Distributed Oblivious RAM for Secure Two-Party Computation *

Steve Lu† Rafail Ostrovsky‡

Abstract

Secure two-party computation protocol allows two players, Alice with secret input x and Bob with secret input y, to jointly execute an arbitrary program $\pi(x, y)$ such that only the output of the program is revealed to one or both players. In the two player setting, under computational assumptions most approaches require to first "unroll" π into a circuit, and then execute this circuit in a private manner. Without unrolling the circuit, an alternative method was proposed by Ostrovsky and Shoup (in STOC 1997) [28] with both players simulating the CPU of an oblivious RAM machine using "off-the-shelf" secure two-party computation to perform CPU simulation with atomic instructions implemented by circuits and relying on one of the players to implement encrypted memory. The simulation of the CPU must be done through circuit-based secure two-party computation, thus CPU "memory" in the Oblivious RAM simulation must be minimized, as otherwise it impacts simulation of each step of the computation. Luckily, there are multiple Oblivious RAM solutions that require $O(1)$ CPU memory in the security parameter. The Ostrovsky-Shoup compiler suffers from two drawbacks:

- The best running time of Oblivious RAM simulation with $O(1)$ memory is still prohibitive and requires $O(\log^2 t/ \log\log t)$ overhead for running programs of length t by Kushilevitz, Lu, and Ostrovsky (in SODA 2012) [23].
- The most problematic part of this approach is that all Oblivious RAM simulations, starting with Goldreich and Ostrovsky (in JACM 1996) [14] require "Oblivious Sorting" that introduce a huge constant into Oblivious RAM simulation that essentially kills all practicality.

In this paper, we observe that in the secure two-party computation, Alice and Bob can simulate two non-communicating databases. We show how to extend the Ostrovsky-Shoup compiler so that two non-communicating servers can eliminate all the drawbacks stated above. More specifically, we design a new Oblivious RAM protocol where a client interacts with two non-communicating servers, that supports n reads and writes, and requires $O(n)$ memory for the servers, $O(1)$ memory for the client, and $O(\log n)$ amortized read/write overhead for data access. The constants in the big-O notation are tiny, and we show that the storage and data access overhead of our solution concretely compares favorably to the state-of-the-art single-server schemes. As alluded to above, our protocol enjoys an important feature from a practical perspective as well. At the heart of almost all previous single-server Oblivious RAM solutions, a crucial but inefficient process known as oblivious sorting was required. In our two-server model, we describe a novel technique to bypass oblivious sorting, and show how this can be carefully blended with existing techniques to attain a more practical Oblivious RAM protocol in comparison to all prior work.

This new protocol leads to a novel application in the realm of secure two-party RAM program computation. We show that our multi-server Oblivious RAM construction can be composed with an extended version of the Ostrovsky-Shoup compiler to obtain a new method for secure two-party RAM computation with lower overhead than (implicitly) existing constructions by a factor of $O(\log n/ \log\log n)$ of Gordon et al. [18].

Keywords: Oblivious RAM, Cloud Computing, Multi-Server Model, Software Protection, Secure Computation.

*This work expands on and is protected by patent application [25].

†Stealth Software Technologies, Inc. E-mail: steve@stealthsoftwareinc.com

‡Department of Computer Science and Department of Mathematics, UCLA. Work done with consulting for Stealth Software Technologies, Inc. E-mail: rafail@cs.ucla.edu

1 Introduction

The concept of outsourcing data storage or computation is widespread in practice. This raises the issue of what happens to the privacy of the data when the outsourcing service is only semi-trusted or untrusted. Encryption can be employed to protect the *content* of the data, but it is apparent that information might be revealed based on how the data is accessed. Simply put, encryption by itself alone does not entirely address the issue of data privacy at hand.

The sequence of reads and writes a client makes to the remotely stored data is known as the *access pattern*. Even if the content of the data is protected by encryption, the server storing the data can deduce information about the encrypted data just by observing and analyzing the access pattern. For instance, the server can correlate this pattern with public information about the client's behavior, such as the purchase or sale of stock. Over time, the server may learn enough information to predict the behavior of the client or the underlying semantics of the data, thereby defeating the purpose of encrypting it in the first place.

A trivial solution would be for the client to access the entire stored database every single read or write. This clearly hides the access pattern, but the per-access overhead is linear in the size of stored data. The question remains:

Is it possible to hide the access pattern with less than linear overhead?

In the model where the client is a Random Access Machine (i.e. RAM model), Goldreich and Ostrovsky [11, 27, 26, 14] introduced the concept of hiding the access pattern in the context of software protection. A small protected CPU would run on a machine with large unprotected RAM. The goal was to obliviously simulate access to RAM, so that the set of instructions ran by the CPU would be protected against an outsider monitoring the RAM. In this manner, an adversary observing the RAM would learn nothing about what instructions were executed except the total number of instructions. The work of [11] featured two solutions using constant client memory: a "square-root" solution and a "recursive square-root" solution. The amortized time overhead of executing a program in the former scheme was $O(\sqrt{n})$, and $O(2^{\sqrt{\log n \log\log n}})$ in the latter. Ostrovsky [27, 26] then discovered what is known as the "hierarchical solution" which had amortized overhead $O(\min\left\{(\log^3 n); (\log^3 t)\right\})$, where t is running time. The subsequent work of Goldreich and Ostrovsky [14] contains the merged results of [27, 26, 11] and featured a simpler method of reshuffling. The work described a way of simulating oblivious RAM with $O(\log^3 n)$ amortized overhead per access for n data items, using constant client storage[1] and $O(n \log n)$ server storage.

While the asymptotic behavior of $O(\log^3 n)$ overhead might seem efficient, this only gives a practical advantage over the trivial solution when $n > \log^3 n$ (without even considering the constants hidden in the O). A database of size $n = 2^{20}$ results in an overhead factor of roughly 8000, and such a large overhead would seem to cast oblivious RAM as outside the realm of practicality. Making oblivious RAM practical would be of great impact, as it can be applied to software protection and several other important problems such as cloud computing, preventing cache attacks, etc. as we discuss later.

A highly interesting and powerful application of Oblivious RAM is in the problem of efficient, secure two-party *program* computation. While there are many ways to model computation, such as with Turing Machines, (Boolean) Circuits, Branching Programs, or Random Access Machines, one representation might be more natural than another, depending on the program. Nearly all secure two-party computation protocols require the program to be specified as a *circuit* between the two parties. Due to a classic result by Pippenger and Fischer [32], any Turing machine running in time T can be transformed into a circuit with only $O(\log T)$ blowup, but it is not known in the RAM model of computation whether there exists such an efficient transformation to circuits. Therefore, even using the most efficient secure two-party protocols for circuits (e.g. IKOS [19] or IPS [20] protocols), there is no clear path on how to apply these to efficiently perform secure RAM computation. We consider the question:

How can one efficiently perform secure two-party computation in the natural RAM model?

RELATED WORK. Subsequent to Goldreich and Ostrovsky [27, 26, 11, 14], works on Oblivious RAM [33, 34, 31, 16, 17] looked at improving the concrete and asymptotic parameters of oblivious RAM. We give a full summary of

[1] We count storage as the number of records or data items stored in memory. We do not count small variables such as counters or loop iterators toward this amount as these typically are tiny compared to the size of a data item, nor the private-key encryption/decryption cost.

these schemes in Section 2. The major practical bottleneck of all these works on Oblivious RAMs is a primitive called *oblivious sorting* that is being called upon as a sub-protocol. Although the methods for oblivious sorting have improved, it still remains as both the critical step and the primary stumbling block of all these schemes. Even if new methods for oblivious RAM are discovered, there is an *inherent limit* to how much these schemes can be improved. It was shown in the original work of Goldreich and Ostrovsky [14] that there is a lower bound for oblivious RAM in this model.

([14], Theorem 6): *To obliviously perform n queries using only $O(1)$ client memory, there is a lower bound of $O(\log n)$ amortized overhead per access.*

We mention several results that are similar to Oblivious RAM but work in slightly different models. The works of Ajtai [1] and Damgård et al. [10] show how to construct oblivious RAM with information-theoretic security with poly-logarithmic overhead in the restricted model where the Adversary can not read memory contents. That is, these results work in a model where an adversary only sees the sequence of accesses and not the data. The work of Boneh, Mazieres and Popa [7] suggests ways to improve the efficiency of the "square-root" solution [11, 14] when memory contents are divided into larger blocks.

Finally, the notion of *Private Information Storage* introduced by Ostrovsky and Shoup [28] allows for private storage and retrieval of data. The work was primarily concentrated in the information theoretic setting. This model differs from Oblivious RAM in the sense that, while the communication complexity of the scheme is sub-linear, the server performs a *linear* amount of work on the database. The work of Ostrovsky and Shoup [28] gives a multi-server solution to this problem in both the computational and the information-theoretic setting and introduces the Ostrovsky-Shoup compiler of transforming Oblivious RAM into secure two-party computation. Quoting directly from their STOC 1997 [28] paper (with citations cross-referenced):

> *Both databases keep shares of the state of the CPU, and additionally one of the databases also keeps the contents of the Oblivious RAM memory. The main reason why we can allow one of the constituent databases to keep both the "share" of the CPU and the Oblivious RAM memory and still show that the view of this constituent database is computationally indistinguishable for all executions is that the Oblivious RAM memory component is kept in an encrypted (and tamper-resistant) form (see [14]), according to a distributed (between both databases) private-key stored in the CPU. For every step of the CPU computation, both databases execute secure two-party function evaluation of [35, 13] which can be implemented based on any one-way trapdoor permutation family (again communicating through the user) in order to both update their shares and output re-encrypted value stored in a tamper-resistant way in Oblivious RAM memory component.*

The current work can be viewed as a generalization of the [28] model where servers must also perform sublinear work. The notion of single-server "PIR Writing" was subsequently formalized in Boneh, Kushilevitz, Ostrovsky and Skeith [6] where they provide a single-server solution. The case of amortized "PIR Writing" of multiple reads and writes was considered in [8].

Also along the lines of oblivious simulation of execution, the result of Pippenger and Fischer [32] shows that a single-tape Turing machine can be obliviously simulated by a two-tape Turing machine with logarithmic overhead.

With regard to secure computation for RAM programs, the implications of the Ostrovsky-Shoup compiler was explored in the work of Naor and Nissim [24] which shows how to convert RAM programs into so-called circuits with "lookup tables" (LUT). This transformation incurs a poly-logarithmic blowup, or more precisely, for a RAM running in time T using space S, there is a family of LUT circuits of size $T \cdot polylog(S)$ that performs the same computation. The work then describes a specific protocol that securely evaluates circuits with lookup tables. [24] also applies to the related model of securely computing branching programs.

The Ostrovsky-Shoup compiler was further explored in the work of Gordon et al. [18] in the case of amortized programs. Namely, consider a client that holds a small input x, and a server that holds a large database D, and the client wishes to repeatedly perform private queries $f(x, D)$. In this model, an expensive initialization (depending only on D) is first performed. Afterwards, if f can be computed in time T with space S with a RAM machine, then

there is a secure two-party protocol computing f in time $O(T) \cdot polylog(S)$ with the client using $O(\log S)$ space and the server using $O(S \cdot polylog(S))$ space.

OUR RESULTS. In this paper, we introduce a new model for oblivious RAM in which we can achieve far better parameters than existing single-server schemes. We mention that our model, like most existing schemes, focuses on computational rather than information-theoretic security, and we only make the mild assumption that one-way functions exist. Instead of having a single server to store our data, we consider using multiple[2] servers to store our data. These servers are assumed to not communicate or collude with each other, but only communicate with the client. From a theoretical point of view, this model has been used in the past to much success such as in the seminal works in the areas of multi-prover Interactive Proof Systems [4] and multi-server Private Information Retrieval [9]. This model is also directly applicable to the Ostrovsky-Shoup compiler for the construction of secure two-party RAM computation protocols.

In our two-server model, we introduce a new approach for oblivious RAM **that completely bypasses oblivious sorting**, which was the inhibiting factor of practicality in most previous schemes (we give a comparison in Section 2.3). To perform a sequence of n reads or writes, our solution achieves $O(\log n)$ amortized overhead per access, $O(n)$ storage for the servers, and constant client memory. This matches the lower bound in the single-server model, and thus *no* single-server solution that uses constant client memory can asymptotically outperform our solution. We have:

Theorem 1 (Informal)**.** *Assume one-way functions exist. Then there exists an Oblivious RAM simulation in the* two-server *model with $O(\log n)$ overhead.*

As mentioned above, this new Oblivious RAM protocol leads to a novel and interesting application to secure *program* computation. We then show how to perform secure two-party RAM computation by adapting our multi-server Oblivious RAM solution to fit the Ostrovsky-Shoup compiler [28]. This allows us (under certain assumptions) to achieve the most efficient secure RAM computation using only logarithmic overhead as opposed to the *poly-logarithmic* overhead of all prior schemes [28, 24, 18].

Theorem 2 (Informal)**.** *Given any secure circuit computation protocol with constant overhead, there exists a two-party secure RAM computation protocol with $O(\log n)$ overhead.*

1.1 Applications

SECURE TWO-PARTY AND MULTIPARTY COMPUTATION. In the case of MPC, we can apply oblivious RAM by letting the participants jointly simulate the client, and have the contents of the server be stored in a secret-shared manner. This application was originally described by Ostrovsky and Shoup [28] in the case of secure two-party computation. As described above, several other subsequent works [24, 10, 18] also investigated the application of Oblivious RAM to the area of secure computation. This can be beneficial in cases where the program we want to securely compute is more suitable to be modeled by a RAM program than a circuit. In particular, we demonstrate in this paper how our two-server ORAM construction can be applied to the case of secure two-party RAM program computation in the semi-honest model to obtain more efficient protocols for this purpose.

SOFTWARE PROTECTION. Original works of Goldreich [11] and Ostrovsky [27] envisioned protecting software using oblivious RAM. A small tamper-resistant CPU could be incorporated in a system with a large amount of unprotected RAM. A program could be run on this CPU by using oblivious RAM to access the large memory. Because this RAM could be monitored by an adversary, the benefit of oblivious RAM is that it hides the access pattern of the program that is running, thus revealing only the running time of the program to the adversary.

CLOUD COMPUTING. With the growing popularity of storing data remotely in the cloud, we want a way to do so privately when the data is sensitive. As mentioned before, simply encrypting all the data is insufficient, and by

[2] In general, we can consider multiple servers. For our purposes, two servers suffice.

implementing oblivious RAM in the cloud, a client can privately store and access sensitive data on an untrusted server.

PREVENTING SIDE-CHANNEL ATTACKS. There are certain side-channel attacks that are based on measuring the RAM accesses that can be prevented by using oblivious RAM. For example, an adversary can mount a cache attack by observing the memory cache of a CPU. This poses a real threat as it can be used for cryptanalysis and has even been observed in practice in the work of Osvik-Shamir-Tromer [29].

PRIVATE DATA STRUCTURES. Rather than protecting an entire program, we can consider the middle ground of data structures. Data structures typically fit neatly into the RAM model, where each read or write is a sequence of accesses to memory. Performing these operations will leak information about the data, and we can use oblivious RAM to mitigate such issues. For example, commercial databases typically offer encryption to protect the data, but to protect the access pattern we can replace the data structures with oblivious ones.

2 Background

2.1 Model

We work in the RAM model, where there is a tiny machine that can run a program performs a sequence of reads or writes to memory locations stored on a large memory. This machine, which we will refer to as the client, can be viewed as a stateful processor with a special data register v that can run a program Π. From a given state Σ of the client and the most recently read element x, $\Pi(\Sigma, x)$ acts as the next instruction function and outputs a read or write query and an updated state Σ'.

Because we wish to hide the type of access performed by the client, we unify both types of accesses into a operation known as a *query*. A sequence of n queries can be viewed as a list of (memory location, data) pairs $(v_1, x_1), \ldots, (v_n, x_n)$, along with a sequence of operations $op_1, \ldots, op_n$, where op_i is a READ or WRITE operation. In the case of READ operations, the corresponding x value is ignored. The sequence of queries, including both the memory location and the data, performed by a client is known as the *access pattern*.

In our model, we wish to obliviously simulate the RAM machine with a client, which can be viewed as having limited storage, that has access to multiple servers with large storage that do not communicate with one another. However, the servers are untrusted and assumed to only be, in the best case, *semi-honest*, i.e. each server follows the protocol but attempts to learn additional information by reviewing the transcript of execution. For our model, we assume that the servers can do slightly more than just I/O, in that they can do computations locally, such as shuffle arrays, as well as perform hashing and basic arithmetic and comparison operations. We mention that there are some standard ways to deal with *malicious* servers, similar to a single-server case [27, 14]. However, due to the fact that the servers in our model are assigned some computational tasks, guarding against malicious servers requires additional techniques (described in the full version) to ensure security.

An oblivious RAM is *secure* if for any two access patterns in the ideal RAM, the corresponding views in the execution of those access patterns of any individual server are computationally indistinguishable. Another way of putting it is that the view of a server can be simulated in a way that is indistinguishable from the view of the server during a real execution.

We also briefly state the model of secure two-party RAM computation which we work in (see, e.g. [12], for a more in-depth treatment of general models of secure computation). Let $f(A, B)$ be a function that can be efficiently computed by a RAM machine, that is to say, there exists a program Π that a client can execute starting with A and B stored in the appropriate input memory locations and halting with the result $f(A, B)$ in the appropriate output memory location on the server. We usually denote the running time $T(n)$ and the space used $S(n)$ which depend on the size of the input n.

We use an ideal/real simulation-based definition of security and also work in the setting of semi-honest adversaries. There are two parties, Alice and Bob that receive inputs A and B respectively and they wish to compute $f(A, B)$. In the ideal world, there is an ideal functionality $\mathcal{F}_f$ that on inputs A and B simply computes $f(A, B)$

and sends the output to Alice and Bob. In the real world, we can think of the Alice and Bob executing a protocol π_f that computes $f(A, B)$. Roughly speaking, we say that π_f *securely realizes* the functionality $\mathcal{F}_f$ if there exists an efficient simulator $\mathcal{S}$ playing the role of the corrupted party in the ideal world can produce an output that is computationally indistinguishable from the view of the corrupted party in the real world.

2.2 Tools

HASHING. In our scheme and in previous schemes, hashing is a central tool in storing the records. For our purposes, the hash functions used for hashing will be viewed as either a random function or a keyed pseudo-random function family F_k. Recall the standard hashing with buckets data structure: there is a table of m buckets, each of size b, and a hash function $h : \mathcal{V} \to \{1 \dots m\}$. A record (v, x) is stored in bucket $h(v)$.

CUCKOO HASHING. A variant of standard hashing known as Cuckoo Hashing was introduced by Pagh and Rodler [30]. In this variant, the hash table does not have buckets, but now two hash functions h_1, h_2 are used. Each record (v, x) can only reside in one of two locations $h_1(v)$ or $h_2(v)$, and it is always inserted into $h_1(v)$. If there was a previous record stored in that location, the previous record is kicked out and sent to its other location, possibly resulting in a chain of kicks. If the chain grows too long or there is a cycle, new hash functions are chosen, and it was shown that this results in an amortized $O(1)$ insertion time. A version of cuckoo hashing with a stash was introduced by Kirsch et al. [21] where it was shown that the probability of having to reset drops exponentially in the size of the stash.

OBLIVIOUS SORTING. A key ingredient in most previous schemes is the notion of oblivious sorting. This is a sorting algorithm such that the sequence of comparisons it makes is independent of the data. For example, the schemes of Batcher [3] and Ajtai et al. [2] are based on sorting networks, and recently a randomized shell sort was introduced by Goodrich [15].

2.3 Comparison to Prior Work

We briefly overview the relevant key techniques used in previous schemes. We summarize all of these in Table 1

SQUARE ROOT SOLUTION. In the work of Goldreich [11] and subsequently Goldreich-Ostrovsky [14], a "square root" solution (ORAM_{GO1}) was introduced for oblivious RAM. This solution was not hierarchical in nature, and instead had a permutation of the entire memory stored in a single array along with a cache of size $\sqrt{n}$ which was scanned in its entirety during every query. After every $\sqrt{n}$ queries, the entire array was obliviously sorted and a new permutation was chosen. This results in an amortized overhead of $O(\sqrt{n} \log n)$ per access.

HIERARCHICAL SOLUTION. In the work of Ostrovsky [27] and subsequently [14], a hierarchical solution was given for oblivious RAM. In this solution, the server holds a hierarchy of bucketed hash tables, growing geometrically in size. New records would be inserted at the smallest level, and as the levels fill up, they would be reshuffled down and re-hashed by using oblivious sorting. A query for v would scan bucket $h_i(v)$ in the hash table on level i. By using the oblivious sorting of Batcher [3], the scheme achieves an $O(\log^4 n)$ amortized query overhead (ORAM_{GO2}), and with AKS [2], an $O(\log^3 n)$ query overhead is achieved (ORAM_{GO3}).

BUCKET SORTING. In the work of Williams-Sion [33], the client was given $O(\sqrt{n})$ working memory instead of $O(1)$. By doing so, it was possible to achieve a more efficient oblivious sorting algorithm by sorting the data locally in chunks of size $\sqrt{n}$ and then sending it back to the server. This resulted in a solution (ORAM_{WS}) with $O(\log^2 n)$ query overhead. This idea of using the client to sort was continued in the work of Williams et al. [34] in which a Bloom filter [5] was introduced to check whether or not an element was stored in a level before querying upon it. This solution (ORAM_{WSC}) was suggested to have $O(\log n \log \log n)$ overhead, but the a more careful analysis of [31] shows that this depends on the number of hash functions used in the Bloom filter.

CUCKOO HASHING. Recently, Pinkas and Reinman [31] suggested a solution in which cuckoo hashing is used instead of standard bucketed hashing. The oblivious sorting algorithm used the more practical one of [15]. This

Scheme	Comp. Overhead	Client Storage	Server Storage	# of Servers	Dist. Prob.[3]
[14]ORAM_{GO1}	$O(\sqrt{n}\log n)$	$O(1)$	$O(n+\sqrt{n})$	1	$negl$
[14]ORAM_{GO2}	$O(\log^4 n)$	$O(1)$	$O(n\log n)$	1	$negl$
[14]ORAM_{GO3}	$O(\log^3 n)$	$O(1)$	$O(n\log n)$	1	$negl$
[33]ORAM_{WS}	$O(\log^2 n)$	$O(\sqrt{n})$	$O(n\log n)$	1	$negl$
[34]ORAM_{WSC}	$O(\log n \log\log n)$	$O(\sqrt{n})$	$O(n)$	1	$poly$
[31]ORAM_{PR}	$O(\log^2 n)$	$O(1)$	$O(n)$	1	$poly$
[16]ORAM_{GM1}	$O(\log^2 n)$	$O(1)$	$O(n)$	1	$negl$
[16]ORAM_{GM2}	$O(\log n)$	$O(n^\nu)$	$O(n)$	1	$negl$
[17]ORAM_{GMOT}	$O(\log n)$	$O(n^\nu)$	$O(n)$	1	$negl$
[23]ORAM_{KLO}	$O(\log^2 n/\log\log n)$	$O(1)$	$O(n)$	1	$negl$
Our Scheme	$O(\log n)$	$O(1)$	$O(n)$	2	$negl$

Table 1: Comparison of oblivious RAM schemes.

resulted in a scheme (ORAM_{PR}) that only used constant client memory, $O(n)$ server storage, and only $O(\log^2 n)$ query overhead where the constant was empirically shown to be as small as 150. The work of Goodrich and Mitzenmacher [16] also made use of cuckoo hashing, although the stashed variant of cuckoo hashing was used for their scheme (ORAM_{GM1}), which resulted in similar parameters. They also suggested a solution (ORAM_{GM2}) where the client has $O(n^\nu)$ memory, in which case they are able to achieve $O(\log n)$ query overhead. A stateless version of this scheme is featured in [17] with similar asymptotics. The best known overhead for schemes with constant client memory come from the work of Kushilevitz, Lu, and Ostrovsky [23] (full version appears on ePrint [22]), where they introduce a new balancing technique for their scheme (ORAM_{KLO}) to achieve an overhead of $O(\log^2 n/\log\log n)$.

3 Our Scheme

3.1 Overview

Our new scheme uses the hierarchical format of Ostrovsky [27]. The general principle behind protocols using this technique can be stated as: the data is encrypted (under semantically secure encryption) and stored in hierarchical levels that reshuffle and move into larger levels as they fill up. To keep track of the movement, for each level we logically divide different time periods into *epochs*, based on how many queries the client has already performed. All parties involved are aware of a counter t that indicates the number of queries performed by the client.

In hierarchical schemes, the reshuffling process is the main bottleneck in efficiency, specifically the need to perform "oblivious sorting" several times. We identify the purposes that oblivious sorting serves during reshuffling and describe methods on how to replace oblivious sorting in our two-server model.

The first purpose of oblivious sorting is to separate real items from "dummy" items. Dummy items are records stored in the levels to help the client hide the fact that it may have already found what it was looking for prior to reaching that level. For example, if the client was searching for virtual memory location v, and it was found on level 3, the client still needs to "go through the motions" and search on the remaining levels to hide the fact that v had already been found. On all subsequent levels in this example, the client would search for "$dummy$" $\circ\, t$ instead of v.

[3]Due to flaws in the way hash functions are used, the security of these schemes could be only polynomially secure. For further discussion on the security analysis of these schemes, see [16, 23, 18].

The second purpose of oblivious sorting is to identify old records being merged with new records. New records are always inserted at the topmost level, and as the levels are reshuffled down, there is the possibility that an old record will run into a new one on some lower level. Because they both have the same virtual memory location v, a collision will occur. To resolve this, when records are being reshuffled, an oblivious sort is performed to place old records next to new ones so that the old records can be effectively erased (re-encrypted as a dummy record).

Finally, oblivious sorting is used to apply a pseudo-random permutation to the records as they are being reshuffled. A permutation is necessary to prevent the server from being able to track entries as they get reshuffled into lower levels.

The key ingredient to our new techniques is the idea of "tagging" the records and letting the two servers do most of the work for the client. A typical record looks like (v, x) where v is the index of the record (virtual memory location), and x is the data stored at that index. In most previous schemes, a hash function was applied to v to determine where the record would be stored in the data structure. Because the client cannot reveal v to the servers, and yet we wish for the servers to do most of the work, the client needs to apply tags to the records. Later, when the client needs to retrieve index location v, the client first computes the tag and then looks up the tag instead of v in the data structure located on the servers.

Note that this tagging must be performed carefully. We want the client to use only $O(1)$ working memory, so it cannot simply keep a list of all the tags it has generated in the past. Instead, the tags must be deterministic so that the client is able to re-create the tag at a future point in time when needed. However, if the tags depend only on v, a server can immediately identify when two encrypted records have the same index location v.

To resolve the apparent tension between these two requirements, we use a pseudo-random function (PRF) applied to v, the level it is stored on, as well as the period of time which it is stored at that level, known as the *epoch*. We describe this in greater detail in our construction.

To begin, we first present a warm-up construction to demonstrate the utility of tagging and using two servers. For a sequence of client queries of length n, this *insecure* strawman construction will have the servers storing $O(n)$ data, the client having $O(1)$ working memory, and the amortized overhead of queries being $O(\log n)$.

3.2 Warm-up Construction

Recall that in our model, there is a client and two servers that only communicate with the client. The client wants to perform a sequence of n data queries, where the t-th query is of the form (READ, v_t, x_t) or (WRITE, v_t, x_t). The client keeps a counter for t. The client makes use of a semantically secure encryption scheme to encrypt all the data being stored on the servers. Whenever the client reads a record from the server, it is implied that the client needs to decrypt it.

In our construction, we use the hierarchical structure of [14] combined with Cuckoo Hashing (similar to [31, 16]) and tagging to store the data. The two servers, $\mathcal{S}_0$ and $\mathcal{S}_1$, store alternating levels of the hierarchical data structure. We allow the data structure to grow in levels as we insert more records. Typically, if there are n records in the data structure, then there will be $O(\log n)$ levels among the two servers. We let N denote the total number of levels.

The top level, level 1, of the hierarchical data structure is an array that holds a small number, c, of records. While some of these may be "dummy" records, from the point of view of the servers each query will always insert a single record at the top level. Level i consists of a table of size $c \cdot 2^i$. This acts as storage for a cuckoo hash table that stores up to $c \cdot 2^{i-1}$ real records at $\alpha = 0.5$ hash table utilization. We call a level *full* if that $c \cdot 2^{i-1}$ (real or dummy) records are stored at that level, and *half full* if $c \cdot 2^{i-2}$ records are stored at that level. Note that we do not think of c necessarily as a constant, and in our main construction c will be on the order of $\log n$.

After c queries, the top level becomes full and will need to be reshuffled into the second level in some oblivious manner. In general, after every $c \cdot 2^{i-1}$ queries level i is reshuffled to level $i + 1$. In our construction, we make use of an optimization[4] by means of a simple observation. By algebra, we see that whenever a multiple of $c \cdot 2^{i-1}$

[4]This optimization has also been pointed out by [31].

queries have been performed, it will also the case that a multiple of $c \cdot 2^{i-j}$ queries have been performed as well, for all $j = 2 \dots i$. With this observation, we can avoid performing obsolete reshuffling at higher levels. Instead, after the t-th query, we first compute i^*, the largest value such that $c \cdot 2^{i^*-1}$ divides t, and then reshuffle all levels from 1 to i^* into level $i^* + 1$.

Every time a level has records reshuffled into or out of it, we call that a new epoch for the level. Thus, for level i, every $c \cdot 2^{i-2}$ accesses by the client will be the start of a new epoch. We denote $e_i = \lfloor t/(c \cdot 2^{i-2}) \rfloor$ as the epoch for level i.

Each level i will have two hash functions, $h_{i,0}$ and $h_{i,1}$, associated with it for the purpose of cuckoo hashing. As mentioned previously, these can be modeled as PRFs parameterized by the level and the epoch.

For the purpose of tagging, the client will have a PRF F and a secret seed s kept private from the two servers.

All levels on both servers are initialized to be empty. To perform a read v query or write (v, y) query on index location v, the client performs the steps in Figure 1.

1. The client allocates temporary storage m, large enough to hold a single record, initialized to a dummy[a] value "$dummy$".
2. Read each entry of the entire top level one at a time. If v is found as some entry (v, x) then store x in m.
3. For subsequent levels $i = 2 \dots N$, perform the following with the server holding level i:
   (a) If v has not already been found, compute the tag for v at this level as $z = F_s(i, e_i, v)$. Else, set $z = F_s(i, e_i, \text{"}dummy\text{"} \circ t)$.
   (b) Fetch into local memory the records (v_0, x_0) and (v_1, x_1) from locations $h_0(z)$ and $h_1(z)$.
   (c) If v is found at one of these locations, i.e. $v = v_b$ for some $b = 0, 1$, then replace v_b with "$dummy$" $\circ\, t$ and store x_b in m.
   (d) Re-encrypt the fetched records and store them back to their original locations, releasing them from local client memory.
4. In the case of a write query, here we overwrite $m = y$.
5. Read each entry of the entire top level one at a time, and re-encrypt each record with the following exception: If the record is of the form (v, x), then overwrite it with (v, m) before re-encrypting it.
6. If (v, x) was not overwritten at the top level, write (v, m) in the first available empty spot (even if m is "$dummy$"), otherwise write a dummy value ("$dummy$" $\circ\, t$, "$dummy$").
7. The client increments the local query counter t. If t is a multiple of c, then a reshuffle step is performed as described below.

[a] Dummy records have been used in slightly different manners in various previous works. To be clear, in this work we treat dummy values more as padding records to be treated as real records.

Figure 1: Warm-up Construction: Query

1. $\mathcal{S}_a$ allocates a temporary array and inserts every (encrypted) record it holds between levels 1 and i. $\mathcal{S}_a$ applies a random permutation to this temporary array and sends its contents one by one to the client.

2. The client re-encrypts each record and sends it to $\mathcal{S}_b$. In this step, both empty and dummy records are treated as real records.

3. $\mathcal{S}_b$ allocates a temporary array and inserts every record it holds between levels 1 and i as well as the records it received from the client in the previous step. $\mathcal{S}_b$ applies a random permutation to this temporary array and sends its contents one by one to the client.

4. The client re-encrypts each record and sends it to $\mathcal{S}_a$, announcing that it is empty if the record is empty, and tagging non-empty records (v, x) with the output of the PRF $F_s(i+1, e_{i+1}, v)$, where e_{i+1} is the *new* epoch of level $i+1$. Note that v may be a virtual memory address or a dummy location. In this step, dummy records are treated as real records and we are only concerned with eliminating empty records.

5. $\mathcal{S}_a$ now holds $c \cdot 2^{i-1}$ tagged records. It allocates a temporary table of size $c \cdot 2^{i+1}$ and it uses h_0 and h_1 for level $i+1$ and epoch e_{i+1} to hash these records into this temporary table. If the insertion fails, new hash functions are selected. $\mathcal{S}_a$ sends the temporary table one record at a time to the client.

6. The client re-encrypts each record where empty records are encrypted as ("$empty$", "$empty$") so that $\mathcal{S}_b$ does not know it is empty, and sends it to $\mathcal{S}_b$. $\mathcal{S}_b$ then stores these records in level $i+1$ in the order in which they were received.

Figure 2: Warm-up Construction: Reshuffle

After every c queries, the top level becomes full and needs to be reshuffled down. This reshuffle may cause a cascade as previously indicated, and we now describe the reshuffling process that takes place at level i. Recall that we require the reshuffling to be done at the deepest level possible, i.e. i is the largest value such that $c \cdot 2^{i-1} | t$. We first prove a lemma describing the state of the levels prior to the reshuffle.

Lemma 1. *Let t be a multiple of c and let i be the largest integer such that $c \cdot 2^{i-1} | t$. Then, prior to reshuffling, the top level is full, levels $2 \dots i$ are half full, and level $i+1$ is empty. After reshuffling, levels $1 \dots i$ are empty and level $i+1$ is half full.*

Proof. Each query inserts exactly one record at the top level, and thus after every c queries the top level becomes full and is reshuffled down to a lower level. Since t is a multiple of c, the top level is full prior to reshuffling.

Next, we show that level $i+1$ is empty. Write $t = kc \cdot 2^{i-1}$ for some integer k. By the maximality of i, k must be odd. If $k = 1$, then this is the first time we are reshuffling into level $i+1$, which is initialized to be empty. If $k > 1$, then during the previous query $t' = (k-1)c \cdot 2^{i-1}$ it is the case that there is some $i' > i$ such that $c \cdot 2^{i'-1} | t$, and thus level $i+1$ was reshuffled and emptied into a lower level. Since the state of level $i+1$ does not change between queries t' and t, it remains empty before reshuffling.

Indeed, observe that for any level j, the state of level j only changes every $c \cdot 2^{j-2}$ queries. For any positive integer k, on query $kc \cdot 2^{j-2}$, it is the case that lower levels are shuffled into level j when k is odd, and level j is shuffled into a lower level when k is even. Since t is a multiple of $c \cdot 2^{i-1}$, we can write $t = k_j c \cdot 2^{j-2}$ for some even k_j for every level $2 \le j \le i$. Thus just prior to query t, every level $2 \le j \le i$ had records reshuffled into it and is non-empty.

Finally, we show that any level except the top must be either empty or half full. When records are shuffled out of a level, it is left empty, so it only remains to show that any time records are shuffled into a level that it becomes half full (note that due to the alternating nature of shuffling, we never shuffle records into a partially full level). After the t-th query, we have already shown that the top level is full and contains c records, and each subsequent level up to i is non-empty. By induction, we assume that levels $2 \le j \le i$ are exactly half full, and therefore contain $\frac{c \cdot 2^{j-1}}{2}$

records. After reshuffling, all the records are placed in level $i+1$, and there will be $c + \sum_{j=2}^{i} c \cdot 2^{j-2} = c \cdot 2^{i-1}$ records. This is exactly half of $c \cdot 2^i$, which is the capacity of level $i+1$.

□

To reshuffle levels $1 \ldots i$ into level $i+1$, suppose $\mathcal{S}_b$ holds level $i+1$ and let $\mathcal{S}_a = \mathcal{S}_{1-b}$ be the other server. The steps in Figure 2 are performed.

3.3 Analysis of Warm-up Construction

We now turn our attention to analyzing the storage and communication overhead.

Claim 1. *For a sequence of n queries, with a top level buffer of size c and a total of N levels, the construction described in Section 3.2 uses $O(n)$ memory for each server, $O(1)$ working memory for the client, and $O(c+N)$ amortized overhead for queries. If c is a constant, this results in $O(\log n)$ amortized overhead for queries.*

Proof. Aside from the top level, each level i is of size $c \cdot 2^i$ and can store up to $c \cdot 2^{i-1}$ records. Recall that N, the number of levels, is chosen so that $c \cdot 2^N$ is $O(n)$. The total amount of storage used by the two servers combined is $c + \sum_{j=2}^{N} c \cdot 2^j = c \cdot 2^{N+1}$, which is $O(n)$.

As in previous schemes, we don't count the private keys nor the counter t of the client as storage because these are typically small relative to the size of a record. Instead, we measure the working memory of the client as the number of records it needs to simultaneously store and process. In the query portion of the scheme, the client will simultaneously hold at most 2 records: the record matching its query, and the record pulled from the hash table. During reshuffling, the client accesses the records one at a time, so only the size of a single record needs to be allocated in the working memory of the client in this case. All in all, the client uses $O(1)$ working memory.

Finally, during a read or write operation, the client scans the entire top buffer, which contains c elements, and 2 elements from each of the N remaining levels. This is $O(c+N)$, but we also need to incorporate the amortized cost of reshuffling. Every $c \cdot 2^{i-1}$ queries, the contents of level i are caught in the reshuffle step, being passed through the client 3 times. This gives a total amortized overhead of $\sum_{i=1}^{N} \frac{3c \cdot 2^i}{c \cdot 2^{i-1}} = 6N$, which is also $O(c+N)$.

□

3.4 Full Construction

A recent result [23] points out that hash overflows leads to an adversary distinguishing different access patterns. Plain cuckoo hashing and the variant of cuckoo hashing with a constant stash [16] yield a polynomial chance of this event occurring. The work of Goodrich-Mitzenmacher [16] shows that cuckoo hashing with *logarithmic* size stash yields a superpolynomially small chance (in n) of overflow, under the assumption that the size of the table is $\Omega(\log^7 n)$. Thus, as a starting point, we use the level layout of [16], where smaller levels are standard hash tables with buckets and larger levels are cuckoo hash tables with a stash of size $O(\log n)$. Furthermore, we use a version of the "shared stash" idea introduced by [17] and subsequently used in [16, 23]. We emphasize that this is where the similarities end with existing schemes and that significant modifications must be diligently balanced to yield a scheme with our desired parameters. Before we begin describing our full construction, we take a quick glance at the balancing dynamics involved in choosing the right parameters for our scheme. Our goal is to achieve $O(\log n)$ amortized overhead per query, while maintaining that the hash tables do not overflow with all but negligible probability.

Recall that the hybrid construction in [16] uses standard hashing with buckets for lower levels, up until the point where a level contains $\log^7 n$ elements, where it switches to cuckoo hashing with a stash of size $\log n$. For the probability of overflow to be negligible for standard hashing, the buckets must be of size $\log n$. To perform a read query, a bucket is scanned at each of the smaller levels, and the entire stash is scanned along with 2 elements of the cuckoo hash table at the larger levels. This operation already incurs a total of $O(\log n \log\log n)$ reads for the small

levels and $O(\log^2 n)$ for the larger levels. We now summarize the series of modifications that need to be made to the structure of the scheme:

Reduce Bucket Size. The standard hash tables will now use buckets of size $3 \log n / \log \log n$. This causes the total amount of reads for the small levels to drop down to $O(\log n)$. This produces a negative side effect: a bucket will now overflow with $\frac{1}{n^2}$ probability.

Standard Hash with Stash. We introduce a stash of size $\log n$ to the standard hash tables to hold the overflows from the now reduced bucket sizes. We prove in Section 3.5 that the probability of overflowing the stash is negligible. This produces a negative side effect: each stash must be read at the smaller levels, bringing us back to $O(\log n \log \log n)$ reads for the smaller levels.

Cache the Stash.[17, 16, 23] For both the smaller levels and larger levels, the stash of size $\log n$ will not be stored at that level, but the entire stash is instead re-inserted into the hierarchy. In fact, by choosing the top level to be of size $O(\log n)$, we can fit the entire stash into the top level. We show how this step is done during a reshuffle. Now, because there is no longer a stash at any level, the total amount read from all the levels combined will be $O(\log n)$. This will cause the levels to be reshuffled more often, but we show that it is at most by a constant factor.

We now give the full details of our scheme.

Let $c = 2 \log n$, where c is taken to be the size of the top level ($i = 1$) as in the warm-up scheme. We split the top level in half so that each server holds half of the top level, and for subsequent levels, server $\mathcal{S}_{i \mod 2}$ holds level i. Let ℓ_{cuckoo} be the level such that $c \cdot 2^{\ell_{cuckoo}-1}$ is $\Omega(\log^7 n)$, e.g. $7 \log \log n$. For levels $i = 2, \ldots, \ell_{cuckoo} - 1$, level i will be a standard hash table consisting of $c \cdot 2^{i-1}$ buckets, each of size $3 \log n / \log \log n$, along with a "mental"[5] stash of size $\log n$. For levels $i = \ell_{cuckoo}, \ldots, N$, level i will be a cuckoo hash table that can hold up to $c \cdot 2^{i-1}$ elements, which is of size $c \cdot 2^i$, along with a "mental" stash of size $\log n$.

The client keeps a local counter t of how many queries have been performed so far, as well as a counter s to indicate how many dummy stash elements were created. We describe how a query is performed in Figure 3. To reshuffle levels $1 \ldots i$ into level $i + 1$, suppose $\mathcal{S}_b$ holds level $i + 1$ and let $\mathcal{S}_a = \mathcal{S}_{1-b}$ be the other server. The steps in Figure 4 are performed.

3.5 Standard Hash with Stash

Let $m = polylog(n)$, $b = 3 \log n / \log \log n$ and $s = \log(n)$. We show that in the case of hashing m values into a standard hash table of size m with buckets of size b and a stash of size s, the probability of overflow is negligible in n. Recall that whenever a bucket overflows, the element is inserted into the stash and is not considered an overflow. We take a look at a standard balls and bins argument, and we start by calculating the probability that more than b balls end up in any bin when we throw m balls into m bins. The probability that there are exactly i balls in any bin can be bounded as:

$$\Pr[\text{Bin has } i \text{ balls}] = \binom{m}{i} \frac{1}{m^i} \left(1 - \frac{1}{m}\right)^{m-i} \leq \left(\frac{me}{i}\right)^i \frac{1}{m^i}$$

Taking the probability over all $i = b, \ldots, m$ we get $Pr[\text{Bin has at least } b \text{ balls}] \leq \left(\frac{e}{b}\right)^b \frac{1}{1-e/b}$. We denote this event by $\mathcal{E}_{a,i}$, i.e. the event that bin a has more than i balls.

Plugging in $b = 3 \log n / \log \log n$, we get the classic result (if $\log \log n > 3 \log \log \log n$):

[5]There will be no physical stash at this level, but during reshuffles a temporary stash is created for the purpose of hashing which will subsequently be re-inserted back to the top level.

1. The client allocates temporary storage m, large enough to hold a single record, initialized to a dummy value "$dummy$".
2. Read each entry of the entire top level from both servers one at a time. If v is found as some entry (v, x) then store x in m.
3. For small levels $i = 2 \dots \ell_{cuckoo} - 1$, perform the following with the server holding level i:
   (a) If v has not already been found, compute the tag for v at this level as $z = F_s(i, e_i, v)$. Else, set $z = F_s(i, e_i, \text{"}dummy\text{"} \circ t)$.
   (b) Fetch into local memory the bucket corresponding to $h(z)$ one element at a time, i.e. fetch (v_j, x_j) for $j = 1, \dots, 3 \log n / \log \log n$ from bucket $h(z)$ one element at a time.
   (c) If v is found in some record (v_i, x_i), then replace v_i with "$dummy$" $\circ\, t$ and store x_i in m.
   (d) Re-encrypt the fetched records and store them back to their original locations, releasing them from local client memory.
4. For large levels $i = \ell_{cuckoo} \dots N$, perform the following with the server holding level i:
   (a) If v has not already been found, compute the tag for v at this level as $z = F_s(i, e_i, v)$. Else, set $z = F_s(i, e_i, \text{"}dummy\text{"} \circ t)$.
   (b) Fetch into local memory the records (v_0, x_0) and (v_1, x_1) from locations $h_0(z)$ and $h_1(z)$.
   (c) If v is found at one of these locations, i.e. $v = v_b$ for some $b = 0, 1$, then replace v_b with "$dummy$" $\circ\, t$ and store x_b in m.
   (d) Re-encrypt the fetched records and store them back to their original locations, releasing them from local client memory.
5. In the case of a write query, here we overwrite $m = y$.
6. Read each entry of the entire top level one at a time, and re-encrypt each record with the following exception: If the record is of the form (v, x), then overwrite it with (v, m) before re-encrypting it.
7. If (v, x) was not overwritten at the top level, write (v, m) in the first available empty spot (even if m is "$dummy$"), otherwise write a dummy value ("$dummy$" $\circ\, t$, "$dummy$").
8. The client increments the local query counter t. If t is a multiple of $c/2$, then a reshuffle step is performed as described below.

Figure 3: Main Construction: Query

$$
\begin{aligned}
\Pr[\mathcal{E}_{a,b}] &\leq \left(\frac{e}{b}\right)^b \frac{1}{1 - e/b} \\
&\leq \left(\frac{e}{b}\right)^b 2 \\
&\leq 2 \left(\frac{e}{3 \log n / \log \log n}\right)^{3 \log n / \log \log n} \\
&\leq 2 \left(e^{-3 \log n + 3 \frac{\log \log \log n}{\log \log n} \log n}\right) \\
&\leq \frac{2}{n^2}
\end{aligned}
$$

1. $\mathcal{S}_a$ allocates a temporary array and inserts every (encrypted) record it holds between levels 1 and i. $\mathcal{S}_a$ applies a random permutation to this temporary array and sends its contents one by one to the client.
2. The client re-encrypts each record and sends it to $\mathcal{S}_b$. In this step, both empty and dummy records are treated as real records.
3. $\mathcal{S}_b$ allocates a temporary array and inserts every record it holds between levels 1 and i as well as the records it received from the client in the previous step. $\mathcal{S}_b$ applies a random permutation to this temporary array and sends its contents one by one to the client.
4. The client re-encrypts each record and sends it to $\mathcal{S}_a$, announcing that it is empty if the record is empty, and tagging remaining records (v, x) with the output of the PRF $F_s(i+1, e_{i+1}, v)$, where e_{i+1} is the *new* epoch of level $i+1$. Note that v may be a virtual memory address, a dummy value, or a stash dummy value. In this step, dummy records are treated as real records and we are only concerned with eliminating empty records.
5. $\mathcal{S}_a$ now holds $c \cdot 2^{i-1}$ tagged records. It allocates a temporary hash table (standard or cuckoo, depending on the level), with a stash of size $\log n$ and it uses the hash functions corresponding to level $i+1$ and epoch e_{i+1} to hash these records into this temporary table. If the insertion fails, new hash functions are selected (we will show this happens with negligible probability). $\mathcal{S}_a$ then informs the client the number of elements inside the stash, σ, then sends both the temporary table and the stash one record at a time to the client.
6. As the client receives records from $\mathcal{S}_a$ one at a time, it re-encrypts each record and sends them to $\mathcal{S}_b$ without modifying the contents except:
   (a) The first σ empty records in the table the client receives from $\mathcal{S}_a$ are encrypted as $(\text{“}stashdummy\text{”} \circ s, \text{“}empty\text{”})$, incrementing s each time. Note that a table is always more than half empty, and therefore we can always find σ empty slots.
   (b) Subsequent empty records from the table are encrypted as $(\text{“}empty\text{”}, \text{“}empty\text{”})$.
   (c) Every empty record in the stash is re-encrypted as $(\text{“}stashdummy\text{”} \circ s, \text{“}empty\text{”})$, incrementing s each time.
7. $\mathcal{S}_b$ stores the table records in level $i+1$ in the order in which they were received, and stores the stash records at the top level.

Figure 4: Main Construction: Reshuffle

Although the probabilities of overflow of each bin are not independent, we can calculate the probability that at least $c = \log\log\log n$ of them overflow (more than b balls) as if they were independent. This is because the probability that a bin overflows reduces the chance that another bin overflows, and thus treating them independently gives an upper bound. The probability that exactly j bins (independently) overflow can be calculated as:

$$\Pr[j \text{ bins overflow}] \leq \binom{m}{j}\left(\frac{2}{n^2}\right)^j\left(1-\frac{2}{n^2}\right)^{m-j} \leq \left(\frac{me}{j}\right)^j\frac{1}{n^{2j}} \leq \left(\frac{e}{j}\right)^j\frac{1}{n^j}$$

Taking the probability over all $j = c, \ldots, m$ we get $Pr[\text{at least } c \text{ bins overflow}] \leq \frac{1}{n^c}\left(\frac{e}{c}\right)^c\frac{1}{1-e/c}$, which is dominated by $\frac{1}{n^c}$ and hence negligible in n.

Finally, we need to bound the amount of overflow that is caused by each bucket. We argue that with negligible probability will a bucket contain more than $d = \log\log\log n \log n / \log\log n$ elements. We get (if $\log\log n >$

$3(\log\log\log n - \log\log\log\log n)$):

$$\begin{aligned}
\Pr[\mathcal{E}_{a,c}] &\leq \left(\frac{e}{d}\right)^d \frac{1}{1-e/d} \\
&\leq \left(\frac{e}{d}\right)^d 2 \\
&\leq 2\left(\frac{e}{3\log\log\log n \log n/\log\log n}\right)^{3\log\log\log\log n/\log\log n} \\
&\leq 2\left(e^{-3\log\log\log n\log n + 3\frac{\log\log\log n - \log\log\log\log n}{\log\log n}\log\log\log n\log n}\right) \\
&\leq 2\left(e^{-2\log\log\log n\log n}\right) \\
&\leq \frac{2}{n^{2\log\log\log n}}
\end{aligned}$$

Taking the Union Bound, the probability that all bins contain fewer than d elements is $\frac{2m}{n^{2\log\log\log n}}$ which is negligible in n.

Finally, we have to calculate the probability that there are more than s elements in the stash. Observe that if all bins have fewer than d elements, and if at most c bins overflow, then there will be at most $cd = (\log\log\log n)^2 \log n/\log\log n$ elements in the stash, which (asymptotically) is less than s. The probability that each of these conditions fail is negligible, and by the Union Bound, the probability that either of them fail is still negligible.

3.6 Analysis of Main Construction

Theorem 3. *For a sequence of n queries, the main construction uses $O(n)$ memory for each server, $O(1)$ working memory for the client, and $O(\log n)$ amortized overhead for queries.*

Proof. Computing the sizes of the levels, level 1 is of size $c = 2\log n$, split between the servers, levels $i = 2, \ldots, \ell_{cuckoo} - 1$ are of size $c \cdot 2^{i-1} \cdot 3\log n/\log\log n$ each, giving a total of $O(\log^9 n)$ size, since $\ell_{cuckoo} = 7\log\log n$. Levels $i = \ell_{cuckoo}, \ldots, N$ are of size $c \cdot 2^i$ each, where $c \cdot 2^N = n$, hence there is a total of $O(n)$ size. Note that the additional elements added in by the stash dummy elements can be counted as follows: every $c/2$ steps, we insert another $\log n$ stash dummy records into the hierarchy. Therefore, after n steps, at most $2n\log n/c = n$ stash dummy records have been inserted, and we can simply accommodate this by adding one extra level at the bottom.

Clearly, the client uses constant working memory as it only transmits records one at a time.

When the client performs the read operation, it reads $2\log n$ records from the top level, $3\log n/\log\log n$ elements from each level $i = 2, \ldots, \ell_{cuckoo} - 1$, and 2 elements from each level $i = \ell_{cuckoo}, \ldots, N$. Since $\ell_{cuckoo} = 7\log\log n$ and $N = \log n - \log\log n - 1$, this gives a total of roughly $25\log n$ elements read.

Because we re-insert the stash (which is half the size of the top level), we need to reshuffle twice as often. Following the warm-up analysis, recall that each reshuffle only moves an element in the level at most 3 times. We sketch the analysis of the amortized overhead:

- For levels $2, \ldots, \ell_{cuckoo} - 1$, each level contains $c \cdot 2^{i-1} 3\log n/\log\log n$ elements and needs to be reshuffled every $c \cdot 2^{i-1}/2$ steps. This incurs an amortized overhead of:

$$3\sum_{i=2}^{7\log\log n - 1} \frac{c \cdot 2^{i-1} 3\log n/\log\log n}{c \cdot 2^{i-2}} = O(\log n)$$

with a constant of roughly 125.

- For levels $\ell_{cuckoo}, \ldots, N$, each level contains $c \cdot 2^i$ elements and needs to be reshuffled every $c \cdot 2^{i-1}/2$ steps. This incurs an amortized overhead of:

$$3 \sum_{i=7 \log\log n}^{\log n - \log\log n - 1} \frac{c \cdot 2^i}{c \cdot 2^{i-2}} = O(\log n)$$

with a constant of roughly 10.

□

Before we prove the security of our construction, we state a few important lemmas.

Lemma 2. *At all times during the execution of the scheme, any record of the form $(v, *)$ will appear at most once in the hierarchy unless $v =$ "$empty$".*

Proof. An index v must be either a virtual memory location, a dummy element, a stash dummy element, or empty. Virtual memory locations are only introduced into the hierarchy either from a read query that found v at a lower level and moved it to the top, or from a write query that did not find v in the hierarchy. A dummy element "$dummy$" $\circ\, t$ can only be introduced during query t, and it can be introduced at most once. Similarly, stash dummy elements can only be introduced once as s is incremented after every such entry. □

Lemma 3. *The same v will not be queried upon twice between reshuffles at any level.*

Proof. Once v is queried upon at a level, i, either it is a "$dummy$" $\circ\, t$ value (in which case it will trivially never be queried again, as t is incremented at the next query), or it is some virtual memory location. In the latter case, v will be written to the top level after the query, and subsequent queries to v will find v before it reaches level i, and the only way v can reach a deeper level is if i is reshuffled. □

Lemma 4. *Every level except the top will always be empty or half-full (a half-full standard hash contains a number of records equal to half the number of buckets) and this state depends only on t.*

Proof. This lemma is similar to Lemma 1, except now we have to take into account stashing. However, whenever we stash an element, our reshuffling algorithm also introduces a stash dummy element inside the table. This causes the amount of elements to be shuffled into any level to always be the same. □

Lemma 5. *Any time a level i is reshuffled, its stash is included in the shuffle.*

Proof. Similar to the proof of Lemma 1, we observe that the only way a level is shuffled is if all previous levels are shuffled as well and become empty. Because the stash of level i was stored in the hierarchy above level i, no elements of the stash will fall below level i unless caused by a reshuffle, in which case it will be shuffled with level i. □

Theorem 4. *Under the assumption that one-way functions exists, the main construction is a secure two-server oblivious RAM.*

Proof. One-way functions allow private-key encryption and authentication. We use method of [26] to prevent tampering and thus must only show how to protect the access pattern.

We show how to simulate the view of a server's access pattern during the execution of the protocol upon any sequence of queries $q_1, \ldots, q_n$ knowing only the length n. We begin by first making the observation that every record is encrypted and will be re-encrypted whenever it is accessed. By the semantic security of the encryption,

we can assume that all these data contents are computationally indistinguishable from the encryption of any other contents. We also replace both the hash functions (which are modeled as PRFs) and the tagging PRF by random functions.

We first consider the view of each server during a reshuffle. If the server is playing the role of $\mathcal{S}_a$, after its initial message out, it sees a random sequence of encrypted records (real or dummy) with tags, and announced empty records. By Lemma 2, all the hidden records will contain elements with unique v's, and hence their tags will also be unique with overwhelming probability. The tags came from a random function that had not been previously used, and so the tags that the server sees are independent from its view. Furthermore, because of Lemmas 4 and 5, the number of empty records revealed will be deterministic and will not reveal any additional information. Thus, we can simulate this view by calculating the number of pre-determined items of each type, and use encryptions of 0 for all of them and tagging the appropriate records with completely random tags.

If the server is playing the role of $\mathcal{S}_b$ during a reshuffle, it will receive a sequence of encrypted records which reveals no information. Next, after it shuffles these records and sends them out, it receives back another sequence of encrypted records which also reveals no information. This view can be trivially simulated.

Finally, we argue that the sequence of reads can also be simulated. By the above arguments, we see that what each server holds at level i is nearly independent of its view, except for the fact that the tags of the records stored at that level are consistent with the hash function used at that level. By Lemma 3, between two reshuffles, the sequence of queries made to level i will all be distinct, but they may arbitrarily intersect the elements contained in level i. However, because only a negligible fraction of hash functions do not agree with the records in level i (i.e. would cause an overflow), the distribution of the outputs of the hash function applied to any sequence of distinct queries is statistically close to uniform[6]. Thus, we can simulate the probes to level i between reshuffles by a random sequence of probes.

□

4 Application to Secure Two-Party RAM Computation

In this section, we describe how our multi-party Oblivious RAM simulation can be applied to the setting of secure two-party computation on RAM programs. The idea of using Oblivious RAM for the purpose of secure computation has been suggested in the literature [24, 10, 18] and we outline the high-level idea of its use.

Consider the setting of two (semi-honest) parties, Alice and Bob, who wish to securely compute some function f (computed as some RAM program Π that runs in time $T = T(n)$ and uses $S = S(n)$ space) on their inputs A and B. Observe that in an Oblivious RAM, the view of the server can be simulated, so the idea is to let Alice or Bob play the role of the server (or in the case of our construction, two servers). However, in the case of Oblivious RAM, the privacy of the data is not protected from the client, so in order to securely run Π, we need to somehow simulate the client as well. In order to do so, we let the state of the client be *shared* between Alice and Bob so that neither party learns what is going on until the end of the computation when their outputs are revealed. In order to compute on this shared state, each *fixed instruction* of Π is encoded as a circuit. We emphasize that rather than unrolling the entire program into a circuit, which may be quite inefficient, we are only representing each atomic instruction as a circuit.

Because the joint state secure computation occurs at each step in the program, we want to minimize the amount of computation and communication overhead incurred by this step. In particular, in order for Alice and Bob to jointly compute Π simulate the state of the client efficiently, the client state should be as small as possible. This means that even if an ORAM solutions is efficient in terms of computation or communication overhead, we cannot use it if the footprint of the client is too large. In particular, works that require the memory of the client to be $O(\sqrt{n})$ (e.g. [33, 34]) or $O(n^\nu)$ (e.g. [16, 17]) will incur too much overhead per step of the program. The currently most

[6]Note that this does not hold true for plain cuckoo hashing, where there is a noticeable difference between a uniform hash function and one that makes a consistent cuckoo hash table.

efficient (single-server) ORAM protocol that is suitable for this purpose comes from the work of [23].

We point out that when modeling the client, we can either treat it as operating on bits or on "words". By this we mean the client may need, for example, pointers of $O(\log S)$ bits so that it can index into memory. The notion of a client having constant memory can implicitly mean that we are operating on words and these can each hold sufficiently many bits to perform the necessary instructions. However, when simulating the client steps using a circuit, we need to operate on bits rather than words.

Because of this, the client state may in fact be larger than a constant number of bits despite having only a constant number of words. In order not to accrue any additional overhead when performing the simulation of the client state, we need to use an efficient MPC that has only *constant* overhead. For example, the protocols of IKOS [19] or IPS [20] suit this situation.

By using our two-server ORAM solution in the Ostrovsky-Shoup compiler, we are able to achieve lower overhead for secure RAM computation than any known single-server ORAM solution. We have:

Theorem 5. *Suppose there exists a symmetric-key encryption scheme and a hash function modeled as a random function or an efficient PRF (e.g. the constant-overhead PRF of IKOS [19] under the existence of one-way functions or the PRF suggested in [18] under the DDH assumption in the OT-hybrid model). Suppose there exists a two-party secure circuit computation protocol (that is based on secret sharing) with constant overhead, e.g. [19, 20]. Then to securely compute a RAM program that runs in $T(n)$ time and uses $S(n)$ space, there exists a two-party secure RAM computation protocol in the semi-honest model with $O(\log T(n))$ multiplicative overhead in the security parameter. If the client computes on bits instead of words, there is an additional implicit $O(\log S)$ multiplicative overhead.*

Proof Sketch. We give a construction of such a scheme and argue that it is secure.

We follow the construction of the Ostrovsky-Shoup [28] compiler and let Alice and Bob hold inputs A and B respectively, and let Π be the program they wish to securely compute. Initialize the two-server ORAM as follows: let Alice play the role of one server and let Bob play the role of the other server. They jointly simulate the state of the client in our two-server ORAM protocol initialized to the secret sharing of the initial state. The parties then proceed by secret sharing A and B with each other. The two parties run the MPC protocol on the instructions that tells the client to obliviously insert (via ORAM) A and B into the locations inside the RAM where the program Π expects to read them as input. At the end of this process, Alice and Bob hold their respective encrypted server data as well as the shared state of the client.

Then, Alice and Bob begin to jointly execute the instructions of Π. Namely, they start with a shared state Σ and a shared value x and they perform the secure two-party computation on the circuit representing the step $\Pi(\Sigma, x)$ to receive a new shared state Σ' and a read or write operation op. The operation is converted into a sequence of oblivious instructions $op'_1, \ldots, op'_\ell$ by running the MPC on the two-server ORAM protocol steps. When the operation involves reading or writing from the server Alice is holding, Bob sends Alice his share of that instruction and Alice reconstructs the instruction and executes it on her server before re-sharing the result. Similarly, Alice reveals her share to Bob when the operation involves reading or writing from his server. At the end of execution of Π, Alice and Bob recombine shares to retrieve the output.

We follow the (standard) proof technique of composition of simulation of CPU and simulation of Oblivious RAM in which we invoke the simulatability of both the underlying MPC and ORAM (see also [18]). To simulate the view of one party, say Alice, we begin by generating a uniformly random share of the initial state for her view. As her input and Bob's input are being stored on the servers obliviously, we simulate the intermediate state shares Σ as random shares as well. To simulate the instruction execution via $(\Sigma', op) \leftarrow \Pi(\Sigma, x)$, again we generate uniform random shares for the intermediate state as well as the values retrieved. In a real execution, the resulting operation op is then converted into a sequence of oblivious instructions $op'_1, \ldots, op'_\ell$, and by the simulatability of the underlying oblivious RAM, we can in fact simulate the sequence by replacing op with a dummy operation. The simulator runs the sequence of oblivious instructions induced by this dummy operation and writes the sequences of Alice's memory probes to the simulated view.

Finally, when the output is about to be reconstructed, the simulator (which knows the result via interaction with the ideal functionality) sets the revealed share to be $r \oplus f(A, B)$ where r is the random share of the data for Alice during this final step.

Remark. We describe new techniques (to be expanded upon in the full version) on how to extend two-server ORAM simulation to be secure against *fully malicious* adversaries. This allows us, when combined with constant-overhead MPC secure against malicious adversaries (e.g. the IPS compiler [20]), to construct secure two-party RAM computation secure against malicious adversaries.

To ensure security against malicious adversaries, we first apply the methods of [27, 14] where each encrypted value stored on the server is also given a label about the address and epoch, then authenticated with a secure Message Authentication Code (MAC). Whenever the client retrieves data from the server, it decrypts and then verifies the MAC. To maintain efficiency, one can use the constant-overhead MACs constructed in [19]. Note that if the word size is larger than the security parameter, then any this does not incur any additional overhead as it can factored in to the word size overhead.

However, due to the fact that our two-server solution now offloads the duty of sorting to the servers, there needs to be an additional step in verifying the correctness of the sort. The client does not have enough memory to hold this entire array, so in order to verify the sort, the client queries adjacent positions i and $i + 1$ on the server and verifies that they are in the right order. Note that the server cannot fake these locations because the MAC of the label ensures that the data is in the correct location. Such a check is only performed during a reshuffle step, so it does not impact the overall amortized overhead of the scheme itself.

Finally, to apply this to maliciously secure two-party RAM computation, we must simulate the internal atomic instructions of the CPU using an MPC that is also secure against malicious adversaries. In order to do this efficiently, we use the IPS compiler [20] to obtain constant-overhead MPC (for circuits) secure against malicious adversaries in the OT-hybrid model.

5 Conclusion and Open Problems

In this paper, we introduced a new multi-server model for oblivious RAM and constructed a two-server scheme in this model. The scheme is secure against semi-honest servers, and with additional checks (described in the full version) is secure against malicious servers. The parameters of the scheme – $O(1)$ client memory, $O(n)$ server memory, and $O(\log n)$ overhead – match the lower bound of single-server oblivious RAM. The natural open problem to ask is whether or not the same lower bound holds, or if a better scheme can be constructed in this new model.

Our scheme was constructed under the assumption of the existence of one-way functions. We ask the open question of whether or not information-theoretic multi-server oblivious RAM can be constructed with similar parameters. One naive way of doing so would be to duplicate each server and use information-theoretic secret sharing between each server and its duplicate in order to replace encryption. The interesting question is to ask whether one can do so with fewer servers or perhaps better performance.

As an application of our two-server ORAM protocol, we also showed how to construct a two-party secure RAM computation protocol that is more efficient than existing constructions [28, 24, 18].

References

[1] Miklós Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In *STOC*, pages 181–190, 2010.

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. An o(n log n) sorting network. In *STOC*, pages 1–9, 1983.

[3] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[4] Michael Ben-Or, Shafi Goldwasser, Joe Kilian, and Avi Wigderson. Multi-prover interactive proofs: How to remove intractability assumptions. In *STOC*, pages 113–131, 1988.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[6] Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E. Skeith III. Public key encryption that allows pir queries. In *CRYPTO*, pages 50–67, 2007.

[7] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious ram practical. CSAIL Technical Report, MIT-CSAIL-TR-2011-018, 2011.

[8] Nishanth Chandran, Rafail Ostrovsky, and William E. Skeith III. Public-key encryption with efficient amortized updates. In *SCN*, pages 17–35, 2010.

[9] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, pages 41–50, 1995.

[10] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious ram without random oracles. In *TCC*, pages 144–163, 2011.

[11] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987.

[12] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, Cambridge, UK, 2001.

[13] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

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

[15] Michael T. Goodrich. Randomized shellsort: A simple oblivious sorting algorithm. In *SODA*, pages 1262–1277, 2010.

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

[17] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. *CoRR*, abs/1105.4125v1, May 2011.

[18] Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure computation with sublinear amortized work. Cryptology ePrint Archive, Report 2011/482, 2011.

[19] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *STOC*, pages 433–442, 2008.

[20] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, pages 572–591, 2008.

[21] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. In *ESA*, pages 611–622, 2008.

[22] 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.

[23] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *SODA*, 2012. To appear. See full version on ePrint [22].

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

[25] Rafail Ostrovsky. Apparatus system and method to efficiently search and modify information stored on remote servers, while hiding the access pattern. U.S. Patent Application No. 12,768,617. April 27, 2010.

[26] Rafail Ostrovsky. *Software protection and simulation on oblivious RAMs.* PhD thesis, Massachusetts Institute of Technology.

[27] Rafail Ostrovsky. Efficient computation on oblivious rams. In *STOC*, pages 514–523, 1990.

[28] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[29] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006.

[30] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *ESA*, pages 121–133, 2001.

[31] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *CRYPTO*, pages 502–519, 2010.

[32] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[33] Peter Williams and Radu Sion. Usable PIR. In *NDSS*, 2008.

[34] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security*, pages 139–148, 2008.

[35] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.