Oblivious Network RAM*

Dana Dachman-Soled[1,3] Chang Liu[2] Charalampos Papamanthou[1,3]
Elaine Shi[2,3] Uzi Vishkin[1,3]

Abstract

Oblivious RAM (ORAM) is a cryptographic primitive that allows a trusted CPU to securely access untrusted memory, such that the access patterns reveal nothing about sensitive data. ORAM is known to have broad applications in secure processor design and secure multi-party computation for big data. Unfortunately, due to a well-known logarithmic lower bound by Goldreich and Ostrovsky (Journal of the ACM, '96), ORAM is bound to incur a moderate cost in practice. In particular, with the latest developments in ORAM constructions, we are quickly approaching this limit, and the room for performance improvement is small.

In this paper, we consider new models of computation in which the cost of obliviousness can be fundamentally reduced in comparison with the standard ORAM model. We propose the Oblivious Network RAM model of computation, where a CPU communicates with multiple memory banks, such that the adversary observes only which bank the CPU is communicating with, but not the address offset within each memory bank. In other words, obliviousness within each bank comes for free—either because the architecture prevents a malicious party from observing the address accessed within a bank, or because another solution is employed to obfuscate memory accesses within each bank—and hence we only need to obfuscate the communication patterns between the CPU and the memory banks. We present several new constructions for obliviously simulating general or parallel programs in the Network RAM model. We describe applications of the Network RAM model in secure processor design and in distributed storage applications with a network adversary.

1 Introduction

Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky [18,19], allows a *trusted* CPU (or a trusted computational node) to obliviously access *untrusted* memory (or storage) during computation, such that an adversary cannot gain any sensitive information by observing the data access patterns. Although the community initially viewed ORAM mainly from a theoretical perspective, there has recently been an upsurge in research on both new efficient algorithms (c.f. [8,13,22,39,43,47,50]) and practical systems [9,11,12,21,33,38,41,42,48,52] for ORAM. Still the most efficient ORAM implementations [10,41,43] require a relatively large bandwidth blowup, and part of this is inevitable in the standard ORAM model. Fundamentally, a well-known lower bound by Goldreich and Ostrovsky states that any ORAM scheme with constant CPU cache must incur at least $\Omega(\log N)$ blowup, where N is the number of memory words, in terms of bandwidth and runtime. To make ORAM

*danadach@ece.umd.edu, liuchang@cs.umd.edu, cpap@umd.edu, elaine@cs.umd.edu, vishkin@umiacs.umd.edu,
1: University of Maryland, Department of Electrical and Computer Engineering
2: University of Maryland, Department of Computer Science
3: University of Maryland Institute for Advanced Computer Studies (UMIACS)

techniques practical in real-life applications, we wish to further reduce its performance overhead. However, since latest ORAM schemes [43,47] have practical performance that approaches the limit of the Goldreich-Ostrovsky lower bound, the room for improvement is small in the standard ORAM model. In this paper, we investigate the following question:

> *In what alternative, realistic models of computation can we significantly lower the cost of oblivious data accesses?*

Motivated by practical applications, we propose the Network RAM (NRAM) model of computation and correspondingly, Oblivious Network RAM (O-NRAM). In this new model, one or more CPUs interact with M memory banks during execution. Therefore, each memory reference includes a *bank identifier*, and an *offset* within the specified memory bank. We assume that an *adversary cannot observe the address offset within a memory bank, but can observe which memory bank the CPU is communicating with.* In other words, obliviousness within each bank "comes for free". Under such a threat model, an Oblivious NRAM (O-NRAM) can be informally defined as an NRAM whose observable memory traces (consisting of the bank identifiers for each memory request) do not leak information about a program's private inputs (beyond the length of the execution). In other words, in an O-NRAM, the sequence of bank identifiers accessed during a program's execution must be provably obfuscated.

1.1 Practical Applications

The NRAM and O-NRAM models of computation are motivated by two primary application domains:

- **Secure processor architecture.** Today, secure processor architectures [1,12,33,38,44,45] are designed assuming that the memory system is *passive* and untrusted. In particular, an adversary can observe both memory contents and memory addresses during program execution. To secure against such an adversary, the trusted CPU must both encrypt data written to memory, and obfuscate memory access patterns.

  Our new O-NRAM model provides a realistic alternative that has been mentioned in the architecture community [33, 34] and was inspired by the Module Parallel Computer (MPC) model of Melhorn and Vishkin [35]. The idea is to introduce *trusted* decryption logic on the memory DIMMs (for decrypting memory addresses). This way, the CPU can encrypt the memory addresses before transmitting them over the insecure memory bus. In contrast with traditional passive memory, we refer to this new type of memory technology as *active memory*. In a simple model where a CPU communicates with a single active memory bank, obliviousness is automatically guaranteed, since the adversary can observe only *encrypted* memory contents and addresses. However, when there are multiple such active memory banks, we must obfuscate which memory bank the CPU is communicating with.

- **Distributed storage with a network adversary.** Consider a scenario where a client (or a compute node) stores private, encrypted data on multiple distributed storage servers. We consider a setting where all endpoints (including the client and the storage servers) are *trusted*, but the network is an *untrusted* intermediary. In practice, trust in a storage server can be bootstrapped through means of trusted hardware such as the Trusted Platform Module (TPM) or as IBM 4758; and network communication between endpoints can be encrypted using standard

SSL. Trusted storage servers have also been built in the systems community [3]. On the other hand, the untrusted network intermediary can take different forms in practice, e.g., an untrusted network router or WiFi access point, untrusted peers in a peer-to-peer network such as Bitcoin or TOR, or packet sniffers in the same local area network. Achieving oblivious data access against such a network adversary is precisely captured by our O-NRAM model.

1.2 Results and Contributions

We introduce the Oblivious Network RAM model, and we conduct the first *systematic* study to understand the "cost of obliviousness" in this model. We consider running both *sequential* programs and *parallel* progams in this new setting. We propose novel algorithms capable of exploiting the "free obliviousness" within each bank, such that the cost of obliviousness is significantly lower in comparison with the standard Oblivious (Parallel) RAMs. We give a summary of our results below.

First, observe that if there are only $O(1)$ number of memory banks, there is a trivial solution with $O(1)$ cost: just make one memory access (real or dummy) to each bank for each step of execution. On the other hand, if there are $\Omega(N)$ memory banks each of constant size (where N denotes the total number of memory words), then the problem approaches standard ORAM [18,19] or OPRAM [7]. The intermediate parameters are therefore the most interesting. For simplicity, in this section, we mainly state our results for the most interesting case when the number of banks $M = O(\sqrt{N})$, and each bank can store up to $O(\sqrt{N})$ words. In Sections 3, 4 and 5, our results will be stated for more general parameter choices.

We now state our main results. An overview of our results is also provided in Table 1.

"Sequential-to-sequential" compiler. First, we show that any RAM program can be obliviously simulated on a Network RAM, consuming only $O(1)$ words of local CPU cache, with $\widehat{O}(\log N)$ blowup in both runtime and bandwidth, where–throughout the paper–when we say the complexity of our scheme is $\widehat{O}(f(N))$, we mean that for any choice of $h(N) = \omega(f(N))$, our scheme attains complexity $g(N) = O(h(N))$. Further, when the RAM program has $\Omega(\log^2 N)$ memory word size, it can be obliviously simulated on Network RAM with only $\widehat{O}(1)$ bandwidth blowup (assuming non-uniform memory word sizes as used by Stefanov et al. in [42]).

In comparison, the best known (constant CPU cache) ORAM scheme has roughly $\widehat{O}(\log N)$ bandwidth blowup for $\Omega(\log^2 N)$ memory word size [47]. For smaller memory words, the best known ORAM scheme has $O(\log^2 / \log\log N)$ blowup in both runtime and bandwidth [26].

"Parallel-to-sequential" compiler. We demonstrate that parallelism can facilitate obliviousness, by showing that programs with a "sufficient degree of parallelism" – specifically, programs whose degree of parallelism $P = \omega(M \log N)$ – can be obliviously simulated in the Network RAM model with only $O(1)$ blowup in runtime and bandwidth. Here, we consider parallelism as a property of the program, but are not in fact executing the program on a parallel machine. The overhead stated above is for the sequential setting, i.e., considering that both the NRAM and the O-NRAM has a single processor.

"Parallel-to-parallel" compiler. Finally, we consider oblivious simulation in the parallel setting. We show that for any parallel program executing in t parallel-steps with $P = M^{1+\delta}$ processors, we can obliviously simulate the program on a Network PRAM with $P' = O(P/\log^* P)$ processors, running in time $O(t \log^* P)$ time, thereby achieving $O(\log^* P)$ blowup in parallel runtime and bandwidth, and optimal total work. In comparison, the best known OPRAM scheme has poly $\log N$

Setting	RAM to O-NRAM blowup	*c.f.* Best known ORAM blowup
Sequential-to-sequential compiler		
W =small	$\widehat{O}(\log N)$	$O(\log^2 N/\log\log N)$ [26]
$W = \Omega(\log^2 N)$	bandwidth: $\widehat{O}(1)$ runtime: $\widehat{O}(\log N)$	bandwidth: $\widehat{O}(\log N)$ [47] runtime: $O(\log^2 N/\log\log N)$ [26]
$W = \Omega(N^\epsilon)$	$\widehat{O}(1)$	$\widehat{O}(\log N)$ [47]
Parallel-to-sequential compiler		
$\omega(M \log N)$-parallel	$O(1)$	Same as standard ORAM
Parallel-to-parallel compiler		
$M^{1+\delta}$-parallel for any const $\delta > 0$	$O(\log^* N)$	best known: poly $\log N$ [7] lower bound: $\Omega(\log N)$

Table 1: **A systematic study of "cost of obliviousness" in the Network ORAM model.** W denotes the memory word size in # bits, N denotes the total number of memory words, and M denotes the number of memory banks. For simplicity, this table assumes that $M = O(\sqrt{N})$, and each bank has $O(\sqrt{N})$ words. Like implicit in existing ORAM works [19, 26], small word size assumes at least $\log N$ bits per word—enough to store a virtual address of the word.

blowup in parallel runtime and bandwidth,

1.3 Technical Highlights

Our most interesting technique is in constructing the parallel-to-parallel compiler. We achieve this through an intermediate stepping stone where we first construct a parallel-to-sequential compiler (which may be of independent interest).

At a high level, the idea is to assign each virtual address to a pseudorandom memory bank (and this assignment stays the same during the entire execution). Suppose that a program is sufficiently parallel such that it always makes memory requests in $P = \omega(M \log N)$-sized batches. For now, assume that all memory requests within a batch operate on *distinct* virtual addresses – if not we can leverage a hash table to suppress duplicates, using an additional "scratch" bank as the CPU's working memory. Then, clearly each memory bank will in expectation serve P/M requests for each batch. With a simple Chernoff bound, we can conclude that each memory bank will serve $O(P/M) + f(N)$, for any $f(N) \in \omega(\log N)$ requests for each batch, except with *negligible* probability. In a sequential setting, we can easily achieve $O(1)$ bandwidth and runtime blowup: for each batch of memory requests, the CPU will sequentially access each bank $O(P/M) + f(N)$, for any $f(N) \in \omega(\log N)$ number of times, padding with dummy accesses if necessary (see Section 4).

However, additional difficulties arise when we try to execute the above algorithm in parallel. In each step, there is a batch of P memory requests, one coming from each processor. However, each processor cannot perform its own memory request, since the adversary can observe which processor is talking to which memory bank and can detect duplicates (note this problem did not exist in the sequential case since there was only one processor). Instead, we wish to

1. "sort" the memory requests according to their corresponding banks while suppressing duplicates; and

2. pad the number of accesses to each bank to a worst-case maximum – as mentioned earlier, if we suppressed duplicate addresses, each bank has $P/M + \omega(\log N)$ requests with probability $1 - \text{negl}(N)$.

At this point, we can assign processors to the memory requests in a round-robin manner, such that which processor accesses which bank is "fixed". Now, to achieve the above two tasks in $O(\log^* P)$ parallel time, we need to employ non-trivial parallel algorithms for "colored compaction" [4] and "static hashing" [5, 17], while using a scratch bank as working memory (see Section 5).

1.4 Related Work

ORAM: the theory. Oblivious RAM (ORAM) was first proposed in a seminal work by Goldreich and Ostrovsky [18, 19] where they laid a vision of employing an ORAM-capable secure processor to protect software against piracy. In their work, Goldreich and Ostrovsky showed both a poly-logarithmic upper-bound (commonly referred to as the hierarchical ORAM framework) and a logarithmic lower-bound for ORAM—both under constant CPU cache. Goldreich and Ostrovsky's hierarchical construction was improved in several subsequent works [6, 20, 22, 26, 36, 49–51]. Recently, Shi *et al.* proposed a new, tree-based paradigm for constructing ORAMs [39], thus leading to several new constructions that are conceptually simple and practically efficient [8, 13, 43, 47]. Notably, Circuit ORAM [47] partially resolved the tightness of the Goldreich-Ostrovsky lower bound, by showing that certain stronger interpretations of their lower bound are indeed tight.

Theoretically, the best known ORAM scheme (with constant CPU cache) for small $O(\log N)$-sized memory words[1] is a construction by Kushilevitz *et al.* [26], achieving $O(\log^2 N / \log \log N)$ bandwidth and runtime blowup. Path ORAM (variant with $O(1)$ CPU cache [48]) and Circuit ORAM can achieve better bounds for bigger memory words. For example, Circuit ORAM achieves $O(\log N)\omega(1)$ bandwidth blowup for a word size of $\Omega(\log^2 N)$ bits; and for $O(\log N)\omega(1)$ runtime blowup for a memory word size of N^ϵ bits where $0 < \epsilon < 1$ is any constant within the specified range.

ORAMs with larger CPU cache sizes (caching up to N^α words for any constant $0 < \alpha < 1$) have been suggested for cloud storage outsouring applications [20, 42, 51]. In this setting, Goodrich and Mitzenmacher [20] first showed how to achieve $O(\log N)$ bandwidth and runtime blowup.

Other than secure processors and cloud outsourcing, ORAM is also noted as a key primitive for scaling secure multi-party computation to big data [23, 29, 47, 48]. In this context, Wang *et al.* [47, 48] pointed out that the most relevant ORAM metric should be the circuit size rather than the traditionally considered bandwidth metrics. In the secure computation context, Lu and Ostrovsky [30] proposed a two-server ORAM scheme that achieves $O(\log N)$ runtime blowup. Similarly, ORAM can also be applied in other RAM-model cryptographic primitives such as (reusable) Garbled RAM [14–16, 31, 32].

Recently, Boyle, Chung and Pass [7] proposed the notion of Oblivious Parallel RAM, and presented a construction for oblivious simulation of PRAMs in the PRAM model. Our result is

[1] Every memory word must be large enough to store the logical memory address.

incomparible to their result: Our security model is weaker than theirs since we assume obliviousness within each memory bank comes for free; on the other hand, we obtain far better asymptotical and concrete performance. Goodrich and Mitzenmacher [20] observed that computational tasks that can be expressed in the streaming map-reduce model can be transformed into efficient oblivious counterparts in the standard (sequential) RAM model. Like Goodrich and Mitzenmacher, we consider how programs with inherent parallelism can be transformed into efficient, oblivious counterparts in the sequential setting—but our techniques apply to the NRAM model of computation.

ORAM: implementations. Path ORAM [43] and variants [10] enabled the first ORAM-capable secure processors to be prototyped [9, 11, 12, 33, 34, 38]. Several notable implementation have been endeavored [40, 41, 51, 52] for cloud outsourcing applications. Recent efforts have also implemented ORAMs in a secure computation context [23, 48]. Circuit ORAM is currently the scheme of choice in the secure computation context [47]. Liu, Hicks, and Shi [28] were the first to apply programming language techniques to the problem of oblivious program execution. Their vision of memory-trace oblivious program execution was subsequently implemented both on an ORAM-capable secure processor backend [27], and on a secure computation backend [29].

2 Definitions

2.1 Background: Random Access Machines (RAM)

We consider RAM programs to be interactive stateful systems $\langle \Pi, \mathsf{state}, D \rangle$, consisting of a memory array D of N memory words, a CPU state denoted state, and a next instruction function Π which given the current CPU state and a value rdata read from memory, outputs the next instruction I and an updated CPU state denoted state':

$$(\mathsf{state}', I) \leftarrow \Pi(\mathsf{state}, \mathsf{rdata})$$

Each instruction I is of the form $I = (\mathsf{op}, \ldots)$, where op is called the op-code whose value is read, write, or stop. The initial CPU state is set to $(\mathsf{start}, *, \mathsf{state}_{\mathsf{init}})$. Upon input x, the RAM machine executes, computes output z and terminates. CPU state is reset to $(\mathsf{start}, *, \mathsf{state}_{\mathsf{init}})$ when the computation on the current input terminates.

On input x, the execution of the RAM proceeds as follows. If $\mathsf{state} = (\mathsf{start}, *, \mathsf{state}_{\mathsf{init}})$, set $\mathsf{state} := (\mathsf{start}, x, \mathsf{state}_{\mathsf{init}})$, and $\mathsf{rdata} := 0$. Now, repeat the $\mathsf{doNext}()$ till termination, where $\mathsf{doNext}()$ is defined as below:

$\mathsf{doNext}()$

1. Compute $(I, \mathsf{state}') = \Pi(\mathsf{state}, \mathsf{rdata})$. Set $\mathsf{state} := \mathsf{state}'$.
2. If $I = (\mathsf{stop}, z)$ then terminate with output z.
3. If $I = (\mathsf{write}, \mathsf{vaddr}, \mathsf{wdata})$ then set $D[\mathsf{vaddr}] := \mathsf{wdata}$.
4. If $I = (\mathsf{read}, \mathsf{vaddr}, \perp)$ then set $\mathsf{rdata} := D[\mathsf{vaddr}]$.

2.2 Network RAM (NRAM)

Nework RAM. A Network RAM (NRAM) is the same as a regular RAM, except that memory is distributed across multiple banks, $\mathsf{Bank}_1, \ldots, \mathsf{Bank}_M$. In an NRAM, every virtual address vaddr

can be written in the format $\mathsf{vaddr} := (m, \mathit{offset})$, where $m \in [M]$ is the bank identifier, and *offset* is the offset within the Bank_m. Otherwise, the definition of NRAM is identical to the definition of RAM.

Probablistic NRAM. Similar to the probablistic RAM notion formalized by Goldreich and Ostrovsky [18, 19], we additionally define a *probablistic NRAM*. A probablistic NRAM is an NRAM whose CPU state is initialized with randomness ρ (that is unobservable to the adversary). If an NRAM is deterministic, we can simply assume that the CPU's initial randomness is fixed to $\rho := 0$. Therefore, a deterministic NRAM can be considered as a special case of a probablistic NRAM.

Outcome of execution. Throughout the paper, we use the notation $\mathsf{RAM}(x)$ or $\mathsf{NRAM}(x)$ to denote the outcome of executing a RAM or NRAM on input x. Similarly, for a probablistic NRAM, we use the notation $\mathsf{NRAM}_\rho(x)$ to denote the outcome of executing on input x, when the CPU's initial randomness is ρ.

2.3 Oblivious Network RAM (O-NRAM)

Observable traces. To define Oblivious Network RAM, we need to first specify which part of the memory trace an adversary is allowed to observe during a program's execution. As mentioned earlier in the introduction, each memory bank has trusted logic for encrypting and decrypting the memory offset. The offset within a bank is transferred in encrypted format on the memory bus. Hence, *for each memory access* $\mathsf{op} :=$ *"*read*"* *or* $\mathsf{op} :=$ *"*write*"* *to virtual address* $\mathsf{vaddr} := (m, \mathit{offset})$*, the adversary observes only the op-code* op *and the bank identifier* m*, but not the offset within the bank.*

Definition 1 (Observable traces)**.** *For a probabilistic* NRAM*, we use the notation* $\mathsf{Tr}_\rho(\mathsf{NRAM}, x)$ *to denote its observable traces upon input* x*, and initial CPU randomness* ρ*:*

$$\mathsf{Tr}_\rho(\mathsf{NRAM}, x) := \{(\mathsf{op}_1, m_1), (\mathsf{op}_2, m_2), \ldots, (\mathsf{op}_T, m_T)\}$$

where T *is the total execution time of the* NRAM*, and* (op_i, m_i) *is the op-code and memory bank identifier during step* $i \in [T]$ *of the execution.*

We remark that one can consider a slight variant model where the opcodes $\{\mathsf{op}_i\}_{i\in[T]}$ are also hidden from the adversary. Since to hide whether the operation is a read or write, one can simply perform one read and one write for each operation – the differences between these two models are insignificant for technical purposes. Therefore, in this paper, we consider the model whose observable traces are defined in Definition 1).

Oblivious Network RAM. Intuitively, an NRAM is said to be oblivious, if for any two inputs x_0 and x_1 resulting in the same execution time, their observable memory traces are computationally indistinguishable to an adversary.

For simplicity, we define obliviousness for NRAMs that run in deterministic T time regardless of the inputs and the CPU's initial randomness. One can also think of T as the worst-case execution time, and that the program is always padded to the worst-case execution time. Oblivious NRAM can also be similarly defined when its runtime is randomized – however we omit the definition in this paper.

Definition 2 (Oblivious Network RAM)**.** *Consider an* NRAM *that runs in deterministic time* $T = \text{poly}(\lambda)$*. The* NRAM *is said to be computationally oblivious if no polynomial-time adversary* $\mathcal{A}$ *can win the following security game with more than* $\frac{1}{2} + \text{negl}(\lambda)$ *probability. Similarly, the* NRAM *is said to be statistically oblivious if no adversary, even computationally unbounded ones, can win the following game with more than* $\frac{1}{2} + \text{negl}(\lambda)$ *probability.*

- $\mathcal{A}$ *chooses two inputs* x_0 *and* x_1 *and submits them to a challenger.*
- *The challenger selects* $\rho \in \{0,1\}^\lambda$*, and a random bit* $b \in \{0,1\}$*. The challenger executes* NRAM *with initial randomness* ρ *and input* x_b *for exactly* T *steps, and gives the adversary* $\mathsf{Tr}_\rho(\mathsf{NRAM}, x_b)$*.*
- $\mathcal{A}$ *outputs a guess* b' *of* b*, and wins the game if* $b' = b$*.*

2.4 Notion of Simulation

Definition 3 (Simulation)**.** *We say that a deterministic* $\mathsf{RAM} := \langle \Pi, \mathsf{state}, D \rangle$ *can be correctly simulated by another probabilistic* $\mathsf{NRAM} := \langle \Pi', \mathsf{state}', D' \rangle$ *if for any input* x *for any initial CPU randomness* ρ*,* $\mathsf{RAM}(x) = \mathsf{NRAM}_\rho(x)$*. Moreover, if* NRAM *is oblivious, we say that* NRAM *is an oblivious simulation of* RAM*.*

Below, we explain some subtleties regarding the model, and define the metrics for oblivious simulation.

Uniform vs. non-uniform memory word size. The O-NRAM simulation can either employ uniform memory word size or non-uniform memory word size. For example, the non-uniform word size model has been employed for recursion-based ORAMs in the literature [43, 47]. In particular, Stefanov *et al.* describe a parametrization trick where they use a smaller word size for position map levels of the recursion [43].

Metrics for simulation overhead. In the ORAM literature, several performance metrics have been considered. To avoid confusion, we now explicitly define two metrics that we will adopt later. If an NRAM correctly simulates a RAM, we can quantify the overhead of the NRAM using the following metrics.

- **Runtime blowup.** If a RAM runs in time T, and its oblivious simulation runs in time T', then the runtime blowup is defined to be T'/T. This notion is adopted by Goldreich and Ostrovsky in their original ORAM paper [18, 19].
- **Bandwidth blowup.** If a RAM transfers Y bits between the CPU and memory, and its oblivious simulation transfers Y' bits, then the bandwidth blowup is defined to be Y'/Y. Clearly, if the oblivious simulation is in a uniform word size model, then bandwidth blowup is equivalent to runtime blowup. However, bandwidth blowup may not be equal to runtime blowup in a non-uniform word size model.

In this paper, we consider oblivious simulation of RAMs in the NRAM model, and we focus on the case when the Oblivious NRAM has only $O(1)$ words of CPU cache.

3 Sequential Oblivious Simulation

We first consider oblivious simulation of arbitrary RAMs in the NRAM model.

3.1 First Attempt: Oblivious NRAM with $O(M)$ CPU Cache

Let M denote the number of memory banks in our NRAM, where each bank has $O(N/M)$ capacity. We first describe a simple Oblivious NRAM with $O(M)$ CPU private cache. Under a non-uniform memory word size model, Our O-NRAM construction achieves $O(1)$ bandwidth blowup under $\Omega(\log^2 N)$ memory word size. Later, in Section 3.2, we will describe how to reduce the CPU cache to $O(1)$ by introducing an additional scratch memory bank of $O(M)$ in size. In particular, an interesting parametrization point is when $M = O(\sqrt{N})$.

Our idea is inspired by the partition-based ORAM idea described by Stefanov, Shi, and Song [42]. For simplicity, Like many earlier ORAM works [19, 39], we focus on presenting the algorithm for making memory accesses, namely the Access algorithm. A description of the full O-NRAM construction is apparent from the Access algorithm: basically, the CPU interleaves computation (namely, computing the next-instruction function Π) with the memory accesses.

CPU private cache. The CPU needs to maintain the following metadata:

- A *position map* that stores which bank each memory word currently resides in. We use the notation position[vaddr] to denote the bank identifier for the memory word at virtual address vaddr. Although storing the position map takes $O(N \log M)$ bits of CPU cache, we will later describe a recursion technique [39, 42] that can reduce this storage to $O(1)$; and
- An *eviction cache* consisting of M *queues*. The queues are used for temporarily buffering memory words before they are obliviously written back to the memory banks. Each queue $m \in [M]$ can be considered as an extension of the m-th memory bank. The eviction cache is $O(M) + f(N)$, for any $f(N) = \omega(\log N)$ in size. For now, consider that the eviction cache is stored in the CPU cache, such that accesses to the eviction cache do not introduce memory accesses. Later in Section 3.2, we will move the eviction cache to a separate scratch bank – it turns out that to do this, there is a small technicality that requires us to use a deamortized Cuckoo hash table [2].

Memory access operations. Figure 1 describes the algorithm for making a memory access. To access a memory word identified by virtual address vaddr, the CPU first looks up the position map $m :=$ position[vaddr] to find the bank identifier m where the memory word vaddr currently resides. Then, the CPU fetches the memory word vaddr from the bank m. Since the set of vaddr's stored in any bank may be discontiguous, we first assume that each bank implements a hash table such that one can look up each memory location by its vaddr. Later, we will describe how to instantiate this hash table (Theorem 1). After fetching the memory word vaddr, the CPU assigns it to a fresh random bank $\widetilde{m}$. However, to avoid leaking information, the memory word is not immediately written back to the bank $\widetilde{m}$. Instead, recall that the CPU maintains M queues for buffering memory words before write-back. At this point, the memory word at address vaddr is added to queue[$\widetilde{m}$] — signifying that the memory word vaddr is scheduled to be written back to Bank$_{\widetilde{m}}$.

```
Access(op, vaddr, wdata):
 1: m̃ ← UniformRandom(1 . . . M)
 2: m := position[vaddr], position[vaddr] := m̃
 3: if word at vaddr is in queue[m] then
 4:     rdata := queue[m].ReadAndRm(vaddr)
 5:     ReadBank(m, ⊥)
 6: else
 7:     queue[m].ReadAndRm(⊥)
    /* Dummy operation, needed when the eviction queues are stored in a scratch bank, see Section 3.2 */
 8:     rdata := ReadBank(m, vaddr)
    /* Each bank implements a hash table with good worst-case cost (Theorem 1). */
 9: end if
10: if op = read then wdata := rdata
11: queue[m̃] := queue[m̃].push(vaddr, wdata)
12: Call SeqEvict(ν)
13: return rdata
```

Figure 1: **Algorithm for data access**. Read or write a memory word identified by vaddr. If op = read, the input parameter wdata = None, and the Access operation returns the newly fetched word. If op = write, the Access operation writes the specified wdata to the memory word identified by vaddr, and returns the old value of the word at vaddr.

Background eviction. To prevent the CPU's eviction queues from overflowing, a background eviction process obliviously evicts words from the queues back to the memory banks. One possible eviction strategy is that, on each data access, the CPU chooses $\nu = 2$ queues for eviction – by sequentially cycling through the queues. When a queue is chosen for eviction, an arbitrary block is popped from the queue and written back to the corresponding memory bank. If the chosen queue is empty, a dummy block is evicted to prevent leaking information. Stefanov *et al.* proved that such an eviction process is fast enough so that the CPU's eviction cache load is bounded by $O(M)$ except with negligible probability [42] – assuming that $M = \omega(\log N)$.

Lemma 1. *The CPU's eviction cache is bounded by* $O(M) + f(N)$*, for any* $f(N) = \omega(\log N)$ *words except with* $\mathrm{negl}(N)$ *probability.*

Proof. The proof follows from Stefanov *et al.* [42], and is a straightforward application of Chernoff bound. □

Instantiating the per-bank hash table. In Figures 1 and 2, we assume that each bank implements a hash table with good worst-case performance. We now describe how to instantiate this hash table to achieve $\widehat{O}(1)$ cost per operation except with negligible failure probability.

A first idea is to implement a standard Cuckoo hash table [37] for each memory bank. In this way, lookup is worst-case constant time, whereas insertion is average-case constant time, and worst-case $\widehat{O}(\log N)$ time to achieve a failure probability negligible in N. To ensure obliviousness, we can not reveal the insertion time – for example, insertion time can reveal the current usage of each bank, which in turns leaks additional information about the access pattern. However, we do not wish to incur this worst-case insertion time upon every write-back.

```
Evict(m):
1: if len(queue[m]) = 0 then
2:     WriteBank(m, ⊥, None)
3: else
4:     (vaddr, wdata) := queue[m].pop()
5:     WriteBank(m, vaddr, wdata)
6: end if
```

```
SeqEvict(ν):
// Let cnt denote a stateful counter.
1: Repeat ν times:
2:     cnt := (cnt + 1) mod M
3:     Evict(cnt)
```

Figure 2: **Background evicition algorithms with eviction rate ν.** SeqEvict linearly cycles through the eviction queues to evict from. If a queue selected for eviction is empty, evict a dummy word for obliviousness. Counter cnt is a global variable.

To deal with this issue, we will rely on a deamortized Cuckoo hash table such as the one described by Arbitman *et al.* [2]. Their idea is to rely on a small queue that temporarily buffers the pending work associated with insertions. Upon every operation, perform a fixed amount of work at a rate faster than the amortized insertion cost of a standard Cuckoo hash table. For our application, we require that the failure probability be negligible in N. Therefore, we introduce a modified version of Arbitman *et al.*'s theorem [2] as stated below.

Theorem 1 (Deamortized Cuckoo hash table: negligible failure probability version)**.** *There exists a deamortized Cuckoo hash table of capacity $s = \text{poly}(N)$ such that with probability $1 - \text{negl}(N)$, each insertion, deletion, and lookup operation is performed in worst-case $\widehat{O}(1)$ time (not counting the cost of operating on the queue) – as long as at any point in time at most s elements are stored in the data structure. The above deamortized Cuckoo hash table consumes $O(s) + O(N^\delta)$ space where $0 < \delta < 1$ is a constant.*

In the above, the $O(s)$ part of the space corresponds to the two tables T_0 and T_1 for storing the elements of the hash table, and the additional $O(N^\delta)$ space is for implementing the pending work queue (see Arbitman *et al.* [2] for more details). Specifically, Arbitman *et al.* suggested that the work queue be implemented with constant number k of standard hash tables which are N^δ, for $\delta < 1$, in size. To achieve negligible failure probability, we instead set $k = k(N)$ to be any $k(N) = \omega(1)$. See Appendix A for details of our modified construction and analysis.

Recursion. In the above scheme, the CPU stores both a position map of $\Theta(N \log N)$ bits, and an eviction cache containing $\Theta(M)$ memory words. On each data access, the CPU reads $\Theta(w)$ bits assuming each a memory word is of w bits. Therefore, the bandwidth blowup is $O(1)$.

We now show how to rely on a recursion idea to avoid storing this position map inside the CPU — for now, assume that the CPU still stores the eviction cache, which we will get rid of in Section 3.2. The idea is to recursively store the position map in smaller Oblivious NRAMs. Specifically, consider a sequence of ONRAMs denoted $\mathsf{ONRAM}_0, \mathsf{ONRAM}_1, \ldots, \mathsf{ONRAM}_d$, where ONRAM_0 is the actual data ONRAM, whereas all other ONRAMs are metadata ONRAMs. The position map of ONRAM_i is stored in ONRAM_{i+1}, and the recursion is repeated until we reach an ONRAM_d of constant size. To access a memory word in ONRAM_0, the client first makes a position map lookup in ONRAM_1 which triggers a recursive call to look up the position of the position in ONRAM_2, and so on.

The original binary-tree ORAM [39] described a simple way to parametrize the recursion, using a *uniform* memory word size across all recursion levels. Later schemes [43,47], however, described new

tricks to parametrize the recursion, where a different memory word size is chosen for all the metadata levels than the data level (i.e., ONRAM_0) – the latter trick allows one to reduce the bandwidth blowup for reasonably big memory words size. Below, we describe these parametrizations, state the bandwidth blowup and runtime blowup we achieve in each setting. Recall that as mentioned earlier, the bandwidth blowup and runtime blowup equate for a uniform memory word size setting; however, under non-uniform memory word sizes, the two metrics may not equate.

- *Uniform memory word size.* The depth of recursion is smaller when the memory word is larger.
  - Assume that each memory word is at least $c \log N$ bits in size for some constant $c > 1$. In this case, the recursion depth is $O(\log N)$. Hence, the resulting O-NRAM has $\widehat{O}(\log N)$ runtime blowup and bandwidth blowup.
  - Assume that each memory word is at least N^ϵ bits in size for some constant $0 < \epsilon < 1$. In this case, the recursion depth is $O(1)$. Hence, the resulting O-NRAM has $\widehat{O}(1)$ runtime blowup and bandwidth blowup.
- *Non-uniform memory word size.* Using a parametrization trick by Stefanov *et al.* [43], we can also parametrize the position map recursion levels to have a different word size than the data level. Of particular interest is the following point of parametrization:
  - Assume that each memory word of the original RAM is $W = \Omega(\log^2 N)$ bits — this will be the word size for the data level of the recursion. For the position map levels, we will use a word size of $\Theta(\log N)$ bits. In this way, the recursion depth is $O(\log N)$. For each access, the total number of bits transferred include one data word of W bits, and $O(\log N)$ words of $O(\log N)$ bits. Thus, we achieve $\widehat{O}(1)$ bandwidth blowup, but $\widehat{O}(\log N)$ run-time blowup.

Finally, observe that *we need not create separate memory banks for each level of the recursion.* In fact, the recursion levels can simply reuse the memory banks of the top data level, introducing only a constant factor blowup in the size of the memory bank.

3.2 Achieving $O(1)$ Words of CPU Cache

We now explain how to reduce the CPU cache size to $O(1)$, while storing the eviction queues in a separate scratch bank. It turns out that there is a small technicality when we try to do so, requiring the use of a special data structure as described below. When we move the eviction queues to the scratch bank, we would like each queue to support the operations: $\mathsf{pop}()$, $\mathsf{push}()$ and $\mathsf{ReadAndRm}()$, as required by algorithms in Figures 1 and 2 with worst-case $\widehat{O}(1)$ cost except with $\mathrm{negl}(N)$ failure probability. While a simple queue supports $\mathsf{pop}()$ and $\mathsf{push}()$ with these time bounds, it does not not support $\mathsf{ReadAndRm}()$. To achieve this, the scratch bank will maintain the following data structures:

- Store M eviction queues supporting only $\mathsf{pop}()$ and $\mathsf{push}()$ operations. The total number of elements in all queues does not exceed $O(M) + f(N)$ for any $f(N) = \omega(\log N)$ except with negligible failure probability. It is not hard to see that these M eviction queues can be implemented with $O(M) + f(N)$ for any $f(N) = \omega(\log N)$ space in total and $O(1)$ cost per operation.

- Separately, store the entries of all M eviction queues in a single deamortized Cuckoo hash table [2] inside the scratch bank. Such a deamortized Cuckoo hash table can achieve $\widehat{O}(1)$ cost per operation (insertion, removal, lookup) except with negligible failure probability. When an element is *popped from* or *pushed to* any of the eviction queues, it is also inserted or removed in this big deamortized Cuckoo hash table. However, when an element must be *read and removed* from any of the eviction queues, then the element is looked up from the big hash table and it is just marked as **deleted**. When time comes for this element to be popped from some queue during the eviction process, a dummy eviction is performed.

Theorem 2 (O-NRAM simulation of arbitrary RAM programs: **uniform** word size model)**.** *Any N-word RAM with a word size of $W = f(N) \log N$ bits can be simulated by an Oblivious NRAM that consumes $O(1)$ words of CPU cache, and with $O(M)$ memory banks each of $O(M + N/M + N^\delta)$ words in size, for any constant $0 < \delta < 1$. The oblivious NRAM simulation incurs $\widehat{O}(\log_{f(N)} N)$ run-time blowup and bandwidth blowup. As special cases of interest:*

- *When the word size is $W = N^\epsilon$ bits, the run-time blowup and bandwidth blowup are both $\widehat{O}(1)$.*
- *When the word size is $W = c \log N$ bits for some constant $c > 1$, the run-time blowup and bandwidth blowup are both $\widehat{O}(\log N)$.*

Theorem 3 (O-NRAM simulation of arbitrary RAM programs: **non-uniform** word size model)**.** *Any N-word RAM with a word size of $W = \Omega(\log^2 N)$ bits can be simulated by an Oblivious NRAM (with non-uniform word sizes) that consumes $O(W)$ bits of CPU cache, and with $O(M)$ memory banks each of $O(W \cdot (M + N/M + N^\delta))$ bits in size. Further, the oblivious NRAM simulation incurs $\widehat{O}(1)$ bandwidth blowup and $\widehat{O}(\log N)$ run-time blowup.*

Note that for the non-uniform version of the theorem, we state the memory bank and cache sizes in terms of *bits* instead of *words* to avoid confusion. In both the uniform and non-uniform versions of the theorem, an interesting point of parametrization is when $M = O(\sqrt{N})$, and each bank is $O(W\sqrt{N})$ *bits* in size.

3.3 Obliviousness Proof

We first argue that the base scheme described in Section 3.1 is oblivious by Definition 2– in fact its obliviousness proof is trivial. Each data access will perform a ReadBank operation on a random bank. The choice of this bank is determined by position[vaddr] whose value was chosen independently at random earlier, and the value has not been observed by the adversary. Besides the ReadBank operation, each memory access also performs eviction ν times. Since eviction simply scans through the banks and performs WriteBank operations on them, it is clearly oblivious. Note that each ReadBank (or WriteBank) operation is in fact a hash table lookup (or hash table write). For each hash table operation, the number of memory accesses is always padded to a maximum value to ensure obliviousness. In particular, we use a hash table with $\widehat{O}(1)$ worst-cast cost per operation as mentioned in Theorem 1. Next, it is not hard to see that the recursion does not break the obliviousness, since in a recursive ONRAM, each memory access would simply make a lookup in each ONRAM_i from $i = d$ (smallest metadata ONRAM) to $i = 0$ (top-level data ONRAM).

Next, for the techniques described in Section 3.2 to reduce the CPU cache to $O(1)$, the key difference is to use a scratch bank to implement the hash table and the eviction queues. From

Figure 1, it is clear that when and what operations are made to the scratch bank are deterministic and independent of any secrets. Further, for the eviction queues and the hash table, each operation will always be padded to the maximum number of accesses (to the scratch bank).

Based on the above arguments, it is easy to construct a simulator which, without knowing the inputs to the program, simulates the "observable trace". The simulated trace has identical distribution as the real trace.

3.4 Extension: Oblivious Data Structures in the NRAM Model

Since our O-NRAM construction is a recursion-based construction, we can apply similar techniques as Wang *et al.* [46] to efficiently realize a class of oblivious data structures in the NRAM model.

Theorem 4. *Oblivious* `stack`, `queue`, `priority_queue`, `map` *and* `set` *can be realized in the NRAM model with only* $\widehat{O}(1)$ *bandwidth blowup and runtime blowup, assuming* $O(1)$ *words of CPU cache, and* $O(M)$ *memory banks each of* $O(M + N/M + N^\delta)$ *words in size. The above holds as long as each entry in the data structure contains* $\Omega(\log N)$ *bits.*

Proof. Observe that the oblivious data structure techniques by Wang *et al.* [46] are directly applicable based on our generic O-NRAM construction which is recursion based. Wang *et al.* use a $O(\log N)$-sized CPU cache. Here this CPU cache can be stored in a scratch bank with $\omega(1)$ standard hash tables each N^δ words in size. □

4 Sequential Oblivious Simulation of Parallel Programs

We are eventually interested in parallel oblivious simulation of parallel programs (see Section 5). As a stepping stone, we first consider sequential oblivious simulation of parallel programs. However, we emphasize that the results in this section can be of independent interest in their own right. In particular, one way to interpret these results is that "parallelism facilitates obliviousness". Specifically, if a program exhibits a sufficient degree of parallelism, then this program can be made oblivious at only const overhead in the Network RAM model. The intuition for why this is so, is that instructions in each parallel time step can be executed in any order. Since subsequences of instructions can be executed in an arbitrary order during the simulation, many sequences of memory requests can be mapped to the same access pattern, and thus the request sequence is partially obfuscated.

4.1 Parallel RAM

To formally characterize what it means for a program to exhibit a sufficient degree of parallelism, we will formally define a P-parallel RAM. In this section, the reader should think of parallelism as a property of the program to be simulated – we actually characterize costs assuming both the non-oblivious and the oblivious programs are executed on a sequential machine (different from Section 5).

An P-parallel RAM machine is the same as a RAM machine, except the next instruction function outputs P instructions which can be executed in parallel.

Definition 4 (P-parallel RAM)**.** *An* P-Parallel RAM *is a RAM which has a next instruction function* $\Pi = \Pi_1, \ldots, \Pi_P$ *such that on input* $(\mathsf{state} = \mathsf{state}_1 || \cdots || \mathsf{state}_P, \mathsf{rdata} = \mathsf{rdata}_1 || \cdots || \mathsf{rdata}_P)$, Π *outputs* P *instructions* $(I_1, \ldots, I_P)$ *and* P *updated states* $\mathsf{state}'_1, \ldots, \mathsf{state}'_P$ *such that for* $p \in [P]$, $(I_p, \mathsf{state}'_p) = \Pi_p(\mathsf{state}_p, \mathsf{rdata}_p)$. *The instructions* $I_1, \ldots, I_P$ *satisfy one of the following:*

- *All of* $I_1, \ldots, I_P$ *are set to* (stop, z) *(with the same* z*).*
- *All of* $I_1, \ldots, I_P$ *are either of the form.* $(\mathsf{read}, \mathsf{vaddr}, \perp)$ *or* $(\mathsf{write}, \mathsf{vaddr}, \mathsf{wdata})$*.*

Finally, the state state *has size at most* $O(P)$*.*

In general, we consider the standard Concurrent Read, Concurrent Write (CRCW) PRAM, where CPUs may read the same data simultaneously, and simultaneous write conflicts are resolved in a canonical way.

As a warmup exercise, we will first consider a special case where in each parallel step, the memory requests made by each processor has distinct addresses—we refer to this model as a *restricted* PRAM. Later in Section 4.3, we will extend the result to the general CRCW PRAM case.

Definition 5 (Restricted P-parallel RAM)**.** *For a* P*-parallel RAM denoted* $\mathsf{PRAM} := \langle D, \mathsf{state}_1, \ldots, \mathsf{state}_P, \Pi_1, \ldots \Pi_P \rangle$*, if every batch of instructions* $I_1, \ldots, I_P$ *have unique* vaddr*'s, we say that* PRAM *is a* restricted P*-parallel RAM.*

4.2 Warmup: Restricted Parallel RAM to Oblivious NRAM

Our goal is to compile any P-parallel RAM (not necessarily restricted), into an efficient O-NRAM. As an intermediate step that facilitates presentation, we begin with a basic construction of O-NRAM from any *restricted*, parallel RAM. In the following section, we extend to a construction of O-NRAM from any parallel RAM (not necessarily restricted).

Let $\mathsf{PRAM} := \langle D, \mathsf{state}_1, \ldots, \mathsf{state}_P, \Pi_1, \ldots \Pi_P \rangle$ be a restricted P-Parallel RAM, for $P = \omega(M \log N)$. We now present an O-NRAM simulation of PRAM that requires $M + 1$ memory banks, each with $O(N/M + P)$ physical memory, where N is the database size.

Setup: Pseudorandomly assign memory words to banks. The setup phase takes the initial states of the PRAM, including the memory array D and the initial CPU state, and compiles them into the initial states of the Oblivious NRAM denoted ONRAM.

To do this, the setup algorithm chooses a secret key K, and sets $\mathsf{ONRAM.state} = \mathsf{PRAM.state} || K$. Each memory bank of ONRAM will be initialized as a Cuckoo hash table. Each memory word in the PRAM's initial memory array D will be inserted into the bank numbered $(\mathsf{PRF}_K(\mathsf{vaddr}) \bmod M) + 1$, where vaddr is the virtual address of the word in PRAM. Note that the ONRAM's $(M+1)$-th memory bank is reserved as a scratch bank whose usage will become clear later.

doNext(): *//We only consider* read *and* write *instructions here but not* stop*.*

1: **For** $p := 1$ to P: $(\mathsf{op}_p, \mathsf{vaddr}_p, \mathsf{wdata}_p) := \Pi_p(\mathsf{state}_p, \mathsf{rdata}_p)$

2: $(\mathsf{rdata}_1, \mathsf{rdata}_2, \ldots, \mathsf{rdata}_p) := \mathsf{Access}\left(\{\mathsf{op}_p, \mathsf{vaddr}_p, \mathsf{wdata}_p\}_{p \in [P]}\right)$

Figure 3: **Oblivious simulation of each step of the restricted parallel RAM's execution**.

Simulating each step of the PRAM's execution. Each doNext() operation of the PRAM will be compiled into a sequence of instructions of the ONRAM. We now describe how this compilation works. Our presentation focuses on the case when the next instruction's op-codes are reads or writes. Wait or stop instructions are left unmodified during the compilation.

Access $\left(\{\mathsf{op}_p, \mathsf{vaddr}_p, \mathsf{wdata}_p\}_{p \in P}\right)$:

1: **for** $p = 1$ to P **do**
2: $\quad m \leftarrow \mathsf{PRF}_K(\mathsf{vaddr}_p)$;
3: $\quad \mathsf{queue}[m] := \mathsf{queue}[m].\mathsf{push}(p, \mathsf{op}_p, \mathsf{vaddr}_p, \mathsf{wdata}_p)$;
$\quad$ // queue *is stored in a separate scratch bank.*
4: **end for**
5: **for** $m = 1$ to M **do**
6: $\quad$ **if** $|\mathsf{queue}[m]| > \mathsf{max}$ **then abort**
7: $\quad$ Pad $\mathsf{queue}[m]$ with dummy entries $(\perp, \perp, \perp, \perp)$ so that its size is max;
8: $\quad$ **for** $i = 1$ to max **do**
9: $\quad\quad (p, \mathsf{op}, \mathsf{vaddr}, \mathsf{wdata}) := \mathsf{queue}[m].\mathsf{pop}()$
10: $\quad\quad \mathsf{rdata}_p := \mathsf{ReadBank}(m, \mathsf{vaddr})$
$\quad\quad$ // *Each bank is a deamortized Cuckoo hash table.*
11: $\quad\quad$ **if** op = write **then** $\mathsf{wdata} := \mathsf{rdata}_p$
12: $\quad\quad \mathsf{WriteBank}(m, \mathsf{vaddr}, \mathsf{wdata})$
13: $\quad$ **end for**
14: **end for**
15: **return** $(\mathsf{rdata}_1, \mathsf{rdata}_2, \ldots, \mathsf{rdata}_P)$

Figure 4: **Obliviously serving a batch of P memory requests with distinct virtual addresses.**

As shown in Figure 3, for each doNext instruction, we first compute the batch of instructions $I_1, \ldots, I_P$, by evaluating the P parallel next-instruction circuits $\Pi_1, \ldots, \Pi_P$. This results in P parallel read or write memory operations. This batch of P memory operations (whose memory addresses are guaranteed to be distinct in the restricted parallel RAM model) will then be served using the subroutine Access.

We now elaborate on the the Access subroutine. Each batch will have $P = \omega(M \log N)$ memory operations whose virtual addresses are distinct. Since each virtual address is randomly assigned to one of the M banks, in expectation, each bank will get $P/M = \omega(\log N)$ hits. Using a balls and bins analysis, we show that the number of hits for each bank is highly concentrated around the expectation. In fact, the probability of any $\omega(\log N)$ deviation from the expectation is negligible in N. Therefore, we will choose an appropriate $\mathsf{max} := O(P/M) + f(N)$, for any $f(N) = \omega(\log N)$ for each bank, and make precisely max number of accesses to each memory bank. Specifically, the Access algorithm first scans through the batch of $P = \omega(M \log N)$ memory operations, and assigns them to M queues, where the m-th queue stores requests assigned to the m-th memory bank. Then, the Access algorithm sequentially serves the requests to memory banks $1, 2, \ldots, M$ in a linear order, padding the number of accesses to each bank to max. This way, the access patterns to the banks are guaranteed to be oblivious.

The description of Figure 4 makes use of M queues with a total size of $P = \omega(M \log N)$ words. It is not hard to see that these queues can be stored in an additional scratch bank of size $O(P)$, incurring only constant number of accesses to the scratch bank per queue operation. Further, in Figure 4, the time at which the queues are accessed, and the number of times they are accessed are not dependent on input data (notice that Line 7 can be done by linearly scanning through each

queue, incurring a max cost each queue).

Cost analysis. Since $\mathsf{max} = O(P/M) + f(N)$, for any $f(N) = \omega(\log N)$ in Figure 4 (see Theorem 5), it is not hard to see each batch of $P = \omega(M \log N)$ memory operations will incur $\Theta(P)$ accesses to data banks in total, and $\Theta(P)$ accesses to the scratch bank. Therefore, the ONRAM incurs only a constant factor more total work and bandwidth than the underlying PRAM.

Theorem 5. *Let* PRF *be a family of pseudorandom functions, and* PRAM *be a restricted* P*-Parallel RAM for* $P = \omega(M \log N)$*. Let* $\mathsf{max} := O(P/M) + f(N)$*, for any* $f(N) = \omega(\log N)$*, in Figure 4. Then, the construction described above is an oblivious simulation of* PRAM *using* M *banks each of* $O(N/M + P)$ *words in size. Moreover, the oblivious simulation performs total work that is constant factor larger than that of the underlying* PRAM*.*

Proof. Assuming the execution never aborts (Line 6 in Figure 4), then Theorem 5 follows immediately, since the access pattern is deterministic and independent of the inputs. Therefore, it suffices to show that the abort happens with negligible probability on Line 6. This is shown in the following lemma. □

Lemma 2. *Let* $\mathsf{max} := O(P/M) + f(N)$*, for any* $f(N) = \omega(\log N)$ *with appropriately chosen constants. For any* PRAM *and any input* x*, abort on Line 6 of Figure 4 occurs only with negligible probability (over choice of the* PRF*).*

Proof. We first replace PRF with a truly random function f. Note that if we can prove the lemma for a truly random function, then the same should hold for PRF, since otherwise we obtain an adversary breaking pseudorandomness.

We argue that the probability that abort occurs on Line 6 of Figure 4 in a particular step i of the execution is negligible. By taking a union bound over the (polynomial number of) steps of the execution, the lemma follows.

To upper bound the probability of abort in some step i, consider a thought experiment where we change the order of sampling the random variables: We precompute all the PRAM's instructions up to and including the i-th step of the execution (independently of f), obtaining P distinct virtual addresses, and only then choose the outputs of the random function f on the fly. That is, when each virtual memory address vaddr_p in step i is serviced, we choose $m := f(\mathsf{vaddr}_p)$ uniformly and independently at random. Thus, in step i of the execution, there are P distinct virtual addresses (i.e., balls) to be thrown into M memory banks (i.e., bins). Due to standard Chernoff bounds, for $P = \omega(M \log N)$, the probability that there exists a bin whose load exceeds $O(P/M) + f(N)$, for any $f(N) = \omega(\log N)$ is $N^{-\omega(1)}$, which is negligible in N. □

4.3 Parallel RAM to Oblivious NRAM

Use a hash table to suppress duplicates. In Section 4.2, we describe how to obliviously simulate a restricted parallel-RAM in the NRAM model. We now generalize this result to support any P-parallel RAM, not necessarily restricted ones. The difference is that for a generic P-parallel RAM, each batch of P memory operations generated by the next-instruction circuit need not have distinct virtual addresses. For simplicity, imagine that the entire batch of memory operations are reads. In the extreme case, if all $P = \omega(M \log N)$ operations correspond to the same virtual address residing in bank m, then the CPU should not read bank m as many as P number of times. To

Access $\left(\{\mathsf{op}_p, \mathsf{vaddr}_p, \mathsf{wdata}_p, p\}_{p \in P}\right)$:

/* HTable, queue, *and* result *data structures are stored in a scratch bank. For obliviousness, operations on these data structures must be padded to the worst-case cost as we elaborate in the text.* */

1: **for** $p = 1$ to P: $\mathsf{HTable}[\mathsf{op}_p, \mathsf{vaddr}_p] := (\mathsf{wdata}_p, p)$ // *hash table insertions*
2: **for** $\{(\mathsf{op}, \mathsf{vaddr}), \mathsf{wdata}, p\} \in \mathsf{HTable}$ **do** // *iterate through hash table*
3: $m := \mathsf{PRF}_K(\mathsf{vaddr})$
4: $\mathsf{queue}[m] := \mathsf{queue}[m].\mathsf{push}(\mathsf{op}, \mathsf{vaddr}, \mathsf{wdata})$;
5: **end for**
6: **for** $m = 1$ to M **do**
7: **if** $|\mathsf{queue}[m]| > \mathsf{max}$ **then abort**
8: Pad $\mathsf{queue}[m]$ with dummy entries $(\perp, \perp, \perp)$ so that its size is max;
9: **for** $i = 1$ to max **do**
10: $(\mathsf{op}, \mathsf{vaddr}, \mathsf{wdata}, p) := \mathsf{queue}[m].\mathsf{pop}()$
11: $\mathsf{result}[p] := \mathsf{ReadBank}(m, \mathsf{vaddr})$
12: **if** op = write **then** wdata := rdata
13: $\mathsf{WriteBank}(m, \mathsf{vaddr}, \mathsf{wdata})$
14: **end for**
15: **end for**
16: **return** $(\mathsf{result}[1], \ldots, \mathsf{result}[p])$ // *hash table lookups*

Figure 5: **Obliviously serving a batch of P memory request, not necessarily with distinct virtual addresses.**

address this issue, we rely on an additional Cuckoo hash table [37] denoted HTable to suppress the duplicate requests (see Figure 5, and the doNext function is defined the same way as Section 4.2).

The HTable will be stored in the scratch bank. We can employ a standard Cuckoo hash table that need not be deamortized. As shown in Figure 5, we need to support hash table insertions, lookups, and moreover, we need to be able to iterate through the hash table. We now make a few remarks important for ensuring obliviousness. Line 1 of Figure 5 performs $P = \omega(M \log N)$ number of insertions into the Cuckoo hash table. Due to standard Cuckoo hash analysis, we know that these insertions will take $O(P) + f(N)$, for any $f(N) = \omega(\log N)$ total time except with negligible probability. Therefore, to execute Line 1 obliviously, we simply need to pad with dummy insertions up to some $\mathsf{max}' = O(P) + f(N)$, for any $f(N) = \omega(\log N)$.

Next, we describe how to execute the loop at Line 2 obliviously. The total size of the Cuckoo hash table is $O(P)$. To iterate over the hash table, we simply make a linear scan through the hash table. Some entries will correspond to dummy elements. When iterating over these dummy elements, we simply perform dummy operations for the **for** loop. Finally, observe that Line 16 performs lookups to the Cuckoo hash table, and each hash table lookup requires worst-case $O(1)$ accesses to the scratch bank.

Cost analysis. Since $\mathsf{max} = O(P/M) + f(N)$, for any $f(N) = \omega(\log N)$ in Figure 5 (see Theorem 5), it is not hard to see each batch of $P = \omega(M \log N)$ memory operations will incur $O(P)$ accesses to data banks in total, and $O(P)$ accesses to the scratch bank. Note that this takes into account the fact that Line 1 and the for-loop starting at Line 2 are padded with dummy accesses. Therefore, the ONRAM incurs only a constant factor more total work and bandwidth than the

underlying PRAM.

Theorem 6. *Let* $\mathsf{max} = O(P/M) + f(N)$*, for any* $f(N) = \omega(\log N)$ *for some appropriate choice of constants. Assume that* PRF *is a secure pseudorandom function, and* PRAM *is a* P*-Parallel RAM for* $P = \omega(M \log N)$*. Then, the above construction obliviously simulates* PRAM *in the NRAM model, incurring only a constant factor blowup in total work and bandwidth consumption.*

Proof. (sketch.) Similar to the proof of Theorem 5, except that now we have the additional hash table. Note that obliviousness still holds, since, as discussed above, each batch of P memory requests requires $O(P)$ accesses to the scratch bank, and this can be padded with dummy accesses to ensure the number of scratch bank accesses remains the same in each execution. □

5 Parallel Oblivious Simulation of Parallel Programs

In the previous section, we considered sequential oblivious simulation of programs that exhibit parallelism – there, we considered parallelism as being a property of the program which will actually be executed on a sequential machine. In this section we consider *parallel* and oblivious simulations of parallel programs. Here, the programs will actually be executed on a parallel machine, and we consider classical metrics such as parallel runtime and total work as in the parallel algorithms literature.

We introduce the *Network PRAM* model – informally, this is a Network RAM with parallel processing capability. Our goal in this section will be to compile a PRAM into an Oblivious Network PRAM (O-NPRAM), *a.k.a.,* the "parallel-to-parallel compiler".

Our O-NPRAM is the Network RAM analog of the Oblivious Parallel RAM (OPRAM) model by Boyle *et al.* [7]. Goldreich and Ostrovsky's logarithmic ORAM lower bound (in the sequential execution model) directly implies the following lower bound for standard OPRAM [7]: Let PRAM be an arbitrary PRAM with P processors running in parallel time t. Then, any P-parallel OPRAM simulating PRAM must incur $\Omega(t \log N)$ parallel time. Clearly, OPRAM would also work in our Network RAM model albeit not the most efficient, since it is not exploiting the fact that the addresses in each bank are inherently oblivious. In this section, we show how to perform oblivious parallel simulation of "sufficiently parallel" programs in the Network RAM model, incurring only $O(\log^* N)$ blowup in parallel runtime, and achieving optimal total work. Our techniques make use of fascinating results in the parallel algorithms literature [4, 5, 24].

5.1 Network PRAM (NPRAM) Definitions

Similar to our NRAM definition, an NPRAM is much the same as a standard PRAM, except that 1) memory is distributed across multiple banks, $\mathsf{Bank}_1, \ldots, \mathsf{Bank}_M$; and 2) every virtual address vaddr can be written in the format $\mathsf{vaddr} := (m, \mathit{offset})$, where m is the bank identifier, and *offset* is the offset within the Bank_m. We use the notation P-parallel NPRAM to denote an NPRAM with P parallel processors, each with $O(1)$ words of cache. If processors are initialized with secret randomness unobservable to the adversary, we refer to this as a probablistic NPRAM.

Observable traces. In the NPRAM model, we assume that an adversary can observe the following parts of the memory trace: 1) which processor is making the request; 2) whether this is a read or write request; and 3) which bank the request is going to. The adversary is unable to observe the offset within a memory bank.

Definition 6 (Observable traces for NPRAM). *For a probabilistic P-parallel* NPRAM*, we use the notation* $\mathsf{Tr}_\rho(\mathsf{NPRAM}, x)$ *to denote its observable traces upon input x, and initial CPU randomness ρ (collective randomness over all processors):*

$$\mathsf{Tr}_\rho(\mathsf{NPRAM}, x) := \left[\left((\mathsf{op}_1^1, m_1^1), \ldots, (\mathsf{op}_1^P, m_1^P)\right), \ldots, \left((\mathsf{op}_T^1, m_T^1), \ldots, (\mathsf{op}_T^P, m_T^P)\right)\right]$$

where T is the total parallel execution time of the NPRAM*, and $\{(\mathsf{op}_i^1, m_i^1), \ldots, (\mathsf{op}_i^P, m_i^P)\}$ is of the op-codes and memory bank identifiers for each processor during parallel step $i \in [T]$ of the execution.*

Based on the above notion of observable memory trace, an Oblivious NPRAM can be defined in a similar manner as the notion of O-NRAM (Definition 2).

Metrics. We consider classical metrics adopted in the vast literature on parallel algorithms, namely, the parallel runtime and the total work. In particular, to characterize the oblivious simulation overhead, we will consider

- **Parallel runtime blowup.** The blowup of the parallel runtime comparing the O-NPRAM and the NPRAM.
- **Total work blowup.** The blowup of the total work comparing the O-NPRAM and the NPRAM. If the total work blowup is $O(1)$, we say that the O-NPRAM achieves *optimal* total work.

5.2 Construction of Oblivious Network PRAM

Preliminary: colored compaction. The colored compaction problem [4] is the following:

> Given n objects of m different colors, initially placed in a single source array, move the objects to m different destination arrays, one for each color. In this paper, we assume that the *space for the m destination arrays are preallocated*. We use the notation d_i to denote the number of objects colored i for $i \in [m]$.

Lemma 3 (Log*-time parallel algorithm for colored compaction [4]). *There is a constant $\epsilon > 0$ such that for all given n, m, τ, $d_1, \ldots, d_m \in \mathbb{N}$, with $m = O(n^{1-\delta})$ for arbitrary fixed $\delta > 0$, and $\tau \geq \log^* n$, there exists a parallel algorithm for the colored compaction problem (assuming preallocated destination arrays) with n objects, m colors, and $d_1, \ldots, d_m$ number of objects for each color, executing in $O(\tau)$ time on $\lceil n/\tau \rceil$ processors, consuming $O(n + \sum_{i=1}^m d_i)$ space, and succeeding with probability at least $1 - 2^{-n^\epsilon}$.*

Preliminary: parallel static hashing. We will also rely on a parallel, static hashing algorithm [5, 24], by Bast and Hagerup. The static parallel hashing problem takes n elements (possibly with duplicates), and in parallel creates a hash table of size $O(n)$ of these elements, such that later each element can be visited in $O(1)$ time. In our setting, we rely on the parallel hashing to suppress duplicate memory requests. Bast and Hagerup show the following lemma:

Lemma 4 (Log*-time parallel static hashing [5, 24]). *There is a constant $\epsilon > 0$ such that for all $\tau \geq \log^* n$, there is a parallel, static hashing algorithm, such that hashing n elements (which*

$\underline{\mathsf{parAccess}}\left(\{\mathsf{op}_p, \mathsf{vaddr}_p, \mathsf{wdata}_p\}_{p\in P}\right)$:

/ All steps can be executed in $O(\log^* P)$ time with $P' = O(P/\log^* P)$ processors with all except negligible probability.*/*

1: Using the scratch bank as memory, run the parallel hashing algorithm on the batch of $P = M^{1+\delta}$ memory requests to suppress duplicate addresses. Denote the resulting set as S, and pad S with dummy requests to the maximum length P.

2: In parallel, assign colors to each memory request in the array S. For each real memory access $\{\mathsf{op}, \mathsf{vaddr}, \mathsf{wdata}\}$, its color is defined as $\mathsf{PRF}_K(\mathsf{vaddr})$. Each dummy memory access is assigned a random color. It is not hard to see that each color has no more than $\mathsf{max} := P/M + f(N)$, for any $f(N) = \omega(\log N)$ requests except with negligible probability.

3: Using the scratch bank as memory, run the parallel colored compaction algorithm to assign the array S to M preallocated queues each of size max (residing in the scratch bank).

4: Now, each queue $i \in [M]$ contains max number of requests intended for bank i – some real, some dummy. Serve all memory requests in the M queues in parallel. Each processor $i \in [P']$ is assigned the k-th memory request iff $(k \mod P') = i$. Dummy requests incur accesses to the corresponding banks as well.
For each request coming from processor p, the result of the fetch is stored in an array $\mathsf{result}[p]$ in the scratch bank.

Figure 6: **Obliviously serving a batch of P memory requests using $P' := O(P/\log^* P)$ processors in $O(\log^* P)$ time.** In Steps 1, 2, and 3, each processor will make exactly one access to the scratch bank in each parallel execution step – *even if the processor is idle in this step, it makes a dummy access to the scratch bank.* Steps 1 through 3 are always padded to the worst-case parallel runtime.

need not be distinct) can be done in $O(\tau)$ parallel time, with $O(n/\tau)$ processors and $O(n)$ space, succeeding with $1 - 2^{-(\log n)^{\tau/\log^* n}} - 2^{-n^\epsilon}$ probability.

Construction. We now present a construction that allows us to compile a P-parallel PRAM, where $P = M^{1+\delta}$ for any constant $\delta > 0$, into a $O(P/\log^* P)$-parallel Oblivious NPRAM. The resulting NPRAM has $O(\log^* P)$ blowup in parallel runtime, and is optimal in total amount of work.

In the original P-parallel PRAM, each of the P processors does constant amount of work in each step. In the oblivious simulation, this can trivially be simulated in $O(\log^* P)$ time with $O(P/\log^* P)$ processors. Therefore, clearly the key is how to obliviously fetch a batch of P memory accesses in parallel with $O(P/\log^* P)$ processors, and $O(\log^* P)$ time. We describe such an algorithm in Figure 6. Using a scratch bank as working memory, we first call the parallel hashing algorithm to suppress duplicate memory requests. Next, we call the parallel colored compaction algorithm to assign memory request to their respective queues – depending on the destination memory bank. Finally, we make these memory accesses, including dummy ones, in parallel.

Theorem 7. *Let* PRF *be a secure pseudorandom function, let $M = N^\epsilon$ for any constant $\epsilon > 0$. Let* PRAM *be a P-parallel RAM for $P = M^{1+\delta}$, for constant $\delta > 0$. Then, there exists an Oblivious NPRAM simulation of* PRAM *with the following properties:*

- *The Oblivious NPRAM consumes M banks each of which $O(N/M + P)$ words in size.*

- *If the underlying* PRAM *executes in t parallel steps, then the Oblivious NPRAM executes in $O(t \log^* P)$ parallel steps utilizing $O(P/\log^* P)$ processors. We also say that the NPRAM has $O(\log^* P)$ blowup in parallel runtime.*
- *The total work of the Oblivious NPRAM is asymptotically the same as the underlying* PRAM.

Proof. We now prove security and costs separately.

Security proof. Observe that Steps 1, 2, and 3 in Figure 6 make accesses only to the scratch bank. We make sure that each processor will make exactly one access to the scratch bank in every parallel execution step – even if the processor is idle in this step, it will make a dummy access. Further, Steps 1 through 3 are also padded to the worst-case running time. Therefore, the observable memory traces of Steps 1 through 3 are perfectly simulatable without knowing secret inputs.

For Step 4 of the algorithm, since each of the M queues are of fixed length max, and each element is assigned to each processor in a round-robin manner, the bank number each processor will access is clearly independent of any secret inputs, and can be perfectly simulated (recall that dummy request incur accesses to the corresponding banks as well).

Costs. First, due to Lemma 2, each of the M queues will get $P/M + f(N)$, for any $f(N) = \omega(\log N)$ memory requests with probability $1 - \text{negl}(N)$. This part of the argument is the same as Section 4. Now, observe that the parallel runtime for Steps 2 and 4 are clearly $O(\log^* P)$ with $O(P/\log^* P)$ processors. Based on Lemmas 4 and 3, Steps 1 and 3 can be executed with a worst-case time of $O(\log^* P)$ on $O(P/\log^* P)$ processors as well. We note that the conditions $M = N^\epsilon$ and $P = M^{1+\delta}$ ensure $\text{negl}(N)$ failure probability. □

6 Conclusion

We define a new model for oblivious execution of programs, where an adversary cannot observe the memory offset within each memory bank, but can observe the patterns of communication between the CPU(s) and the memory banks. Under this model, we demonstrate novel *sequential* and *parallel* algorithms that exploit the "free obliviousness" within each bank, and asymptotically lower the cost of oblivious data accesses in comparison with the traditional ORAM [19] and OPRAM [7]. In the process, we propose novel algorithmic techniques that "leverage parallelism for obliviousness". These techniques have not been used in the standard ORAM or OPRAM line of work, and demonstrate interesting connections to the fundamental parallel algorithms literature.

Acknowledgments

This research was funded in part by an NSF grant CNS-1314857, a subcontract from the DARPA PROCEED program, a Sloan Research Fellowship, and Google Faculty Research Awards. The views and conclusions contained herein are those of the authors and should not be interpreted as representing funding agencies.

References

[1] Intel SGX for dummies (intel SGX design objectives). `https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx`.

Table 2: Notations.

Notation	Meaning
N	total number of memory words
P	number of processors in the PRAM
M	number of memory banks in the NRAM
W	number of bits in each memory word
Π (or Π_p)	next instruction circuit (for processor p)
vaddr (or vaddr_p)	virtual address of a memory word (read or written by processor p)
state (or state_p)	CPU state (for processor p)
rdata (or rdata_p)	a memory word fetched from memory (for processor p)
wdata (or wdata_p)	a memory word to be written to memory (for processor p)
$\mathsf{Bank}_m[\mathit{offset}]$	the memory word at physical *offset* in the m-th bank

[2] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Automata, Languages and Programming, 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I*, pages 107–118, 2009.

[3] Sumeet Bajaj and Radu Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Trans. Knowl. Data Eng.*, 26(3):752–765, 2014.

[4] Hannah Bast and Torben Hagerup. Fast parallel space allocation, estimation, and integer sorting. *Inf. Comput.*, 123(1):72–110, November 1995.

[5] Holger Bast and Torben Hagerup. Fast and reliable parallel hashing. In *SPAA*, pages 50–61, 1991.

[6] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. `http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf`, 2011.

[7] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram. `https://eprint.iacr.org/2014/594.pdf`.

[8] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. *CoRR*, abs/1307.3699, 2013.

[9] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[10] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, and Srinivas Devadas. Tiny ORAM: A low-latency, low-area hardware oram controller with integrity verification.

[11] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW Path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[12] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.

[13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. In *Advances in Cryptology - EUROCRYPT 2014*, volume 8441, pages 405–422. 2014.

[15] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Garbled ram revisited, part i. Cryptology ePrint Archive, Report 2014/082, 2014. `http://eprint.iacr.org/`.

[16] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private ram computation. *IACR Cryptology ePrint Archive*, 2014:148, 2014.

[17] Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *32nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 698–710, 1991.

[18] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[19] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[20] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[21] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2012.

[22] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[23] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM CCS*, 2012.

[24] Torben Hagerup. The log-star revolution. In *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, pages 259–278, 1992.

[25] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. In *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, pages 611–622, 2008.

[26] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[27] Chang Liu, Michael Hicks, Austin Harris, Mohit Tiwari, Martin Maas, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. Manuscript.

[28] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 51–65, 2013.

[29] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient ram-model secure computation. In *IEEE S & P*. IEEE Computer Society, 2014.

[30] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography Conference (TCC)*, 2013.

[31] Steve Lu and Rafail Ostrovsky. How to garble ram programs. In *EUROCRYPT*, pages 719–734, 2013.

[32] Steve Lu and Rafail Ostrovsky. Garbled ram revisited, part ii. Cryptology ePrint Archive, Report 2014/083, 2014. `http://eprint.iacr.org/`.

[33] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[34] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. A high-performance oblivious RAM controller on the convey hc-2ex heterogeneous computing platform. In *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)*, 2013.

[35] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted granularity of parallel memories. *Acta Inf.*, 21:339–374, 1984.

[36] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *ACM Symposium on Theory of Computing (STOC)*, 1997.

[37] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[38] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, pages 571–582, 2013.

[39] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[40] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[41] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy (S & P)*, 2013.

[42] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[43] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious ram protocol. In *ACM CCS*, 2013.

[44] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *International conference on Supercomputing*, ICS '03, pages 160–171, 2003.

[45] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *SIGOPS Oper. Syst. Rev.*, 34(5):168–177, November 2000.

[46] Xiao Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *ACM CCS*, 2014.

[47] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the goldreich-ostrovksy lower bound. `http://eprint.iacr.org/2014/672.pdf`.

[48] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, abhi shelat, and Elaine Shi. Scoram: Oblivious ram for secure computation. `http://eprint.iacr.org/2014/671.pdf`.

[49] Peter Williams and Radu Sion. Usable PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[50] Peter Williams and Radu Sion. SR-ORAM: Single round-trip oblivious ram. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[51] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.

[52] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In

CCS, 2012.

A Analysis of Deamortized Cuckoo Hash Table

We first describe our modification of the Cuckoo hash table of Arbitman *et al.* [2]. Throughout this section, we follow [2] nearly verbatim.

Our data structure uses two tables T_0 and T_1, and two auxiliary data structures: a queue, and a cycle-detection mechanism. Each table consists of $r = (1+\epsilon)n$ entries for some small constant $\epsilon > 0$. Elements are inserted into the tables using two hash functions $h_0, h_1 : \mathcal{U} \to 0, ..., r-1$, which are independently chosen at the initialization phase. We assume that the auxiliary data structures satisfy the following properties:

1. The queue is constructed to store $g(N)$, for any $g(N) = \omega(\log N)$, number of elements at any point in time. It should support the operations Lookup, Delete, PushBack, PushFront, and PopFront in worst-case constant time (with overwhelming probability over the randomness of its initialization phase).

2. The cycle-detection mechanism is constructed to store $g(N)$, for any $g(N) = \omega(\log N)$, elements at any point in time. It should support the operations Lookup, Insert and Reset in worst-case $\widehat{O}(\log N)$ time (with all but negligible probability over the randomness of its initialization phase).

An element $x \in \mathcal{U}$ can be stored in exactly one out of three possible places: entry $h_0(x)$ of table T_0, entry $h_1(x)$ of table T_1, or the queue. The lookup procedure is straightforward: when given an element $x \in \mathcal{U}$, query the two tables and if needed, perform lookups in the queue. The deletion procedure is also straightforward by first searching for the element, and then deleting it. Our insertion procedure is parameterized by a value $L = L(N)$, for any $L(N) \in \omega(1)$, and is defined as follows. Given a new element $x \in \mathcal{U}$, we place the pair $(x, 0)$ at the back of the queue (the additional bit 0 indicates that the element should be inserted to table T_0). Then, we take the pair at the head of the queue, denoted (y, b), and place y in entry $T_b[h_b(y)]$. If this entry is not occupied, we again take the pair that is currently stored at the head of the queue, and repeat the same process. If the entry $T_b[h_b(y)]$ is occupied, however, we place its previous occupant z in entry $T_{1-b}[h_{1-b}(z)]$ and so on, as in the above description of cuckoo hashing. After L elements have been moved, we place the current nestless element at the head of the queue, together with a bit indicating the next table to which it should be inserted, and terminate the insertion procedure.

We next restate Theorem 1:

Theorem 8 (Deamortized Cuckoo hash table: negligible failure probability version)**.** *There is an implementation of the above deamortized Cuckoo hash table of capacity s such that with probability $1 - \mathrm{negl}(N)$, each insertion, deletion, and lookup operation is performed in worst-case $\widehat{O}(1)$ time (not counting the cost of operating on the queue) – as long as at any point in time at most s elements are stored in the data structure. The above deamortized Cuckoo hash table consumes $O(s) + O(N^\delta)$ space where $0 < \delta < 1$ is a constant.*

In the following, we describe the instantiation of the auxiliary data structures.

The queue. We will argue that with overwhelming probability the queue contains at most $g(N)$, for any $g(N) \in \omega(\log N)$, elements at any point in time. Therefore, we design the queue to store at

most $g(N)$ elements, and allow the whole data structure to fail if the queue overflows. Although a classical queue can support the operations PushBack, PushHead, and PopFront in constant time, we also need to support the operations Lookup and Delete in constant time. One possible instantiation is to use k, for any $k \in \omega(1)$, arrays $A_1, ..., A_k$ each of size N^δ, for some $\delta < 1$. Each entry of these arrays consists of a data element, a pointer to the previous element in the queue, and a pointer to the next element in the queue. In addition we maintain two global pointers: the first points to the head of the queue, and the second points to the end of the queue. The elements are stored using a function h chosen from a collection of pairwise independent hash functions. Specifically, each element x is stored in the first available entry amongst $\{A_1[h(1,x)], ..., A_k[h(k,x)]\}$. For any element x, the probability that all of its k possible entries are occupied when the queue contains at most $g(N)$ elements is upper bounded by $(g(N)/N^\delta)^k$, which can be made negligible by choosing an appropriate k.

The cycle-detection mechanism. As in the case of the queue, we will argue that with all but negligible probability the cycle-detection mechanism contains at most $g(N)$, for any $g(N) \in \omega(\log N)$, elements at any point in time. Therefore, we design the cycle-detection mechanism to store at most $g(N)$ elements, and allow the whole data structure to fail if the cycle-detection mechanism overflows. One possible instantiation is to use the above-mentioned instantiation of the queue together with any standard augmentation that enables constant time resets.

Note that in our case of negligible failure probability, the size of the queue and the cycle detection mechanism are both bounded by $g(N) = \widehat{O}(\log N)$, instead of being bounded by $\log N$ as in [2]. It is not hard to see that as long as the auxiliary data structures do not fail or overflow, all operations are performed in time $\widehat{O}(1)$. Thus, our goal is to prove that with $1 - \mathrm{negl}(N)$ probability, the data structures do not overflow.

We continue with the following definition, which will be useful for the efficiency analysis.

Definition 7. *Given a set $S \subseteq \mathcal{U}$ and two hash functions $h_0, h_1 : \mathcal{U} \to \{0, ..., r-1\}$, the cuckoo graph is the bipartite graph $G = (L, R, E)$, where $L = R = \{0, ..., r-1\}$ and $E = \{(h_0(x), h_1(x)) : x \in S\}$.*

For an element $x \in \mathcal{U}$ we denote by $C_{S,h_0,h_1}(x)$ the connected component that contains the edge $(h_0(x), h_1(x))$ in the cuckoo graph of the set $S \subseteq \mathcal{U}$ with functions h_0 and h_1.

Similarlty to [2], in order to prove Theorem 8, we require the following lemma:

Lemma 5. *For $T = f(N)$, where $f(N) = \omega(\log N)$, and $f(N) = o(\log N \log\log N)$, and $c_2 = \omega(1)$, we have that for any set $S \subseteq \mathcal{U}$ of size N and for any $x_1, \ldots, x_T \in S$ it holds that*

$$\Pr\left[\sum_{i=1}^{T} |C_{S,h_0,h_1}(x_i)| \geq c_2 T\right] \leq \mathrm{negl}(N),$$

where the probability is taken over the random choice of the functions $h_0, h_1 : \mathcal{U} \to \{0, ..., r-1\}$, for $r = (1+\epsilon)n$.

The proof of Lemma 5 will be discussed in Section A.1.

Denote by $\mathcal{E}_1$ the event in which for every $1 \leq j \leq N/f(N)$, where $f(N) = \omega(\log N)$, and $f(N) = o(\log N \log\log N)$, it holds that

$$\sum_{i=1}^{f(N)} |C_{S,h_0,h_1}(x_i)| \leq c_2 f(N).$$

By using Lemma 5 and applying a union bound, we have that $\mathcal{E}_1$ occurs with probability $1 - \mathrm{negl}(N)$.

We denote by $\mathsf{stash}(S_j, h_0, h_1)$ the number of stashed elements in the cuckoo graph of S_j with hash functions h_0 and h_1. Denote by $\mathcal{E}_2$ the event in which for every $1 \leq j \leq N/f(N)$, it holds that $\mathsf{stash}(S_j, h_0, h_1) \leq k$. A lemma of Kirsch *et al.* [25] implies that for $k = \omega(1)$, the probability of the event $\mathcal{E}_2$ is at least $1 - \mathrm{negl}(N)$.

The following lemmas prove Theorem 8:

Lemma 6. *Let π be a sequence of $p(N)$ operations. Assuming that the events $\mathcal{E}_1$ and $\mathcal{E}_2$ occur, then during the execution of π the queue does not contain more than $2f(N) + k$ elements at any point in time.*

Lemma 7. *Let π be a sequence of $p(N)$ operations. Assuming that the events $\mathcal{E}_1$ and $\mathcal{E}_2$ occur, then during the execution of π the cycle-detection mechanism does not contain more than $(c_2 + 1)f(N)$ elements at any point in time.*

The proofs of Lemmas 6 and 7 follow exactly as in [2], except the $\log N$ parameter from [2] is replaced with $f(N)$ in our proof.

A.1 Proving Lemma 5

As in [2], Lemma 5 is proved via Lemmas 8 and 9 below. Given these, the proof of Lemma 5 follows identically to the proof in [2].

Let $\mathbb{G}(N, N, p)$ denote the distribution on bipartite graphs $G = ([N], [N], E)$ where each edge is independently chosen with probability p. Given a graph G and a vertex v we denote by $C_G(v)$ the connected component of v in G.

Lemma 8. *Let $Np = c$ for some constant $0 < c < 1$. For $T = f(N)$, where $f(N) = \omega(\log N)$, and $f(N) = o(\log N \log\log N)$, and $c_2 = \omega(1)$, we have that for any vertices $v_1, \ldots, v_T \in L \cup R$*

$$\Pr\left[\sum_{i=1}^{T} |C_G(v_i)| \geq c_2 T\right] \leq \mathrm{negl}(N),$$

where the graph $G = (L, R, E)$ is sampled from $\mathbb{G}(N, N, p)$.

We first consider a slightly weaker claim that bounds the size of the union of several connected components:

Lemma 9. *Let $Np = c$ for some constant $0 < c < 1$. For $T = f(N)$, where $f(N) = \omega(\log N)$, and $f(N) = o(\log N \log\log N)$, and $c_2 = O(1)$, we have that for any vertices $v_1, \ldots, v_T \in L \cup R$*

$$\Pr\left[\left|\bigcup_{i=1}^{T} C_G(v_i)\right| \geq c_2 T\right] \leq \mathrm{negl}(N),$$

where the graph $G = (L, R, E)$ is sampled from $\mathbb{G}(N, N, p)$.

The proof of our Lemma 9 follows from Lemma 6.2 of [2]. Specifically, we observe that their Lemma 6.2 works for any choice of T (even though in their statement of Lemma 6.2, they require $T \leq \log N$. In particular, their Lemma 6.2 works for $T = f(N)$.

Next, the proof of our Lemma 8 can be obtained via a slight modification of the proof of Lemma 6.1 of [2]. Specifically, in their proof, they choose a constant c_3 and show that

$$\Pr\left[\sum_{i=1}^{T}|C_G(v_i)| \geq c_2c_3T\right] \leq \Pr\left[\left|\bigcup_{i=1}^{T}C_G(v_i)\right| \geq c_2T\right] + \frac{(c_2e)^{c_3}\cdot T^{2c_3+1}}{c_3^{c_3}\cdot n^{c_3}}.$$

By instead setting $c_3 = \omega(1)$, and using Lemma 9 to upperbound $\Pr\left[\left|\bigcup_{i=1}^{T}C_G(v_i)\right| \geq c_2T\right]$, we have that

$$\Pr\left[\sum_{i=1}^{T}|C_G(v_i)| \geq c_2c_3T\right] \leq \text{negl}(N).$$

To get from Lemma 8 to Lemma 5, we can go through the exact same arguments as [2] to show that $\mathbb{G}(N, N, p)$ is a good approximation of the Cuckoo graph distribution for an appropriate choice of p. Note once again that in our case of negligible failure probability, the size of the queue and the cycle detection mechanism are both bounded by $g(N) = O(f(N))$.