hold, and let g be a generator for $\mathbb{G}$. The Naor-Reingold PRF has input domain $X = \{0,1\}^\kappa$, key-space $\mathcal{K} = \mathbb{G}^\kappa$, and output space $\mathbb{G}$. The function is defined as $F(k;x) = g^{r_0 \Pi_i x_i r_i}$, where $r_i \in \mathbb{G}$ makeup the key, and $x_i \in \{0,1\}$ the input.

Theorem 4. *Assuming DDH is hard in $\mathbb{G}$, and secure OT exists, the protocol described in Figure 6 computes an soPRF, and is secure against semi-honest, polynomial time adversaries.*

Proof. Let $\mathcal{I} = \{i_1, \ldots, i_{|\mathcal{I}|}\}$ denote the set of indices such that $\mathsf{v}_c[i_j] \oplus \mathsf{v}_s[i_j] = 1$. We prove our theorem in the standard way, comparing a real execution of π (in the OT-hybrid world) with an ideal function that outputs $f_{\overline{r}_c,\overline{r}_s}(v_c, v_s) = g^{(r_c^0 \cdot r_s^0)\cdot \Pi_{i\in\mathcal{I}}(r_c^i \cdot r_s^i)}$. We define our ideal world functionality as the following randomized functionality:

Ideal functionality F computing an soPRF:
Server input: $(r_s^0, r_s^1, \ldots, r_s^m, \mathsf{v}_s)$.
Client input: $(r_c^0, r_c^1, \ldots, r_c^m, \mathsf{v}_c)$
The functionality F chooses $b_1, \ldots, b_m$ uniformly and independently at random from $\mathbb{Z}_p^*$.
Server output: $\alpha = r_s^0(\prod_{i=1}^m b_i)(\prod_{i\in\mathcal{I}} r_c^i r_s^i)$
Client output: $\beta = g^{\frac{r_c^0}{\Pi_{i=1}^m b_i}}$ and $\Pi_{i=1}^m b_i$.

Note that in the ideal functionality, we include Πb_i in the output of the server. Although this is not obviously necessary, it turns out that it is important for the simulation; in the real world protocol, the server chooses the b_i values himself, so leaking this information is not "harmful".

Shared Oblivious PRF

Let g be a generator of a group G of prime order p for which the DDH assumption holds. Let $\overline{r}_s, \overline{r}_c$ be the shares of the PRF key.

Inputs: Server: $\mathsf{v}_s \in \{0,1\}^m, \overline{r}_s = (r_s^0, r_s^1, \ldots, r_s^m)$, where each $r_s^i \in \mathbb{Z}_p^*$
Client: $\mathsf{v}_c \in \{0,1\}^m, \overline{r}_c = (r_c^0, r_c^1, \ldots, r_c^m)$, where each $r_c^i \in \mathbb{Z}_p^*$

Protocol:

1. The server samples m values $a_1, \ldots, a_m$ in Z_p^* uniformly at random.
2. The client samples m values $b_1, \ldots, b_m$ in Z_p^* uniformly at random.
3. For each $1 \leq i \leq m$:
   (a) The client and the server run an oblivious transfer protocol, with the server as sender and the client as receiver, using the following inputs:
   - If $\mathsf{v}_s[i] = 0$, the server's input is $(a_i, a_i \cdot r_s^i)$. Otherwise the server's input is $(a_i \cdot r_s^i, a_i)$.
   - The client's input is $\mathsf{v}_c[i]$.

   Let x_i be the output value that the client receives from the OT execution.
   (b) The client and the server run an oblivious transfer protocol, with the client as sender and the server as receiver, using the following inputs:
   - If $\mathsf{v}_c[i] = 0$, the client's input is $(b_i \cdot x_i,\ b_i \cdot x_i \cdot r_c^i)$. Otherwise the client's input is $(b_i \cdot x_i \cdot r_c^i,\ b_i \cdot x_i)$.
   - The server's input is $\mathsf{v}_s[i]$.

   Let y_i be the output value that the server receives from the OT execution.
4. Let $\mathcal{I} \subseteq [m]$ denote the set of indices such that $\mathsf{v}_c[i_j] \oplus \mathsf{v}_s[i_j] = 1$.
   The server computes

$$\alpha = r_s^0 \left(\prod_{i=1}^{m} \frac{y_i}{a_i} \right) = r_s^0 \left(\prod_{i=1}^{m} b_i \right) \left(\prod_{i \in \mathcal{I}} r_c^i r_s^i \right).$$

   The client computes $\beta = g^{\frac{r_c^0}{\prod_{i=1}^m b_i}}$. These are secret shares of the pseudorandom value

$$\beta^\alpha = \left(g^{\frac{r_c^0}{\prod_{i=1}^m b_i}} \right)^{r_s^0 (\prod_{i=1}^m b_i)(\prod_{i \in \mathcal{I}} r_c^i r_s^i)} = g^{r_c^0 r_s^0 \prod_{i \in \mathcal{I}} r_c^i r_s^i}.$$

Outputs: The server outputs α, and the client outputs β.

Figure 6: A construction of a Shared Oblivious PRF

However, we note that this has some implications about the pseudo-randomness of the output if a player is given both shares. We will return to discuss this further below.

We begin our security proof by assuming that an adversary $\mathcal{A}$ controls the client in the protocol. We construct a simulator $\mathcal{S}$ which interacts with the adversary and simulates the execution of the protocol in the semi-honest setting.

Lemma 6. *Let F denote an ideal execution of the soPRF as described above, and let π^{OT} denote*

an execution of the protocol in Figure 6 in the OT-hybrid world. For any semi-honest, polynomial time adversary $\mathcal{A}$ with auxiliary input $z \in \{0,1\}^*$ that corrupts the server in the OT-hybrid world, there exists a semi-honest, polynomial time adversary $\mathcal{S}$ with auxiliary input $z \in \{0,1\}^*$ corrupting the server in the ideal world such that

$$\text{IDEAL}^{(i)}_{F,\mathcal{S}(z)}(\kappa, (r_s^0, r_s^1, \ldots, r_s^m, \mathsf{v}_s)) \stackrel{c}{=} \text{REAL}^{(i)}_{\pi^{\text{OT}},\mathcal{A}(z)}(\kappa, (r_s^0, r_s^1, \ldots, r_s^m, \mathsf{v}_s)).$$

Proof. The simulator $\mathcal{S}$ acts as follows:

1. $\mathcal{S}$ submits the server's input to the ideal functionality and receives output α.
2. $\mathcal{S}$ receives $\mathcal{A}$'s input for the m ideal executions of OT in Step 3a. From these inputs, he computes and stores the value $\Pi_{i=1}^m a_i$.
3. For $i \in \{1, \ldots, m-1\}$, $\mathcal{S}$ simulates the output of the ith execution of the ideal OT in Step 3b by choosing $c_i \in \mathbb{Z}_p^*$ uniformly at random, and sending it to the server. He then simulates the mth ideal OT in Step 3b by computing and sending

$$c_m = \frac{\alpha \cdot \Pi_{i=1}^m a_i}{r_s^0 \cdot \Pi_{i=1}^{m-1} c_i}$$

The only messages the server receives in the OT-hybrid world are the outputs of the m OTs in Step 3b. Therefore, the view of $\mathcal{A}$ in the hybrid world is $\{(y_1, \ldots, y_m), \alpha\}$, where y_i is the output received in the ith execution of OT. In the ideal world, these messages are replaced by $(c_1, \ldots, c_m)$, so the view of the adversary is instead $\{(c_1, \ldots, c_m), \alpha\}$. We must argue that these distributions are indistinguishable, when taken jointly with the output of the honest client:

$$\{(y_1, \ldots, y_m), \alpha, \beta, \Pi_{i=1}^m b_i\} \stackrel{c}{=} \{(c_1, \ldots, c_m), \alpha, \beta, \Pi_{i=1}^m b_i\}$$

where the first distribution is over the random coins of the two parties in the hybrid world, and the second distribution is over the coins of the ideal party and of $\mathcal{S}$ in the ideal world. The distributions on α, β and $\Pi_{i=1}^m b_i$ are clearly identical in both worlds, so we are really concerned only with the distributions on $(y_1, \ldots, y_m)$ and on $(c_1, \ldots, c_m)$ given α, β and $\Pi_{i=1}^m b_i$. Consider first the distributions on $(y_1, \ldots, y_{m-1})$ and $(c_1, \ldots, c_{m-1})$. Since the value of $\Pi_{i=1}^m b_i$ does not restrict the value of any $m-1$ size subset of the b_i values, and, by extension, neither does the value of α or β, it follows that both $(y_1, \ldots, y_{m-1})$ and $(c_1, \ldots, c_{m-1})$ are uniformly distributed over $\{\mathbb{Z}_p^*\}^{m-1}$ given α, β and $\Pi_{i=1}^m b_i$. In the hybrid world, recall that α is computed by the server as:

$$\alpha = \frac{r_s^0 \cdot \Pi_{i=1}^m y_i}{\Pi_{i=1}^m a_i}.$$

Therefore, given $(y_1, \ldots, y_{m-1})$ and α, the value of y_m is fully determined by

$$y_m = \frac{\alpha \cdot \Pi_{i=1}^m a_i}{r_s^0 \cdot \Pi_{i=1}^{m-1} y_i}.$$

Since the simulator chooses c_m in precisely this manner, we conclude that the distributions are identical.

□

Lemma 7. *Let F denote an ideal execution of the soPRF as described above, and let π^{OT} denote an execution of the protocol in Figure 6 in the OT-hybrid world. For any semi-honest, polynomial time adversary $\mathcal{A}$ with auxiliary input $z \in \{0,1\}^*$ that corrupts the client in the OT-hybrid world, there exists a semi-honest, polynomial time adversary $\mathcal{S}$ with auxiliary input $z \in \{0,1\}^*$ corrupting the client in the ideal world such that*

$$\text{IDEAL}^{(i)}_{F,\mathcal{S}(z)}(\kappa, (r_c^0, r_c^1, \ldots, r_c^m, \mathsf{v}_c)) \stackrel{c}{=} \text{REAL}^{(i)}_{\pi^{\mathsf{OT}},\mathcal{A}(z)}(\kappa, (r_c^0, r_c^1, \ldots, r_c^m, \mathsf{v}_c)).$$

Proof. The only messages received by the client are the outputs from the first m executions of OT in Step 3a, $(x_1, \ldots, x_m)$. The simulator simulates these m outputs with random, independently chosen group elements from $\mathbb{Z}_p^*$: $(d_1, \ldots, d_m)$. As before, we need to prove that the distributions

$$\{(y_1, \ldots, y_m), \alpha, \beta, \Pi_{i=1}^m b_i\} \stackrel{c}{=} \{(d_1, \ldots, d_m), \alpha, \beta, \Pi_{i=1}^m b_i\}$$

Here the proof is immediate, since the values of a_i are never known to the distinguisher. It follows that the x_i are each independent, random values in $\mathbb{Z}_p^*$, even when the output of each party is given. Therefore the simulated distribution, $(d_1, \ldots, d_m)$, and the hybrid world distribution are identically distributed.

□

We now argue that our construction has an additional property: given the output $\gamma = f_{\overline{r}_c, \overline{r}_s}(v_c, v_s)$, and the server's share of the secret key, $\overline{r}_s = (r_s^0, r_s^1, \ldots, r_c^m)$, we can generate shares α and β from the appropriate (random) distribution such that $\beta^\alpha = \gamma$. The implication is that the client can safely send β to the server, who holds α, while still ensuring that β^α looks random. We note that this property does not have to hold, because our definition of an soPRF allows α and β to contain information about the shares of the secret key. To make this issue more explicit, consider an soPRF that includes player i's share of the secret key (entirely) within i's output share. The output could still be pseudorandom given one of the two shares, but it is certainly not pseudorandom when one player holds both shares. We remark that we did not require this property in our definition because an soPRF may be useful even without it. For example, in our shuffle protocol, neither player ever obtains both output shares. Instead, the shares are used directly as input to a secure computation, which yields encrypted output.

We argue that our protocol remains secure if $\beta = g^{r_c^0 / \prod_{i=1}^m b_i}$ is sent to the player holding $\alpha = r_s^0 (\prod_{i=1}^m b_i)(\prod_{i \in \mathcal{I}} r_c^i r_s^i)$, but we note that this is *not* true if the share α is sent to the player holding $(\beta, \Pi_{i=1}^m b_i)$. To see that it is secure when β is sent to the player holding α, note that given a pseudo-random value γ and the secret key $(r_s^0, \ldots, r_s^i)$, we can easily simulate shares α and β such that $\beta^\alpha = \gamma$, even without knowing $r_c^0, \ldots, r_c^m$. The simulator simply chooses a random α, and then computes $\beta = \gamma^{-\alpha}$. Since Πb_i is not yet fixed, the simulated shares are always consistent with the correct values of r_c^i for *some* choice of Πb_i. However, note that this simulation fails when Πb_i is already known and fixed. More specifically, without knowing $r_s^0, \ldots, r_s^m$, given $(\gamma, \Pi b_i, r_c^0, \ldots r_c^m)$, β is already well defined, and finding α that is consistent with β and γ requires solving an instance of the discrete log problem. Due to this discussion, in our protocol we will always assign β to the client, since he will occasionally send his share of the soPRF output to the server. We leave it as an open problem to find an efficient soPRF that allows either party to send their secret share to the other.

B A Secret Re-Sharing Scheme

We describe a secret re-sharing protocol between two parties C and S. The protocol implements (under the DDH assumption) an ideal functionality which, given group elements α, β that are the inputs of S, C respectively, outputs uniformly distributed group elements u_S, u_C such that $u_c u_s = \alpha^\beta$.

Recall that we use this protocol to reshare an soPRF value that was shared through exponentiation, obtaining multiplicative shares instead. However, we note that the protocol works for any α, β, and does not rely on the pseudorandomness of α^β.

The protocol. The protocol starts with S choosing random α_1, α_2 such that their product is α, and C choosing random β_1, β_2 such that their sum is β. S sends α_1 to C, who can now compute his output, $\alpha_1^{\beta_1}$. The rest of the protocol is designed to allow S to compute his output, $\alpha_2^\beta \alpha_1^{\beta_2}$, without revealing any extra information. This is done by using blinding (raising to a random power, or multiplying by a random number), and by El-Gamal encryption (and its multiplicative homomorphic properties). The details are described in Figure 7.

Multiplicative Re-Sharing

Let G be a group of prime order p for which the DDH assumption holds.
Inputs: Server (S): α; Client (C): β.
Outputs: Server: u_s; Client: u_c; where u_s, u_c are uniformly distributed subject to $u_c u_s = \alpha^\beta$.
Protocol:

1. The server chooses random $\alpha_1, \alpha_2 \leftarrow G$ such that $\alpha_1 \cdot \alpha_2 = \alpha$. The server also chooses random $r \leftarrow \mathbb{Z}_p$. The server sends α_1, α_2^r to the client.
2. The client chooses random $\beta_1, \beta_2, x \leftarrow \mathcal{Z}_p$ such that $\beta_1 + \beta_2 = \beta$. The client also chooses random $g \leftarrow G$ and sets $h = g^x$. We let $[\![W]\!]$ denote an El Gamal encryption of W with respect to the public key (g, h). The client sends g, h, $[\![\alpha_2^{r\beta}]\!]$, and $[\![\alpha_1^{\beta_2}]\!]$ to the server.
3. The server chooses random $r_2 \leftarrow \mathcal{Z}_p$. Using the homomorphic properties of El Gamal encryption, the server computes $[\![\alpha_2^\beta \alpha_1^{\beta_2} g^{r_2}]\!]$ and sends it to the client.
4. The client decrypts the ciphertext sent in the last round, and sends the result $\alpha_2^\beta \alpha_1^{\beta_2} g^{r_2}$ back to the server. The client outputs $u_c = \alpha_1^{\beta_1}$.
5. The server divides by g^{r_2} and outputs $u_s = \alpha_2^\beta \alpha_1^{\beta_2}$.

Figure 7: A protocol for converting exponentiation-based shares to multiplicative shares

Analysis. Correctness follows straightforwardly: $u_c u_s = \alpha_1^{\beta_1} \alpha_2^\beta \alpha_1^{\beta_2} = \alpha_1^{\beta_1 + \beta 2} \alpha_2^\beta = \alpha_1^\beta \alpha_2^\beta = \alpha^\beta$.

Security for one side is information theoretic, and for the other side is based on DDH assumption. A simulator that gets only the input and output of one of the parties, can simulate the entire view by using random values for the incoming messages, subject to the correct output being computed. Specifically, a simulator getting only the input β and output u_c of the client can simulate the entire view by choosing a random β_1, β_2 that sum up to β, then choosing $\alpha_1 = u_c^{\beta_1^{-1}}$ when simulating the first message. All the other values on incoming messages are chosen randomly (and all values sent out are computed honestly). It is easy to see that this simulated view is statistically close to the real view (as all messages sent to the client in the protocol are blinded by a random number, and since the output u_c is random). A simulator getting only the input α and output u_s of the server

can simulate the entire view by using random values for the incoming message in step 2 (while computing step 1 and 3 honestly), and then sending $u_s g^{r_2}$ in step 4 (for the corresponding values g, r_2 from steps $2, 3$). Using the semantic security of the El Gamal encryption (based on the DDH assumption), it is easy to see that this view is indistinguishable from the view in the real protocol.

□

C Supporting Subprotocols

Notation: In the protocols presented in this section we use the following variables:

- v_C and v_S are shares from the virtual address $\mathsf{v}_C \oplus \mathsf{v}_S$ being sought;
- vir_C and vir_S are shares of $\mathsf{vir}_C \oplus \mathsf{vir}_S$, which is either the real or the dummy address searched in some level;
- done_C and done_S are shares of $\mathsf{done}_C \oplus \mathsf{done}_S$, which indicates whether the virtual address has already been found;
- d_C and d_S are shares of $d_C \oplus d_S$, which stores the retrieved data when the the virtual address has been found;
- $F(r)$ denotes PRF value used for encryption.
- (c_V, c_D) are the ciphertexts (encryptions of the virtual address and the data) stored in a physical position in the ORAM structure;

CheckData

Here we check whether a ciphertext (c_1, c_2) matches the input value $v = \mathsf{v}_C \oplus \mathsf{v}_S$. If it does, then we share the corresponding data between the client and the server as $d_C \oplus d_S$. If the virtual address was already found, i.e. $\mathsf{done}_C \oplus \mathsf{done}_S = 1$, we ignore the check. If the virtual address is just matched, we appropriately set the check bit $\mathsf{done}'_C \oplus \mathsf{done}'_S = 1$, re-encrypt the ciphertext as (c'_1, c'_2) to be stored at the server.

Inputs: Client: $\mathsf{v}_C, \mathsf{rw}_C, d_C$, done_C, $F_K(r_1)$, $F_K(r_2)$, $F_K(r_3)$, $F_K(r_4)$
Server: $\mathsf{v}_S, \mathsf{rw}_S, d_S$, done_S, (c_1, c_2)

Protocol:

1. Decrypt the ciphertext (c_1, c_2) to recover the values $\mathsf{v} = c_1 \oplus F_K(r_1)$ and $d = c_2 \oplus F_K(r_2)$.
2. Check whether the data value needs to be retrieved:
   - If $\mathsf{done}_C \oplus \mathsf{done}_S = 1$, compute two shares done'_S and done'_C of 1, and two new shares d'_S and d'_C of the data $d_S \oplus d_C$.
   - Else if $\mathsf{done}_C \oplus \mathsf{done}_S = 0$ and $\mathsf{v} = \mathsf{v}_C \oplus v_S$, compute two shares done'_S and done'_C of 1. If $\mathsf{rw}_C \oplus \mathsf{rw}_S = read$, compute two shares d'_S and d'_C of the retrieved data d.
   - Else compute shares done'_S and done'_C of 0, and another set of shares of 0: d'_S and d'_C.
3. Compute encryptions to be written back:
   - If $\mathsf{done}_C \oplus \mathsf{done}_S = 0$ and $\mathsf{v} = \mathsf{v}_C \oplus \mathsf{v}_S$, set $c' = (\text{"}dummy\text{"} \oplus F_K(r_3), \text{"}dummy\text{"} \oplus F_K(r_4))$.
   - Else $c' = (\mathsf{v} \oplus F_K(r_3), d \oplus F_K(r_4))$.

Outputs: Client: done'_C, d'_C
Server: $\mathsf{done}'_S, d'_S, c'$

Figure 8: A functionality that enables the players to obliviously check whether a data item matches the target.

GetHashInput

We compute the virtual address that will next be looked-up in a level: the real virtual address, if there was no match so far, or a dummy address depending on the counter t, if the item was already found.

Inputs: Client: $\mathsf{done}_C, \mathsf{vir}_C, t$
Server: $\mathsf{done}_S, \mathsf{vir}_S$

Protocol:

1. If $\mathsf{done}_C \oplus \mathsf{done}_S = 0$, create a random secret sharing $\mathsf{v}_S \oplus \mathsf{v}_C = \mathsf{vir}_C \oplus \mathsf{vir}_S$.
2. If $\mathsf{done}_C \oplus \mathsf{done}_S \neq 0$, create a random secret sharing $\mathsf{v}_S \oplus \mathsf{v}_C = (\text{"}dummy\text{"} \circ t)$.

Outputs: Client: v_C
Server: v_S

Figure 9: A functionality that determines whether a real or a dummy look-up should be performed

DataWrite

We check whether the matched item stored in $\mathsf{vir}_C \oplus \mathsf{vir}_S$ and $d_C \oplus d_S$ should be used to over-write the current position in the top level, which currently stores ciphertext (c_1, c_2). We overwrite if these are empty encryptions, or an older encryption of the same virtual address.

Inputs: Client: $\mathsf{done}_C, \mathsf{vir}_C, d_C, \mathsf{done}_C$, $F_K(r_1)$, $F_K(r_2)$, $F_K(r_3)$, $F_K(r_4)$
Server: $\mathsf{done}_S, \mathsf{vir}_S, d_S, \mathsf{done}_S$, (c_1, c_2)

Protocol:

1. Decrypt the ciphertext (c_1, c_2) to recover the values $v = c_1 \oplus F_K(r_1)$ and $d = c_2 \oplus F_K(r_2)$.
2. Compute $v' = \mathsf{vir}_C \oplus \mathsf{vir}_S$ and $d' = d_C \oplus d_S$.
3. Check whether this is the right place to write an encryption of (v', d'):
   - If $\mathsf{done}_C \oplus \mathsf{done}_S = 1$, compute two random shares done'_S and done'_C of 1, and set $(c'_1, c'_2) = (v \oplus F_K(r_3), d \oplus F_K(r_4))$.
   - Else if $v = \mathsf{vir}_C \oplus \mathsf{vir}_S$ or $v = 0$, compute two random shares done'_S and done'_C of 1, and set $(c'_1, c'_2) = (v' \oplus F_K(r_3), d' \oplus F_K(r_4))$.
   - Else, compute two random shares done'_S and done'_C of 0, and set $(c'_1, c'_2) = (v \oplus F_K(r_3), d \oplus F_K(r_4))$.

Outputs: Client: done_C
Server: $\mathsf{done}'_S, (c'_1, c'_2)$

Figure 10: A functionality for determining whether a value should be written to a given position in the top level.

Distributed Universal Hash Function

We compute a universal hash of a value shared between the client and the server as $u_C \cdot u_S$, and send the encrypted output c to the server. Let G be a prime order group. Let a and b be parameters defining the universal hash function from [22].

Inputs: Client: $u_C \in G$, $F(r) \in \{0,1\}^m$
Server: $u_S \in G, a \in \{0,1\}^{n+m-1}, b \in \{0,1\}^m$

Protocol:

1. Set $u = u_c u_S$, and "cast" u as an integer.
2. For $1 \leq i \leq m$ compute the i-th bit of the hash as $y_i = (\oplus_{j=1}^{n}(u_j \; AND \; a_{i+j-1})) \oplus b_i$.
3. Let $y = y_1 || y_2 || \ldots || y_m$ and $c = y \oplus F(r)$.

Outputs: Client: no output
Server: c

Figure 11: A functionality for the distributed computation of a universal hash function.

Oblivious Swap

We re-order two encrypted values (v_1, d_1) and (v_2, d_2) by their virtual address by decrypting, comparing (according to some criteria, described in Section 4, swapping if necessary, and re-encrypting.

Inputs: Client: $F_K(r_1), F_K(r_2), F_K(r_3), F_K(r_4), F_K(r'_1), F_K(r'_2), F_K(r'_3), F_K(r'_4)$
Server: $(v_1 \oplus F_K(r_1), d_1 \oplus F_K(r_2)), (v_2 \oplus F_K(r_3), d_2 \oplus F_K(r_4))$

Computation:

1. Decrypt the input ciphertexts to recover the values $v_1 = (v_1 \oplus F_K(r_1)) \oplus F_K(r_1)$ and $v_2 = (v_2 \oplus F_K(r_3)) \oplus F_K(r_3)$.
2. Compare the values v_1 and v_2 :
   - If $v_1 \leq v_2$: set $b = 0$, $(c'_1, c'_2) = (v_1 \oplus F_K(r'_1), d_1 \oplus F_K(r'_2))$, and $(c'_3, c'_4) = (v_2 \oplus F_K(r'_3), d_2 \oplus F_K(r'_4))$.
   - If $v_1 > v_2$: set $b = 1$, $(c'_1, c'_2) = (v_2 \oplus F_K(r'_1), d_2 \oplus F_K(r'_2))$, and $(c'_3, c'_4) = (v_1 \oplus F_K(r'_3), d_1 \oplus F_K(r'_4))$.

Outputs: Client: no output
Server: (c'_1, c'_2) and (c'_3, c'_4)

Figure 12: A Functionality that enables the players to obliviously compare and swap two elements. This is used repeatedly for an oblivious sort.

Remove Excess Empties

Let count be an array of n counter variables ranging from 1 to m. Let index be a bucket index from 1 to n. Let real be a boolean flag indicating whether index is associated with a real item or an empty item.

Inputs: Server: $\mathsf{index} \oplus F_K(r_1), \mathsf{count} \oplus F_K(r_2), \mathsf{real}$
Client: $F_K(r_1), F_K(r_2), F_K(r_3), F_K(r_4)$

Computation:

1. Recover the values of count and index by computing $(\mathsf{index} \oplus F_K(r_1)) \oplus F_K(r_1)$ and $(\mathsf{count} \oplus F_K(r_2)) \oplus F_K(r_2)$.
2. If $(\mathsf{count}[\mathsf{index}] < m)$
   - $\mathsf{count}[\mathsf{index}]$ ++;
   - let $(c_1, c_2) = (\mathsf{index} \oplus F_K(r_3), \mathsf{count} \oplus F_K(r_4))$
3. Else if $(\mathsf{count}[\mathsf{index}] == m$ and $\mathsf{real} == \mathsf{false})$
   - let $(c_1, c_2) = (\perp \oplus F_K(r_3), \mathsf{count} \oplus F_K(r_4))$
4. Else if $(\mathsf{count}[\mathsf{index}] == m$ and $\mathsf{real} == \mathsf{true})$
   - let $(c_1, c_2) = (\mathsf{abort!}, \mathsf{abort!})$

Outputs: The output (c_1, c_2) is sent to the server.

Figure 13: A functionality for counting m items in each bucket and removing excess empty items.