Deterministic, Stash-Free Write-Only ORAM

Daniel S. Roche, Adam J. Aviv, Seung Geol Choi, and Travis Mayberry

United States Naval Academy
Annapolis, Maryland, U.S.A.

June 9, 2017

Abstract

Write-Only Oblivious RAM (WoORAM) protocols provide privacy by encrypting the contents of data and also hiding the pattern of *write operations* over that data. WoORAMs provide better privacy than plain encryption and better performance than more general ORAM schemes (which hide both writing *and* reading access patterns), and the write-oblivious setting has been applied to important applications of cloud storage synchronization and encrypted hidden volumes. In this paper, we introduce an entirely new technique for Write-Only ORAM, called DetWoORAM. Unlike previous solutions, DetWoORAM uses a deterministic, sequential writing pattern without the need for any "stashing" of blocks in local state when writes fail. Our protocol, while conceptually simple, provides substantial improvement over prior solutions, both asymptotically and experimentally. In particular, under typical settings the DetWoORAM writes only 2 blocks (sequentially) to backend memory for each block written to the device, which is optimal. We have implemented our solution using the BUSE (block device in user-space) module and tested DetWoORAM against both an encryption only baseline of dm-crypt and prior, randomized WoORAM solutions, measuring only a 3x-14x slowdown compared to an encryption-only baseline and around 6x-19x speedup compared to prior work.

1 Introduction

ORAM Even when data is fully encrypted, the sequence of *which operations have been performed* may be easily observed. This *access pattern* leakage is prevented by using Oblivious RAMs (or ORAMs), which are protocols that allow a client to access files in storage without revealing the sequence of operations over that data. ORAM solutions that have been proposed provide strong privacy by guaranteeing that anyone who observes the entire communication channel between client and backend storage cannot distinguish any series of accesses from random. Due to this strong privacy guarantee, ORAM has been used as a powerful tool in various application settings such as secure cloud storage (e.g., [SS13b, SS13a, MBC14]), secure multi-party computation (e.g., [GKK$^+$12, LHS$^+$14, LWN$^+$15, WCS15, ZWR$^+$16]), and secure processors (e.g., [FRK$^+$15, LHM$^+$15, NFR$^+$17]).

Unfortunately, in order to achieve this obliviousness, ORAM schemes often require a substantial amount of shuffling during every access, requiring more encrypted data to be transferred than just the data being written/read. Even Path ORAM [SvDS+13], one of the most efficient ORAM constructions, has an $\Omega(\log N)$ multiplicative overhead in terms of communication complexity compared to non-private storage.

WoORAM Write-only ORAM (WoORAM) [LD13, BMNO14] introduces a relaxed security notion as compared to ORAMs, where only the write pattern needs to be oblivious. That is, we assume the adversary is able to see the entire history of which blocks have been written to the backend, or to view arbitrary snapshots of the storage, but that the adversary does not know which blocks are being read.

Every ORAM trivially satisfies the properties of WoORAM, but entirely different (and possibly more efficient) WoORAM solutions are available because *a WoORAM may still be secure even if reads are not oblivious*. WoORAM schemes can be used in application settings where adversaries are unable to gather information about physical *reads*. In such settings, the weaker security guarantee of WoORAM still suffice to hide the access patterns from the adversary of limited power.

Deniable storage [BMNO14] is one such application. In this setting, a user has a single encrypted volume and may optionally have a second, hidden volume, the existence of which the user wishes to be able to plausibly deny. For example, a laptop or mobile device owner may be forced to divulge their device decryption password at a border crossing or elsewhere. Even given all snapshots of the disk, an adversary should not be able to guess whether the user has a second hidden volume or not. In this context, it is reasonable to assume that the adversary won't get any information about block reads that have taken place in the disk, since read operations do not usually leave traces on the disk. Based on this observation, a hidden volume encryption (HiVE) for deniable storage was constructed based on WoORAM [BMNO14].

Another example application was proposed in [ACMR17] for *synchronization-based cloud storage and backup services*. Here, the user holds the entire contents of memory locally, and uses a service such as DropBox to synchronize with other devices or store backups. The service provider or an eavesdropper on the network only observes what the user writes to the synchronization folder, but does not see any read operations as these are done locally without the need for network communication. Aviv et al. [ACMR17] showed that WoORAMs can provide efficient protection in this scenario, as well as protection against timing and file size distribution attacks.

In both cases, what makes WoORAMs attractive is that they can achieve security much, much more efficiently than the full read/write oblivious ORAMs such as Path-ORAM. For example, consider storing N size-B data blocks in a non-recursive setting in which the client has enough memory to contain the entire position map of size $O(N \log N)$, Blass et al. [BMNO14] provided a WoORAM construction (hereafter, HiVE-WoORAM) with *optimal* asymptotic communication overhead of $O(B)$ and negligible stash overflow probability. As a comparison, fully-functional read/write ORAM schemes — again, even without the position map — have an overhead of $\Omega(B \log N)$.

	Logical Read		Logical Write		Client memory	Security
	Physical Read	Physical Write	Physical Read	Physical Write		
Path-ORAM [SvDS+13]	$O(B \log^2 N)$	$O(B \log^2 N)$	$O(B \log^2 N)$	$O(B \log^2 N)$	$\omega(B \log N)$	RW
HiVE-WoORAM [BMNO14]	$O(B \log N)$	0	$O(B \log^2 N)$	$O(B \log N)$	$\omega(B \log N)$	W-only
DetWoORAM	$O(B \log N)$	0	$O(B \log N)$	$2B$	$O(B)$	W-only

B denotes the block size (in bits), and N denotes the number of logical blocks. We assume $B = \Omega(\log^2 N)$.

Table 1: Communication complexity and client memory size for various ORAMs in the uniform block setting

Towards better efficiency with realistic client memory Although HiVE-WoORAM has a better asymptotic communication complexity than Path-ORAM in the non-recursive setting (i.e., with client memory of size $O(N \log N)$), the situation is different when the size of the client memory is smaller (i.e., polylogarithmic in N), or when the entire state needs to be frequently synchronized to backend storage. In this case, the client cannot maintain the entire position map in memory, and so the position map storage needs to be outsourced to the server as another WoORAM. This encoding typically occurs via a recursive process, storing the position map in recursively smaller WoORAMS, until the final WoORAM is small enough to fit within client memory. Therefore, in the uniform block setting where every storage block has the same size, both HiVE-WoORAM and Path-ORAM have the same overhead $O(B \log^2 N)$ with poly-logarithmic client memory size.[1]

Hence, we ask the following question:

> *Can we achive WoORAM with better asymptotic communication complexity in the setting of polylogarithmic client memory and uniform blocks?*

1.1 A Deterministic Approach to WoORAMs

In answering the question above, observe that the security requirement of WoORAMs are *much weaker* than that of ORAMs. Namely, only the write operations need to be oblivious, and the read operations can occur using different protocols than that of writing. This opens the door to a *radically different approach* toward constructing a WoORAM scheme.

Traditional approaches. Traditionally, in both ORAM and WoORAM schemes, to write data d, the oblivious algorithm requires selecting k blocks at random from physical storage in order to write. In Path-ORAM, those k blocks form a path in a tree, while in HiVE-WoORAM, they are randomly selected from a storage array of blocks. All k blocks are re-encrypted, but and the new block d is inserted if any existing blocks are empty.

[1] The multiplicative overhead $O(\log^2 N)$ can be reduced to additive overhead of $O(\log^3 N)$ if the size of the block can be non-uniform [SvDS+13, BMNO14]. However, throughout the paper we will consider the uniform block setting, since the two use cases we consider above assume uniform block sizes. We note that our construction still has better additive overhead of $O(\log^2 N)$ even in the non-uniform block setting.

One of the challenges with this approach is that there is the possibility for a write to fail if none of the random k blocks are empty and thus d cannot be inserted. Instead d is placed into a *stash* in reserved client memory until it may be successfully written to the ORAM (or WoORAM) when two or more of the k blocks are empty. Fortunately, the probability of this event is bounded, and thus the size of the stash can also be bounded with negligible stash overflow probability. The schemes will, with overwhelming probability, work for small client memory.

Main observations. After careful inspection of the security proofs, we discovered that *random slots are mainly used to hide read accesses, not write accesses!* That is, the challenge for ORAMs is that successive reads of the same data must occur in a randomly indistinguishable manner. For example, without the technique of choosing random slots, two logical reads on the same address may result in reading the same physical address twice, in which case the read accesses are not oblivious. In the WoORAM setting, however, *the scheme may still be secure even if reads are not oblivious*, since the security requirement doesn't care about physical reads! Based on this observation, we ask:

> *Can we construct a deterministic WoORAM scheme using a radically different framework?*

1.2 Our Work: DetWoORAM

We answer both of the above questions affirmatively. In what follows, we describe the main features and contributions of DetWoORAM.

Deterministic, sequential physical accesses DetWoORAM departs from the traditional approach in constructing a WoORAM scheme in that the write pattern is deterministic and sequential. Roughly speaking, if some logical *write* results in writing the two physical blocks i and j, the next logical *write* will result in writing in physical blocks $(i + 1) \bmod N$ and $(j + 1) \bmod M$, where M is a parameter in the system.

No stash The deterministic nature of the physical writes also implies that a stash is no longer needed. A write will always succeed and always occurs in a free block. Therefore, we were able to *remove the notion of stash completely* in our scheme. To elaborate this point, we give a very simple toy construction that captures these aspects in Section 3.1. Due to the deterministic access pattern and the absence of stash, security analysis of our scheme is extremely simple; in fact, the security of our scheme holds simply by observation.

***Optimal* communication complexity of physical writes** DetWoORAM has better asymptotic communication complexity than previous constructions (see Table 1). In particular, DetWoORAM improves the complexity of write operations compared to HiVE-WoORAM by a factor of $\log N$.

We stress that to the best of our knowledge, DetWoORAM is *the first WoORAM* construction whose physical write cost is $2B$ for a single logical write. In other words, the physical-write overhead is *a single block (additive!)*, which is *exactly optimal in the uniform block setting with small client storage.*

Optimization techniques: ODS and packing We applied two optimization techniques to further reduce the communication complexity and improve practical performance.

First, we created a new write-only oblivious data structure (ODS), in the form of a Trie, to function as the position map. As with previous tree-based ODS schemes [WNL+14, RAC16], our ODS scheme avoids recursive position map lookups by employing a *pointer-based technique.* That is, the pointer to a child node in a Trie node *directly points to a physical location* instead of a logical location, and therefore it is no longer necessary to translate a logical address to a physical address within the Trie itself. We note that the ODS idea has previously been applied to WoORAM by [CCS17], although their overall scheme turns out the be insecure (see Section 6).

With the simpler position map stored as a Trie and the deterministic write-access pattern in DetWoORAM, we can manipulate the parameters to optimize the procedure with DetWoORAM. In particular, we will show how to pack write-updates of the position map Trie into block size chunks. With additional interleaving techniques, we will show that we can achieve a minimal communication complexity of $2B$, one block for the data and one block for position map and other updates. The details of these techniques are described in Section 3.3 and 3.4.

Stateless WoORAM WoORAM is usually considered in a single-client model, but it is sometimes useful to have multiple clients accessing the same WoORAM. Since our scheme has no stash, we can convert our scheme to a stateless WoORAM with no overhead except for storing the encryption key and a few counter variables. On the other hand, previous schemes such as Path-ORAM and HiVE-WoORAM must maintain the stash of size $\omega(B \log N)$, which in the stateless setting must be transferred in its entirety on each write operation in order to maintain obliviousness.

Less randomness and storage for IVs The deterministic and sequential access pattern fits nicely with encryption of each block using *counter mode.* Suppose the previous writes so far have cycled the physical storage i times, and physical block j is about to be encrypted. Then, the block can be encrypted with the counter $i \| j \| 0^{\ell}$, where ℓ depends on how many times one needs to apply a block cipher to encrypt the entire block. That is, we can get indistinguishable symmetric encryption by storing just a single IV.

We stress that the above optimization cannot be applied to previous schemes due to the randomized procedure. For example, HiVE-WoORAM chooses instead random IVs to encrypt each block. These IVs must be stored separately on the server adding to the communication cost overhead.

Additionally, we remark that the implementation [BMNO14] of HiVE-WoORAM used RC4 as a PRG to choose random IVs for performance reasons, but the insecurity of RC4 lead to an attack on the implementation [PS15].

Implementation and experiments We have implemented DetWoORAM using C++ and BUSE (*block device in user space*). We tested our implementation using Bonnie++ and fio on both a spinning-platter hard drive and a solid state drive, comparing the implementation to a baseline that performs encryption only (no obliviousness) as well as to an implementation of HiVE-WoORAM. We found that DetWoORAM incurs only a 3x-14x slowdown compared to an encryption-only baseline and around 6x-19x speedup compared to HiVE-WoORAM.

Insecurity of other proposed WoORAM improvements DataLair [CCS17] and FlatORAM [HvD16] are two other WoORAM schemes that have been proposed recently with the goal of improving the practical performance compared to HiVE-WoORAM. As our secondary contribution, we analyzed these two WoORAM protocols, both of which achieve faster writes by tracking empty blocks within the WoORAM. We show in Section 6 that, unfortunately, neither of these constructions satisfies write-only obliviousness.

2 Background

2.1 Write-only ORAM

ORAM An Oblivious RAM (ORAM) allows a client to store and manipulate an array of N blocks on an untrusted, honest-but-curious server without revealing the data or access patterns to the server. Specifically, the (logical) array of N blocks is indirectly stored into a specialized backend data structure on the server, and an ORAM scheme provides an access protocol that implements each logical access with a sequence of physical accesses to the backend structure. An ORAM scheme is secure if for any two sequences of logical accesses of the same length, the physical accesses produced by the access protocol are computationally indistinguishable.

More formally, let $\vec{y} = (y_1, y_2, \ldots)$ denote a sequence of operations, where each y_i is a $read(a_i)$ or a $write(a_i, d_i)$; here, $a_i \in [0, N)$ denotes the logical address of the block being read or written, and d_i denotes a block of data being written. For an ORAM scheme Π, let $\mathsf{PhysicalAcc}^{\Pi}(\vec{y})$ denote the physical access pattern that its access protocol produces for the logical access sequence $\vec{y}$. We say the scheme Π is *secure* if for any two sequences of operations $\vec{x}$ and $\vec{y}$ of the same length, it holds

$$\mathsf{PhysicalAcc}^{\Pi}(\vec{x}) \;\approx_c\; \mathsf{PhysicalAcc}^{\Pi}(\vec{y}),$$

where $\approx_c$ denotes computational indistingushability (with respect to the security parameter λ).

Since the seminal work by Goldreich and Ostrovsky [GO96], many ORAM schemes have been proposed and studied in the literature; see Section 7 for more related work.

WoORAM Blass et al. [BMNO14] considered a relaxed security notion of write-only ORAM (WoORAM), where only the *write* physical accesses are required to be indistinguishable. In particular, we say an ORAM scheme Π is *write-only oblivious* if for any two sequences of logical accesses $\vec{x}$ and $\vec{y}$ containing the same number of *write* operations, it holds

$$\mathsf{WOnly}(\mathsf{PhysicalAcc}^{\Pi}(\vec{x})) \approx_c \mathsf{WOnly}(\mathsf{PhysicalAcc}^{\Pi}(\vec{y})),$$

where WOnly denotes a function that filters out the read physical accesses but passes the write physical accesses.

They also gave an WoORAM construction which is much more efficient than full ORAM constructions. We will briefly describe their construction below.

2.2 HiVE-WoORAM

Setting In [BMNO14], to store N logical blocks, the server needs a physical array D of $M \geq 2N$ elements, where each element is a pair (a, d), where a is the logical address and d is the actual data. Obviously, all the data in the backend storage D is encrypted with an IND-CPA encrption scheme; throughout the paper, we will implicitly assume that the backend data is encrypted with an IND-CPA encryption scheme, even if we don't use any encryption notations.

The client maintains a buffer, called stash, that temporarily holds the blocks yet to be written to D. We assume for now that the client also maintains the position map *pos* in its memory; the map *pos* translates a logical address into the corresponding physical address.

This protocol depends crucially on parameter k, the number of physical writes per logical write. This is selected to ensure a very low probability of filling up the stash; according to [BMNO14], for $k = 3$, the probability of having more than 50 items in the stash at any given time is bounded by 2^{-64}.

Write algorithm The access protocol for $write(a, d)$ works as follows.

1. Insert (a, d) into stash
2. Choose k physical addresses $r_1, \ldots, r_k$ uniformly at random from $[0, M)$.
3. For $i \in \{1, \ldots, k\}$ do:
   (a) Determine whether $D[r_i]$ is free by checking whether $pos[D[r_i].a] \neq r_i$.
   (b) If $D[r_i]$ is free and stash is nonempty, remove an element (α, δ) from stash, set $D[r_i] \leftarrow \beta$, $D[r_i].a \leftarrow \alpha$, and update the position map $pos[\alpha] \leftarrow r_i$.
   (c) Otherwise, rewrite $D[r_i]$ under a new random IV.

Communication complexity Let $M = O(N)$. Without considering the position map, their access protocol for write has fantastic communication complexity of $O(k(\log N + B))$, where B is the size of a data block. In particular, with $k = 3$ and assuming $B = \Omega(\log N)$, the communication complexity is $O(B)$. However, the size of the position map is $\Omega(N \cdot \log N)$, which is usually too large for the client to store in memory. This issue can be addressed by recursively storing the map in smaller and smaller ORAMs. Taking these recursion steps into account, the eventual communication complexity becomes $O(B \log^2 N)$.

3 Deterministic WoRAM Design

In this section, we describe the algorithm for DetWoORAM construction. We begin by first describing a "toy construction" that has some of the key properties as the final algorithm, such as not employing a stash while using a deterministic write pattern. From this toy construction, we make a series of improvements that lead to our actual DetWoORAM construction with sequential write pattern and $2B$ communication cost per write.

3.1 A *Toy* Deterministic WoRAM Construction

The *toy* deterministic WoRAM construction is inspired by the *square-root* ORAM solution of Goldreich and Ostrovsky [GO96], adapted to the write-only oblivious setting. For now, we set aside the issue of the position map, which one could consider being stored by the user locally and separately from the primary procedure. Later, we will describe a method for storing the position map within an adjacent, smaller WoRAM.

Toy Construction. Physical storage consists of an array D of $2N$ data blocks of B bits each. D is divided into two areas, a *main area* and a *holding area* (see Figure 1), where each area contains N blocks.

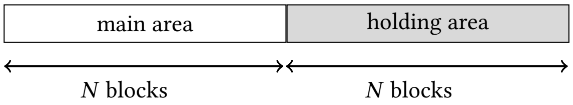

Figure 1: Physical data array D for the toy construction.

Each block of the storage area has an address a, and a user interacts with the system by writing data to an address within the main area, that is $a \in [0, N)$, and reading data back by referring to the address. In the main area, a block is always stored at its address, but in the holding area, blocks are appended as they are written, irrespective of their address.

In order to track where to write the next block in the holding area, we keep a counter i of the number of write operations performed so far. Additionally, as the holding area is not ordered, there needs to be a position map that associates addresses $a \in [0, N)$ to the location in $[0, 2N)$ of the freshest data associated with that address, either at the address within the main area or a more recent write to the holding area. The position map construction will be described later as a write-only oblivious data structure stored with in an adjacent, smaller WoORAM. For now, we abstract position map as a straightforward key-value store with operations $\textsc{getpos}(a) \rightarrow a'$ and $\textsc{setpos}(a, a')$

With parameter N, counter i, and the WoORAM storage array D, where $|D| = 2N$, we can now define the primary operations of the toy WoORAM as in Algorithm 1.

Properties of toy construction. Already, our toy construction has some of the important of the properties of our final construction. As explained below, it provides

Algorithm 1 Operations in Toy Deterministic WoRAM

```
//Perform the $i$-th write, storing data $d$ at address $a$
function WRITE$_i(a, d)$
    $D[N + (i \bmod N)] :=$ ENC$(d)$ //Write to holding area
    SETPOS$(a, N + (i \bmod N))$ //Update position map
    $i := i + 1$ //increment counter

    //Refresh the main area
    if $i \bmod N = 0$ then
        for $a \in [0, N)$ do
            $D[a] :=$ ENC(DEC($D$[GETPOS($a$)]))
            SETPOS$(a, a)$ //Update the position map
        end for
    end if
end function

//Read and return data at address $a$
function READ$(a)$
    return DEC($D$[GETPOS($a$)]) //Return freshest version of the data
end function
```

write obliviousness, it is deterministic, and it does not require a stash.

To see why the toy construction is write-oblivious, first consider that each write occurs sequentially in the holding area and has no correspondence to the actual address of the data. Writing to the holding area does not reveal the address of the data. Second, once the holding area has been filled completely with the freshest data, after N operations, *all* the main area blocks are refreshed with data from the holding area, or re-encrypted if no fresher data is found in the holding area. Since all the main area blocks are written during a refresh, it is impossible to determine which of the addresses occurred in the holding area. In both cases, for a write to the holding area and during a refresh, the block writes are oblivious.

The toy construction also has a *deterministic* write pattern: the i-th write always touches the holding area block at index $N + (i \bmod N)$. As compared to previous ORAM and WoORAM schemes, in which writing (or access) requires randomly selecting a set (or path) of blocks to overwrite with the expectation that at least one of the blocks has the requisite space to store the written data, our construction does not require any random selection and operates in a completely deterministic manner.

Further, as each write is guaranteed to succeed—we always write sequentially to the holding area—there is no need for a stash. To the best of our knowledge, all other WoORAM schemes require a stash to handle failed write attempts. In some sense, one can think of the stash in these systems as providing state information about the current incomplete writes, and to have a stateless system the full size of stash would need to be transferred on every step (even if there is nothing in it). By contrast, our construction has constant state cost, which consists simply of the counter i and the encryption key.

3.2 De-amortizing the toy construction

We can advance upon the toy constructing above by further generalizing the storage procedure via de-amortization of the refresh procedure as well as allowing the main and holding area to be of different sizes. The key idea of de-amortization is that instead of refreshing the main area once the holding area has been fully written, we can perform a few writes to the main area for each write to the holding area, so that it is fully refreshed at the same rate.

In this generalized setting, physical storage consists of a main area of size N as before, and a holding area of size M, where M is arbitrary, so that $|D| = N + M$.

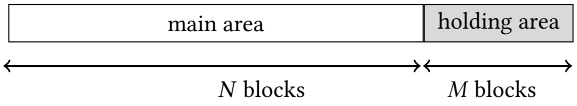

Figure 2: Back end data array D with unequal main and holding areas.

The key to the de-amortized write procedure is that there needs to be a commensurate number of refreshes to the main area for each write to the holding area. After any consecutive M writes to the holding area, the entire main area (of size N) needs to be refreshed, just like what would happen in the amortized toy construction.

When $N = M$, this is simply accomplished by performing one refresh for each write. When the sizes are unequal, we need to perform on average N/M refreshes per write to achieve the same goal. For example, consider the case where $N = 2M$, where the main area is twice as large as the holding area, then $N/M = 2$, and thus we perform two refreshes for every write. After M writes to the holding area, the entire main area will have been refreshed.

It is also possible to have ratios where $M > N$, such as $M = 2N$ where the main area is half the size of the holding area, and in fact, this setting and $M = N \cdot \lceil \log(N) \rceil$ are both critical settings for performance. When $M > N$ this implies that we need to do *less* than one refresh per write, on average. Specifically for $N/M = 1/2$, we perform a refresh on *every other write* to the holding area.

Algorithm 2 has the properties of performing on average N/M refreshes per write, while the read operation is the same as before.

It is straightforward to see that the unequal size, de-amortized solution has the same key properties as the toy construction: it is write-oblivious, deterministic, and does not require a stash. It is clearly deterministic because just as before, writes and refreshes occur sequentially in the holding area and main area, respectively, and this also assures write-obliviousness for the same reasons discussed before. It still does not require a stash because every write will succeed, as the refresh pattern guarantees that the next write to the holding area will always overwrite a block that has already had the chance to be refreshed to the main area.

3.3 Incorporating the Position Map

In this section, we consider methods for implementing a position map for DetWoORAM, and crucially, modifying the procedure so that only a *single position map up-*

Algorithm 2 De-amortized write operation with unequal size main and holding areas.

```
//Perform the i-th write, storing data d at address a
function WRITE($a, d$)
    $D[N + (i \bmod M)] := \textsc{enc}(d)$ //Write to holding area
    $\textsc{setpos}(a, N + (i \bmod M))$ //Update position map

    //Refresh N/M main area blocks per-write
    $s := \lfloor i \cdot N/M \rfloor \bmod N$
    $e := \lfloor (i+1) \cdot N/M \rfloor \bmod N$
    for $a' \in [s, e)$ do
        $D[a'] := \textsc{enc}(\textsc{dec}(D[\textsc{getpos}(a')]))$
        $\textsc{setpos}(a', a)$
    end for
    $i := i + 1$ //increment counter
end function
```

date per write is needed.

We first describe how to modify our algorithm so that we can store the position map recursively within successively smaller DetWoORAMs, and then show how to further improve by using a Trie-based *write-only oblivious data structure* (WoODS) stored within an adjacent DetWoORAM to the main, data-storing one.

Recursively stored position map. One possibility for storing the position map is to pack as many positions as possible into a single block, and then store an adjacent, smaller WoORAM containing these position map blocks only. Then *that* WoORAM's position map is stored in a smaller one, and so on, until the size is a constant and can be stored in memory or refreshed on each write. If at least two positions can be packed into each block, the number of levels in such a *recursive position map* is $O(\log_B N)$.

If we consider each of the recursive WoRAM's using the same write procedure as described in Algorithm 2, a problem quickly emerges. A write requires multiple updates to the position map due to the de-amortized procedure: one update to store the location within the holding area of the newly written data, and some number of updates to store the refreshed main areas. In a recursive setting, as these position map updates must occur for every recursive level of the position map, we can get exponential blow up. One write to the main WoRAM requires $O((1 + M/N)^R)$ writes at the smallest WoRAM, where $R \in O(\log_B N)$ is the number of recursive levels.

In HiVE-WoORAM, this issue is solved using additional state information of "metadata blocks", each containing the actual index of the block as well as the IV used to encrypt that block. These metadata blocks are stored alongside the primary physical blocks for the WoORAM. Crucially, by storing the actual *index* associated with each block in memory, it is no longer necessary to update the position map multiple times for each write. While something similar would work for our system, we solve this problem more efficiently, avoiding the need for extra storage of metadata blocks entirely.

Algorithm 3 DetWoORAM Operations with a Pointer Based Position Map: main area size N, holding area size M, data array D, and counter i

```
//Read and return data for address a
function READ($a$)
    $(a_s, o, q) :=$ GETPOS($a$)
    $B_m =$ DEC($D[a]$)
    if $B_m[o] = q$ then return $B_m$
    else return DEC($D[a_s]$)
    end if
end function

//Perform the i-th write of data d to address a
function WRITE($a, d$)
    $a_s := N + (i \bmod M)$ //Holding address
    $D[a_s] :=$ ENC($d$) //Write to holding area
    $(o, q) :=$ DIFF($d$, DEC($D[a]$)) //Offset o and bit diff q
    SETPOS($a, (a_s, o, q)$) //Update Position Map

    //Refresh N/M main area blocks per-write
    $s := \lfloor i \cdot N/M \rfloor \bmod N$
    $e := \lfloor (i+1) \cdot N/M \rfloor \bmod N$
    for $a_m \in [s, e)$ do
        $(a_s, o, q) :=$ GETPOS($a_m$)
        $D[a_m] :=$ ENC(DEC($D[a_s]$))
        //No position map update needed
    end for
    $i := i + 1$ //increment counter
end function
```

Position map pointers and one-bit diff technique. To improve the position map and remove exponential blow-up in updating the position map, we recognize that we have a distinct advantage in DetWoORAM construction as compared to prior schemes in that for main area blocks, data is always located at its address. The holding area is the only portion of the WoRAM that needs a position map. *The position map does not need to be updated for a refresh if we could determine the freshest block during a read.*

To see this, consider a position map that simply stores a holding-area address. When we perform a read of address a, we need to look in two locations, both in the holding area at where the position map says a is and in the main area at a. Given these two blocks, which is freshest data associated with a?

We can perform a freshness check between two blocks using the *one-bit diff* technique. Specifically, the position map gives a mapping of logical address $a \in [0, N)$ to a tuple (a_s, o, q), where $a_s \in [0, M)$ is an address to the holding area, $o \in [0, B)$ is a bit offset within a block, and $q \in \{0, 1\}$ is the bit value of the freshest block at the offset o. We define the tuple (a_s, o, q) as a *position map pointer.*

Whenever a write occurs for logical address a to holding area a_s, the offset o is chosen so as to *invalidate* the old data at address a in the main area. Specifically, we ensure that the oth bit of the new, fresh data $\textsc{dec}(D[a_s])$ is different from the oth bit of the old, stale data $\textsc{dec}(D[a])$. (If there is no difference between these, then the old data is not really stale and the offset o can take any valid index, say 0.)

Given the pointer (a_s, o, q), a freshness check between two blocks $\textsc{dec}(D[a])$ and $\textsc{dec}(D[a_s])$ is performed as follows:

> *Check if the oth bit of $\textsc{dec}(D[a])$ is q. If so, $\textsc{dec}(D[a])$ is fresh; otherwise $\textsc{dec}(D[a_s])$ is fresh.*

The key observation is that *when a block is refreshed to the main area, there is no need to update the position map with a new pointer*, since the read operation always starts by checking if the block in the main area is fresh. If the main area block is fresh, then there is no need to even look up the holding area position (which may have been rewritten with some newer block for a different logical address). See Algorithm 3 for details of how this is accomplished.

Trie WoODS for Position Map A more efficient solution for storing the position map, as compared to the recursively stored position map, is to use an oblivious data structure (ODS) in the form of a Trie. Recall that Trie edges are labeled, and looking up a node with a keyword $w_1 w_2 \cdots w_\ell$ is performed by starting with a root node and following the edge labeled with w_1, and then with w_2, and all the way through the edge labeled with w_ℓ one by one, finally reaching the target node.

As with previous tree-based ODS schemes [WNL$^+$14, RAC16], our ODS scheme avoids recursive position map lookups by employing a *pointer-based technique*. That is, the pointer to a child node in a Trie node *directly points to a physical location* instead of a logical location, and therefore it is no longer necessary to translate a logical address to a physical address within the Trie itself.

Applying an ODS in a write-only setting (a WoODS or write-only oblivious data structure) is similar to an idea proposed by Chakraborti et. al [CCS17]. A major difference in our construction is that we do not store the data structure within the primary WoORAM. We also allow changing the branching factor of the Trie independently of the block size, so we can tune the secondary WoORAM and flexibly control the number of physical block writes for every logical write, including position map information stored within the Trie.

As the WoODS Trie is stored in an adjacent DetWoORAM construction, we differentiate between the two WoORAMs by referring to the *data WoORAM* as the WoORAM storing data blocks and the *position WoORAM* as the WoORAM storing the nodes of the Trie. The Trie itself acts as the position map, and will map addresses in the data WoRAM's main area to position map pointers referencing the data WoRAM's holding area. The main idea is that given an address a, one can walk the Trie to find a leaf node storing a's position map pointer. The position WoRAM will be strictly smaller than the data WoRAM, but will be implemented using the same DetWoORAM framework (i.e., using the notions of main area, the holding area, and the counter).

Details of the procedure for the position WoORAM is outlined in Algorithm 4. Observe that the functions for the position WoORAM call the READ and WRITE functions from Algorithm 3, but with modified versions of the subroutines for accessing and updating the position map, as the Trie is its own position map.

Algorithm 4 Trie WoODS with N_p nodes and branching factor b. READ and WRITE calls are the routines in Algorithm 3 (with modified subroutines as specified) applied to position WoORAM instantiated with N_p main blocks and M_p holding blocks.

```
//$a \in [0, N_p + N)$ is a position WoORAM or data WoORAM address
function PATH-INDICES($a$)
    if $a = 0$ then return [] //Base case: empty path to root node
    else return [PATH-INDICES($\lfloor (a-1)/b \rfloor$), $(a-1) \bmod b$]
    end if
end function

//Retrieve Trie nodes along a path
function PATH-NODES($a_0, a_1, \ldots, a_{\ell-1}$)
    $B_0 :=$ root node //Root node is kept in local state
    $a := 0$
    for $i = 0, \ldots, \ell - 1$ do
        $ptr := B_i[a_i]$ //The pointer to the $a_i$th child, i.e., $ptr = (a_p, o, q)$
        $a := (a_i + 1) + b \cdot a$
        $B_{i+1} :=$ READ($a$) in Alg. 3 with its subroutine changed as:
           ▷ GETPOS($a$) returns $ptr$
    end for
    return $(B_0, \ldots, B_\ell)$
end function

//$a$ is an address; $src$ is either DATA or TRIE
function GETPOS-TRIE($a, src$)
    if $src =$ DATA then $a_0, a_1, \ldots, a_\ell :=$ PATH-INDICES($N_p + a$)
    else $a_0, a_1, \ldots, a_\ell :=$ PATH-INDICES($a$)
    end if
    $(B_0, \ldots, B_\ell) :=$ PATH-NODES($a_0, a_1, \cdots a_{\ell-1}$)
    return $B_\ell[a_\ell]$
end function

//$a$ is a data WoORAM index; $ptr$ is a pointer
function SETPOS($a, ptr$)
    $a_0, a_1, \ldots, a_\ell :=$ PATH-INDICES($N_p + a$)
    $(B_0, \ldots, B_\ell) :=$ PATH-NODES($a_0, a_1 \cdots a_{\ell-1}$)
    $B_\ell[a_\ell] := ptr$ //Change the leaf first
    for $j = \ell, \ldots, 1$ do //from leaf to root
        Call WRITE($a_{j-1}, B_j$) in Alg. 3 with its subroutines changed as:
           ▷ SETPOS($a_{j-1}, ptr'$) assigns $ptr'$ to $B_{j-1}[a_{j-1}]$
           ▷ GETPOS($a$) returns GETPOS-TRIE($a$, TRIE)
    end for
    if $\ell \neq \lceil \log_b(N_p) \rceil$ then write a dummy Trie node end if
    Root node $:= B_0$
end function
```

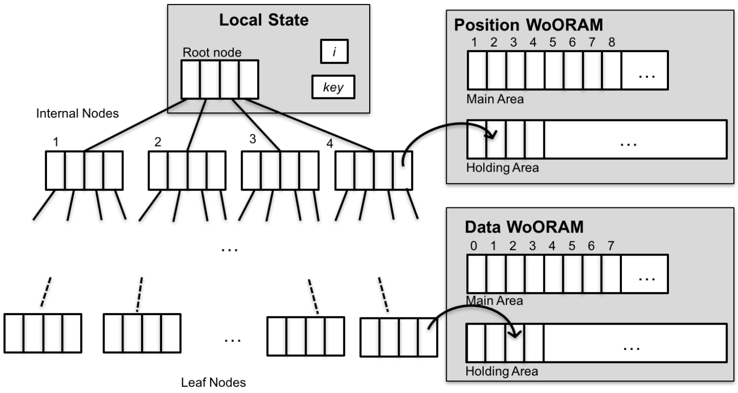

Figure 3: DetWoORAM Diagram with Trie Position Map

As noted, the Trie is stored in an adjacent WoORAM that has a main area and a holding area. The key difference is that the Trie nodes are addressed with the position WoORAM's main area using heap indexing. For example, with a branching factor of $b = 4$, the root node of the Trie has address 0, its children are at address 1, 2, 3, and 4, and their children are at addresses (5, 6, 7, 8), (9, 10, 11, 12), and so on. By using heap indexing, the structure of the Trie reveals the position of its nodes, becoming its own position map. In particular, this indexing avoids the need to store edge labels explicitly; they can instead be stored implicitly according to the heap indexing formulas.

It is still possible for a node of the Trie to have been recently updated and thus the freshest node information to be resident in the holding area of the position WoORAM. As such, each internal node of the Trie stores b position map pointers to the position WoORAM's holding area, one for each of its child nodes. A leaf node in the Trie then stores b position map pointers to the data WoORAM's holding area. The root node of the Trie can be stored as part of the local state, since it is constantly rewritten and read on every operation. A visual of the Trie is provided in Figure 3.

Reading from the Trie to retrieve a position map pointer for the data WoORAM is a straightforward process. One only needs to traverse from the root node to a leaf, following a path dictated by the address a called via GETPOS-TRIE(a, DATA). On each step down the tree, the current Trie node stores the position map pointer of the child node; the corresponding sequence of nodes are retrieved via the PATH-NODES helper function. The position map pointers for the data WoORAM can be found at the correct index in the leaf node along the fetched path.

Updating a pointer in the Trie (by calling SETPOS(a, ptr)) is a bit more involved. An update of the position map for the data WoORAM requires updating a leaf node in the Trie within the position WoORAM. Writing that leaf node will change its pointer, which requires updating the parent node, whose pointer will then also change, and so on up to the root of the Trie. That is, *each write to the main WoORAM requires rewriting an entire path of Trie nodes within the position WoORAM.*

Recall that in DetWoORAM, each write operation not only writes one block to the holding area, but also performs some *refreshes* in the main area. The challenge is, for each refresh, determining where in the holding area fresher data might be. For the data WoORAM, this is achieved simply by performing a lookup in the position map. But for the position WoORAM, there is no position map! Instead, we use the Trie itself to look up the pointer for fresher data in a position WoORAM refresh operation, by calling GETPOS-TRIE(a, TRIE). This is possible again because of the heap indexing; from the index of the Trie node that is being refreshed, we can determine all the indices of the nodes along the path to that one, and then perform lookups for the nodes in that path to find the position WoORAM holding area location of the node being refreshed.

Trie WoODS parameters and analysis We start by calculating N_p, which is the number of Trie nodes as well as the size of the position WoORAM main area. This needs to be large enough so that there is room for N pointers in the leaf nodes, where N is the number of logical addresses in the data WOORAM. With branching factor b, the number of Trie nodes is given by

$$N_p = \left\lfloor \frac{N-2}{b-1} \right\rfloor \in O(N/b). \tag{1}$$

The height of the Trie is then

$$h = \lceil \log_b N_p \rceil \in O(\log_b N), \tag{2}$$

which is the number of Trie nodes that need to be written to the holding area of the position WoORAM on each update (including a potential dummy node).

Each write to the data WoORAM requires rewriting $h \in O(\log_b N)$ nodes in the Trie (for a single path). Each of those writes to the holding area of the position WoORAM needs, on average, N_p/M_p number of refreshes to the main area, where N_p is the size of the position WoORAM's main area and M_p is the size of the position WoORAM's holding area.

Looking up a position in the Trie requires reading $O(\log_b N)$ blocks in the position WoORAM, each of which results in up to two physical reads due to having to check for fresher data. A refresh operation in the position WoORAM also requires a read of the Trie to determine if fresher data for that node exists in the holding area. If the sizes of the main area N_p and holding area M_p for the position WoORAM are not set appropriately, this could lead to $O(\log^2 N)$ reads to perform an update. However, consider that we can control the ratio N_p/M_p. If we set $M_p \gg N_p \log_b N$ then we need to perform only $O(1)$ refreshes per position map update, thus requiring $O(\log N_p)$ reads per update.

If N is a power of b, then the number of leaf nodes in the Trie N/b is also a power of b, and the Trie is a complete b-ary tree of height $\log_b(N/b)$. If N is not a power of b, then the last level of the Trie is incomplete, and leaf node heights may differ by one. In order to preserve write obliviousness, in cases of rewriting a path with smaller height, we add one additional dummy node write.

Finally, observe that the branching factor b can play a role in the performance. With $b = 2$, the *size* of a Trie path is minimized, but the height and number of nodes

N_p are maximal. Increasing b will reduce the height of the Trie and the number of Trie nodes, while increasing the total size of a single path. As we will show next, adjusting the *packing* of position map by setting the branching factor b can be done carefully to achieve write communication cost of exactly $2B$ in a fully sequential write pattern.

3.4 Fully Sequential Physical Write Pattern

In this section, we describe how to achieve fully sequential writing of physical storage and how to minimize the communication cost. This requires interleaving the various storage elements of DetWoORAM such that all the writing, regardless of which part of the construction is being written, occurs sequentially.

To understand the challenge at hand, first consider a simple implementation which aligns the data WoORAM (main and holding areas) adjacent to the storage WoORAM (main and holding area) forming a single storage data array broken into size-B blocks. A write to the data WoORAM will result in a write to the holding area plus M/N average writes to the main area. The position map is also updated, requiring $O(\log_b N)$ nodes in the Trie to be written to the holding area and $O(1)$ refreshes of the position WoORAM's main area, provided $M_p \in \Omega(N_p \log_b N)$. While all these writes occur sequentially within their respective data/position WoORAM main/holding areas, do *not* occur sequentially on the underlying storage device as each of the various WoORAM areas are separated. Furthermore, the writes to the position map are wasteful in that they may update only a few nodes, constituting just a small fraction of the block, but in the uniform access model this in fact requires updating the entire block.

We can improve on this storage layout and achieve a minimum in write performance requiring exactly 2 blocks to be written to physical storage for each block write, where one block is the new data, a half block worth of main area refresh, and a half block worth of position map updates. Further, we can interleave the various WoORAM portions such that those 2 blocks are written sequentially on the physical device.

Data WoORAM Block Interleaving Every logical block write to the data WoORAM results in exactly one block write to the holding area of data WoORAM. Recall that there are two parameters for setting up DetWoORAM: N, the size of the main area, and M, the size of the holding area. These two values need not be the same, and in fact, to achieve sequential writing, we will set $M = 2N$. In this case, on average $\frac{M}{N} = \frac{1}{2}$ block is refreshed to the main area for each logical write.

With adjacent main and holding areas, this could be achieved by performing one full block refresh on *every other* logical write. To make the writing sequential, we will instead refresh *half* of a block on every logical write, resulting in the following storage layout:

$$h_0, \frac{m_0^0}{\square}, h_1, \frac{m_0^1}{\square}, h_2, \frac{m_1^0}{\square}, h_3, \frac{m_1^1}{\square}, \ldots, h_{M-2}, \frac{m_{N-1}^0}{\square}, h_{M-1}, \frac{m_{N-1}^1}{\square}$$

where m_j^0 is the first half of the block m_j, m_j^1 is the back half of m_j, and $\square$ represents empty space. (This empty space will be used to store nodes for the position WoORAM, as we will show next.)

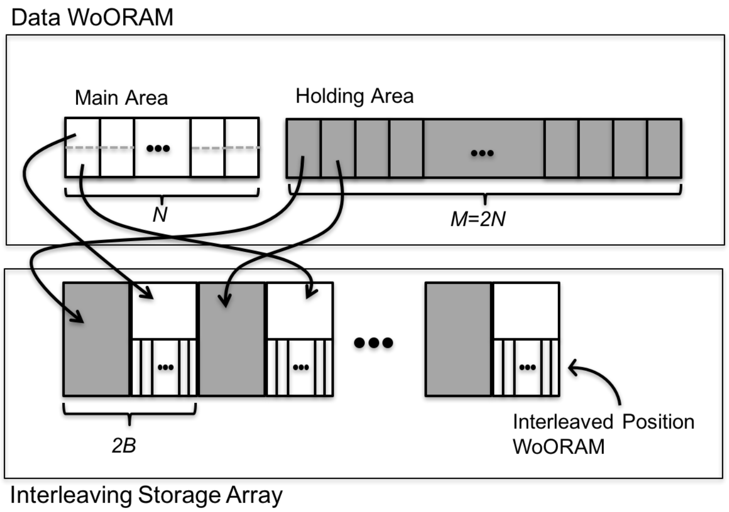

Figure 4: Interleaving of WoORAMs into a Storage Array with $2B$ size sequential writes.

There is a slight complication to reading now, as a single main area block is actually divided between two physical memory locations, resulting in an additional (constant) overhead for reading operations. The benefit is that the writing is fully sequential now: each logical write requires writing sequentially the data being updated (to the holding area), and the next *half* block of data being refreshed (to the main area), plus another half-block containing position map information as we will detail next. Also observe that, under this configuration with $M = 2N$, the total physical memory requirement will be $4N$ blocks.

Position WoORAM Block Interleaving As suggested above, the remaining half-block of data □ in the above construction will be used to store position map information. A diagram storage achieving 2 sequential physical block writes per logical write is shown in Figure 4.

Specifically, these $\frac{B}{2}$ bits will store the Trie nodes written to position WoORAM holding and main areas during a single logical write operation. This is (potentially) possible because the Trie nodes in the position WoORAM are much smaller than the blocks in the data WoORAM. Fully sequential writing will be achieved if and only if all of the Trie nodes written during a single step can always fit into $B/2$ bits.

There are many settings of parameters M, b, and M_p that may make sequential writing possible, depending on logical and physical memory requirements and the physical block size B. We will choose some parameters here and demonstrate that they would work for any conceivable value of N.

For this purpose, set the branching factor $b = 2$, and then recall from (1) and (2) that the number of Trie nodes and the Trie height will be $N_p = N - 2$ and $h =$

$\lceil \lg(N-2) \rceil$, respectively.

Next, set the number of nodes in the position WoORAM holding area to be

$$M_p = N_p \cdot h. \tag{3}$$

This ensures that only (at most) one Trie node needs to be refreshed to the position WoORAM main area when writing an entire path of h Trie nodes during a single logical write operation. (The number of Trie nodes written to the holding area for each operation is always h.) Based on these formulae, we need to have enough space in the $B/2$ bits of a half block to fit $h+1$ Trie nodes.

What remains is to estimate the size of each Trie node. Each node stores $b = 2$ DetWoORAM pointers, each of which contains $\lceil \lg M \rceil$ bits for the holding area position, $\lceil \lg B \rceil$ bits for the block offset, and 1 bit for the bit diff value. The condition that $h+1$ Trie nodes fit into $B/2$ bits then becomes

$$(h+1) \cdot (\lceil \lg M \rceil + \lceil \lg B \rceil + 1) \leq \frac{B}{2} \tag{4}$$

Combining this inequality with all of the previous settings for b, M, and M_p, and assuming a block size of 4096 bytes (so $B = 4096 \cdot 8 = 2^{15}$) as is the default in modern Linux kernels, we have

$$(\lceil \lg (N-2) \rceil + 1) \cdot (\lceil \lg N \rceil + 17) \leq 2^{14}.$$

That inequality is satisfied for values of N up to 6.6×10^{35}, which is *much* more than any conceivable storage size. Further tuning of the b and M parameters could be dome to achieve an even tighter packing and/or better read performance while maintaining 2 physical block writes per logical write.

3.5 Encryption Modes

The deterministic and sequential access pattern fits nicely with encryption of each block using *counter mode*. In particular, We encrypt each DetWoORAM block using AES encryption based on the number $i \| 0^{64}$ as a counter. Recall that the client maintains the global counter i (64-bit long). Assuming the block size B is reasonable (shorter than $2^{64} \cdot 16$ bytes), there will be no collision of IVs, and the security of encryption is guaranteed. We stress that we do not need space for storing IVs due to this optimization which cannot be applied to previous schemes. For example, the randomized procedures in schemes like HiVE-WoORAM, IVs must be stored separately on the server, adding to the communication cost overhead.

However, the physical blocks that store the position map Trie are encrypted with AES in CBC mode. When we pack multiple Trie nodes together, such as during interleaving or packing as described previously, we can encrypt a group of Trie nodes together in one shot using a single IV. Since Trie nodes are much smaller than B, we can place that IV for that group of nodes at the beginning of the block itself, thus avoiding an extra memory access on read or write.

We note that after packing the Trie nodes into blocks, the number of blocks in the main DetWoORAM is significantly larger than that in the Position-DetWoORAM, so that most of the data is encrypted using counter mode.

4 Analysis of DetWoORAM

We formally state the security (obliviousness), and communication complexity of DetWoORAM. Fortunately, the simplicity of the construction makes the proofs relatively straightforward in all cases.

Security proof First we state the security in terms of the definitions in Section 2.1.

Theorem 4.1. *DetWoORAM provides write-only obliviousness.*

Let $\vec{x}$ and $\vec{y}$ denote two sequences of operations in DetWoORAM that contain the same number of write operations.

The sequence of *locations* of physical writes is deterministic and does not in any way depend on the actual locations being written, and the *contents* of physical writes are encrypted using an IND-CPA symmetric cipher. Therefore ii it holds that

$$\mathsf{WOnly}(\mathsf{PhysicalAcc}^{\Pi}(\vec{x})) \quad \approx_c \quad \mathsf{WOnly}(\mathsf{PhysicalAcc}^{\Pi}(\vec{y})),$$

because the locations in these two access patterns are identical, and the contents in the access patterns are indistinguishable from random.

Communication complexity For the complexity analysis, assume that:

- the size ratio M/N is a constant,
- the branching factor b is a constant,
- the block size B is large enough to contain a single path of trie nodes, and
- the position map holding area is at least $O(\log N)$ times larger than the position map's main area.

Asymptotically this means that $B \in \Omega(\log^2 N)$. From a practical standpoint, even in the extreme case of storing a yottabyte of data (2^{80} bytes), with holding area size $M = N$, branching factor $b = 2$, and 4KB blocks (i.e., $B = 4096$), an entire path of trie nodes is still well below the block size at 1496 bytes.

Theorem 4.2. *Under the assumptions above, the number of physical block writes per logical block write in DetWoORAM is $O(1)$. Furthermore, the number of physical block reads per logical read or write operation is $O(\log N)$.*

Let $h \in O(\log_b N)$ for the height of the trie that stores the position map. A single read to DetWoORAM requires at most two block reads and one position map lookup, which requires fetching all h nodes in the Trie path to that position. Fetching a Trie node in the position WoORAM requires accessing the parent node as well, requiring at most $2h$ nodes need to be fetched. In the worst case every node might be packed in a different block, so this is $O(1 + h)$ physical block reads per logical read, which is $O(\log_b N)$.

A single write to DetWoORAM requires at most $1+\lceil M/N \rceil$ block writes to holding and main areas and one update to the position map Trie. Each main area refresh requires an additional block read and position map lookup. Because M/N is a constant, this is $O(1)$ block writes, $O(\log_b N)$ reads, and one trie update.

Updating a single node in the Trie involves first fetching the path to that node in $O(\log_b N)$ physical reads, then writing each node on that path, updating the pointers in the parent nodes from leaf back up to root. This requires h writes to the position map WoORAM, which from the assumptions will require h writes to the holding area plus $O(1)$ refreshes in position map WoORAM's main area. These $O(1)$ refreshes each require looking up $O(\log_b N)$ nodes in the position map WoORAM, for an additional reading cost of up to $O(\log N)$ physical blocks.

All together we get $O(\log_b N)$ physical reads per logical write, and $O(1)$ physical writes per logical write.

5 Implementation

We have implemented our DetWoORAM system, using the Trie-based position map, in an open source C++ library available at `https://github.com/detworam/detworam`. As we will show in this section, comparison benchmarks validate our theoretical results on the efficiency of DetWoORAM, showing it to be many times faster than the previous scheme of HiVE-WoORAM, and only a few times slower than a non-oblivious baseline.

Our library The library relies on BUSE (`https://github.com/acozzette/BUSE`) in order to write a block device driver in userspace, and uses the mbed TLS library (`https://tls.mbed.org/`) for encryption utilities. We also made extensive use of C++ templates in our implementation, which allows for considerable flexibility in choosing the parameters for the DetWoORAM and automatically tuning the performance at compile-time. For example, based on the size and number of backend storage blocks, the exact byte sizes needed to store pointers, relative proportion of data WoORAM to position map WoORAM, trie height, and relative main/holding area sizes will all be seamlessly chosen at compile time.

The implementation is exactly as described in the previous section, with a default Trie branching factor of $b = 64$ unless otherwise noted. The only exception is that we did *not* implement the full interleaving, but rather the packing solution within the position map WoORAM to pack trie nodes into single blocks. Two blocks at a time (from the position map holding and position map main areas) are held in memory while they are being filled sequentially, and then are written back to disk once filled. In total, the result is that rather than having a fully sequential access pattern as we would with full interleaving, we see 4 sequential write patterns in sub-regions of memory.

Comparisons We carefully re-implemented the HiVE-WoORAM (only the WoORAM part, not the hidden volume part), using the same BUSE/mbedTLS library setup. As in their original paper and implementation, our HiVE-WoORAM implementation uses $k = 3$ random physical writes per logical write, and makes use of a recursive position map. The original implementation was as a kernel module for a device mapper, but unfortunately due to Linux kernel changes this module is incompatible with recent Linux kernels. In fact, this incompatibility was part of our motivation to use only standards-compliant userspace C++ code for our DetWoORAM implementation.

	Sequential write		Sequential read	
	MB/sec	overhead	MB/sec	overhead
dm-crypt baseline				
SSD	505.4	—	615.8	—
HDD	111.6	—	126.1	—
HiVE-WoORAM				
SSD	2.6	192x	40.5	15.2x
DetWoORAM				
SSD, $M = 3N$, $b = 64$	**49.7**	10.2x	**260.0**	**2.37x**
SSD, $M = N$, $b = 64$	34.0	14.8x	244.1	2.52x
HDD, $M = 3N$, $b = 64$	29.0	**3.84x**	26.1	4.83x
HDD, $M = N$, $b = 64$	25.0	4.46x	24.1	5.24x

Logical disk size 40GB and block size 4KB in all cases.
Overhead is relative to the dm-crypt baseline for that drive type.
Highlighted values indicate the best WoORAM per column.

Table 2: bonnie++ benchmarking of sequential accesses

	Solid state SSD		Spinning platters HDD	
	MB/sec	overhead	MB/sec	overhead
dm-crypt baseline	154	—	16.4	—
HiVE-WoORAM	8.49	18x	0.051	325x
DetWoORAM	34.4	4.5x	10.2	1.6x

Logical disk size 40GB and block size 4KB in all cases.
Overhead is relative to the dm-crypt baseline for that drive type.
DetWoORAM used $M/N = 3$ and $b = 64$ for all cases.
Highlighted values indicate the best WoORAM per column.

Table 3: fio benchmarking of random reads and writes

For a baseline comparison, we wanted to use the best existing solution with the same general setup as ours. Our baseline uses the linux kernel module `dm-crypt`, which provides an encrypted block device with no obliviousness, connected to simple "passthrough" device that comes with the BUSE distribution. There is no obliviousness in this option; it simply encrypts/decrypts and stores the resulting ciphertext in the same location on disk. This provides a fair baseline to our work, and should help to eliminate any bottlenecks or artifacts of the BUSE layer in order to have a clear comparison with our new DetWoORAM protocol.

Measurement using bonnie++ Table 2 shows the results of running the popular `bonnie++` disk benchmarking tool our the plain encryption as well as different WoORAM settings. All tests were performed with a 40GB logical filesystem within a 200GB partition, using the `btrfs` filesystem.

We tested using 200GB partitions on a 1TB HDD (HGST Travelstar 7200RPM) and on a 256GB SSD (Samsung 850 Pro). We note that both drives are standard commodity disks available for around $100 USD. As expected, the SSD drive is considerably faster for both reading and writing.

Recall that one novel feature of DetWoORAM is that it can flexibly adapt to different storage ratios between logical and physical storage. We tested both with $M = N$, similar to the HiVE-WoORAM, and with more physical space of $M = 3N$, and observed a slight (but statistically meaningful) performance improvement from having more physical disk space (and therefore larger holding area in the DetWoORAM). We also tested with different branching factors ranging from $b = 2$ to $b = 512$, but did not notice any significant timing differences overall, indicating that the position map plays a smaller role in the overall performance.

Overall we can see that the DetWoORAM suffers only a 3x-10x slowdown compared to the baseline, whereas the HiVE-WoORAM is almost 200x slower in the case of writing and 15x slower for reading compared to the same baseline. The results for HiVE-WoORAM are consistent with the results reported in their original paper [BMNO14].

In fact, our DetWoORAM running on an SSD is in most cases *faster* than the baseline running on a spinning disk HDD, providing good evidence that our system is fast enough for practical use. We believe this is largely explained by the *sequential write pattern* of DetWoORAM, which also makes read operations *partially* sequential. For large sequential workloads, the data locality appears to have a very significant effect on performance.

Measurement using fio As has been noted in previous WoORAM works [BMNO14], performing sequential *logical* operations can put WoORAMs in an especially bad light, as the baseline non-oblivious storage will translate the sequential read/write operations to physically sequential addresses, thereby gaining significantly over WoORAMs that need to obscure the logical address of each operation.

Interestingly, our DetWoORAM is a somewhat "in-between" case here, as the write pattern is completely sequential, and the read pattern is *partially* sequential: the main area of storage corresponds exactly to physical addresses, but the holding area and position map do not. We used a second disk performance measurement tool `fio` (`https://github.com/axboe/fio`) in order to perform *random* reads and writes, as opposed to the sequential read/write pattern of the `bonnie++` benchmarks. The results are shown in Table 3, which shows the throughput for random reads and writes of 4KB-4MB sized blocks in direct access to the device without any filesystem mounted.

As expected, the performance degradation for HDD compared to SSD in all cases was significant for the random reads and writes. As with the bonnie++ benchmarks, but more dramatically here, our DetWoORAM running on an SSD outperformed the baseline running on the HDD. Even more surprisingly, on the HDD our DetWoORAM was only 1.6x slower than the baseline. This can be explained in part by the fact that our scheme actually *turns random writes into sequential writes*, so although it performs more writes than the baseline, they will be more compact in this experiment.

6 Insecurity of DataLair and FlatORAM

Two recent papers [CCS17, HvD16] have also proposed to improve the performance of HiVE-WoORAM. While these papers contain some new and promising ideas, and in particular [CCS17] proposed the use of a B-tree ODS similar to our Trie ODS for the position map, unfortunately they both violate the notion of write-only obliviousness.

Intuitively, both of these schemes identify that a bottleneck in HiVE-WoORAM is in identifying free blocks from the random blocks chosen, and propose to modify the random block choosing scheme in order to find free blocks more efficiently with fewer dummy writes. Unfortunately, this improvement in both cases leaks a small amount of information about which blocks are free or not, and thereby allows an adversary to distinguish between whether recent writes have been to the same address, or to different addresses. We formalize this notion and prove the insecurity of these schemes below.

6.1 Insecurity of DataLair

Let N be the number of logical blocks. DetaLair sets $2N$ to the number of physical blocks so that the number of free physical blocks is always N. In DataLair [CCS17, Section IV], every ORAMWrite considers two disjoint sets of k items:

- Free set S_0: A set of k blocks chosen randomly among the N free physical blocks.
- Random set S_1: A set of k blocks chosen randomly among the entire $2N$ physical blocks.

To make sure that S_0 and S_1 are disjoint, some elements may be removed and addional steps of sampling may be done. Based on the two sets, the ORAM writes a data block as follows:

```
ORAMWrite($d$): // $d$ is a data block
1. Insert $d$ in stash
2. Create two sets $S_0$ and $S_1$ as described above.
3. Choose $k$ blocks $U = \{u_1, \ldots, u_k\}$ as follows:
   For $i = 1$ to $k$:
     $b_i \leftarrow \{0, 1\}$, and fetch (and remove) $u_i$ from $S_{b_i}$.
     If $b_i = 0$ and stash is not empty:
       Take out a data item from stash and write it in $u_i$.
     Otherwise, reencrypt $u_i$.
```

We assume $N > 2k$. The actual scheme chooses a large N and $k = 3$.

Insecurity of the scheme We note that the access pattern of a single ORAMWrite is hidden. However, that alone is not sufficient to show the write obliviousness. In particular, security breaks down when one considers *multiple ORAMWrite operations.*

Observe that the above algorithm is more likely to choose a free block than a non-free block; with probability 1/2, a chosen block will be from S_0 and thereby always free, and with probability 1/2, a chosen block will be from S_1 and thereby sometimes free. This tendency towards choosing free blocks leaks information. To clarify our point, consider the following two sequences of logical writes:

$$\mathsf{seq}^0 = (write_0, write_0, write_2), \qquad \mathsf{seq}^1 = (write_0, write_1, write_2)$$

Here, $write_i$ denotes writing data to a logical address i. Let $U_i = (u_{i,1}, \ldots, u_{i,k})$ be the set of chosen blocks from the ith ORAMWrite. Let d_ℓ be the data in *logical block* ℓ.

Then, in seq^0, physical block $\gamma \in U_1$ containing d_0 will be probably freed up thanks to the second $write_0$, and the last $write_2$ may be able to choose γ as a free block. However, in seq^1, the block γ cannot be freed up by $write_1$, since γ contains d_0! So, the last $write_2$ can choose γ only as a non-free block. Due to the different probablity weights in choosing free blocks vs. non-free blocks, U_1 and U_3 have a slightly higher chance of overlapping in seq^0 than in seq^1, and security breaks down. To clarify our point, we give an attack in Appendix A.

6.2 Insecurity of FlatORAM

FlatORAM has a similar vulnerability to DataLair. The system has a new component called an occupancy map, Occ. For physical address u, $\mathrm{Occ}[u]$ is a bit indicating whether u is occupied or not. As with FlatORAM, set $2N$ to be the number of physical blocks. Their ORAM write algorithm proceeds as follows:

```
ORAMWrite at logical address $\ell$:
1. Let *old* be the old physical block holding data in logical address $\ell$.
2. Forever:
    (a) Choose a random physical block ID $u$.
    (b) If $\mathrm{Occ}[u] = 0$:
            $\mathrm{Occ}[u] := 1$, $\mathrm{Occ}[old] := 0$, Write data in $u$.
            Return.
        Else:
            Rewrite the same data in $u$ (i.e., dummy write).
```

Consider the following two sequences of logical writes:

$$\mathsf{seq}^0 = (write_0, write_0, write_2), \qquad \mathsf{seq}^1 = (write_0, write_1, write_2)$$

Let $U = (u_1, u_2, u_3 \ldots)$ be the access pattern resulting from ORAMWrites. Given an access pattern U, the adversary tries to tell if it is from seq^0 or seq^1 as follows:

- If $|U| = 3$ and $u_1 = u_3$ output 0; otherwise output a random bit.

Let $p^b = \Pr[|U| = 3, u_1 = u_3]$ from seq^b. We show that $p^0 - p^1$ is non-negligible, which proves that the adversary is a good distinguisher.

Consider seq^0 first. We have $\Pr[|U| = 3] = (1/2)^3$; that is, u_1, u_2, u_3 are all free blocks. Note u_1 becomes free after the second $write_0$ in seq^0. Therefore, we have

$$p^0 = 1/8 \cdot \Pr[u_1 = u_3 |\; |U| = 3] = 1/8N$$

Now consider seq^1. We have $\Pr[|U| = 3] = (1/2)^3$. However, u_1 can never be freed here, since subsequent operations are all $write_1$'s. Therefore, we have

$$p^1 = 1/8 \cdot \Pr[u_1 = u_3 |\; |U| = 3] = 0$$

7 Related Work

Oblivious RAM (ORAM) and applications. ORAM protects the access pattern so that it is infeasible to guess which operation is occurring and on which item. Since the seminal work by Goldreich and Ostrovsky [GO96], many works have focused on improving efficiency and security of ORAM (for example [SvDS$^+$13, RFK$^+$15, MMB15, SZA$^+$16] just to name a few; see the references therein).

ORAM plays as an important tool to achieve secure cloud storage [SS13b, SS13a, MBC14] and secure multi-party computation [GKK$^+$12, LHS$^+$14, LWN$^+$15, WCS15, ZWR$^+$16] and secure processors [FRK$^+$15, LHM$^+$15, NFR$^+$17]. There also have been works to hide the access pattern of protocols accessing individual data structures, e.g., maps, priority queues, stacks, and queues and graph algorithms on the cloud server [Tof11, BSA13, WNL$^+$14, RAC16]. The work of [JMTS16] considers obliviousness in the P2P content sharing system.

Write-only obliviousness Blass et al. [BMNO14] considers write-only ORAM (WoORAM), and gave a WoORAM construction much more efficient than the traditional ORAM constructions. They applied WoORAM to deniable storage scenarios and gave a WoORAM-based construction of hidden volume encryption (HiVE). Aviv et al. [ACMR17] gave a construction of oblivious synchronization and backup for the cloud environment. They observed that write-only obliviousness is sufficient for the scenario, since the client stores a complete local copy of his data, and therefore *read* accesses are naturally hidden from the adversary.

Deniable storage Anderson et al. [ANS98] proposed staganography-based approaches, that is, hiding blocks within cover files or random data. There are works based on his suggestion [MK99, PTZ03], but they don't allow deniability against a snapshot adversary.

Another approach is hidden volumes. Unfortunately, existing solutions such as TrueCrypt (discontinued now) [Tru17], Mobiflage [SM14] and MobiPluto [CWCZ15] are secure only against a single-snapshot adversary. HIVE [BMNO14] provides security even against a multiple-snapshot adversary. DEFY [PGP15] is the deniable log-structured file system specifically designed for flash-based, solidstate drives; although it is secure against a multiple-snapshot adversary, it doesn't scale well.

8 Conclusion

We presented DetWoORAM, a stash-free, deterministic write-only oblivious ORAM with sequential write patterns. This scheme achieves asymptotic improvement in write communication costs, $O(B \log N)$, requiring exactly $2B$ physical writes per logical write. We further showed that prior schemes to improve on HiVE-WoORAM are insecure. Finally, we implemented and evaluated DetWoORAM, and, for sequential writing, it incurs only a 3-4.5x overhead on HDD and 10-14x on SSD compared to using encryption only. It is 19x faster than HiVE-WoORAM, the previous best, secure scheme. It is also practical; the theoretical write complexity is optimal, and DetWoORAM with an SSD backend has similar (sometimes better) performance compared to using encryption only on a spinning-platter HDD in a similar price range.

References

[ACMR17] Adam J Aviv, Seung Geol Choi, Travis Mayberry, and Daniel S Roche. ObliviSync: Practical oblivious file backup and synchronization. In *Network and Distributed System Security Sympoium*, NDSS'17, 2017.

[ANS98] Ross J. Anderson, Roger M. Needham, and Adi Shamir. The steganographic file system. In *Information Hiding, Second International Workshop, Portland, Oregon, USA, April 14-17, 1998, Proceedings*, pages 73–82, 1998.

[BMNO14] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious RAM. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 203–214, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.

[BSA13] Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. Data-oblivious graph algorithms for secure computation and outsourcing. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng, editors, *ASIACCS 13*, pages 207–218, Hangzhou, China, May 8–10, 2013. ACM Press.

[CCS17] Anrin Chakraborti, Chen Chen, and Radu Sion. DataLair: Efficient block storage with plausible deniability against multi-snapshot adversaries. In *Proceedings on Privacy Enhancing Technologies*, volume 3, pages 175–193. De Gruyter, 2017.

[CWCZ15] Bing Chang, Zhan Wang, Bo Chen, and Fengwei Zhang. Mobipluto: File system friendly deniable storage for mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 381–390, 2015.

[FRK$^+$15] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 103–116, 2015.

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

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

[HvD16] Syed Kamran Haider and Marten van Dijk. Flat ORAM: A simplified write-only oblivious RAM construction for secure processor architectures. *CoRR*, abs/1611.01571, 2016.

[JMTS16] Yaoqi Jia, Tarik Moataz, Shruti Tople, and Prateek Saxena. Oblivp2p: An oblivious peer-to-peer content sharing system. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 945–962, 2016.

[LD13] Lichun Li and Anwitaman Datta. Write-only oblivious RAM based privacy-preserved access of outsourced data. Cryptology ePrint Archive, Report 2013/694, 2013. http://eprint.iacr.org/2013/694.

[LHM+15] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 87–101, 2015.

[LHS+14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient RAM-model secure computation. In *2014 IEEE Symposium on Security and Privacy*, pages 623–638, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[MBC14] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining ORAM and PIR. In *NDSS 2014*, San Diego, CA, USA, February 23–26, 2014. The Internet Society.

[MK99] Andrew D. McDonald and Markus G. Kuhn. Stegfs: A steganographic file system for linux. In *Information Hiding, Third International Workshop, IH'99, Dresden, Germany, September 29 - October 1, 1999, Proceedings*, pages 462–477, 1999.

[MMB15] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication ORAM with small blocksize. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 862–873, Denver, CO, USA, October 12–16, 2015. ACM Press.

[NFR+17] Kartik Nayak, Christopher Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. Hop: Hardware makes obfuscation practical. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[PGP15] Timothy Peters, Mark Gondree, and Zachary N. J. Peterson. DEFY: A deniable, encrypted file system for log-structured storage. In *NDSS 2015*, San Diego, CA, USA, February 8–11, 2015. The Internet Society.

[PS15] Kenneth G. Paterson and Mario Strefler. A practical attack against the use of RC4 in the HIVE hidden volume encryption system. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *ASIACCS 15*, pages 475–482, Singapore, April 14–17, 2015. ACM Press.

[PTZ03] HweeHwa Pang, Kian-Lee Tan, and Xuan Zhou. Stegfs: A steganographic file system. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 657–667, 2003.

[RAC16] Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi. A practical oblivious map data structure with secure deletion and history independence. In *2016 IEEE Symposium on Security and Privacy*, pages 178–197, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

[RFK$^+$15] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 415–430, 2015.

[SM14] Adam Skillen and Mohammad Mannan. Mobiflage: Deniable storage encryptionfor mobile devices. *IEEE Trans. Dependable Sec. Comput.*, 11(3):224–237, 2014.

[SS13a] Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy*, pages 253–267, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press.

[SS13b] Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious distributed cloud data store. In *NDSS 2013*, San Diego, CA, USA, February 24–27, 2013. The Internet Society.

[SvDS$^+$13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 299–310, Berlin, Germany, November 4–8, 2013. ACM Press.

[SZA$^+$16] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. TaoStore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy*, pages 198–217, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

[Tof11] Tomas Toft. Brief announcement: Secure data structures based on multi-party computation. In Cyril Gavoille and Pierre Fraigniaud, editors, *30th ACM PODC*, pages 291–292, San Jose, CA, USA, June 6–8, 2011. ACM.

[Tru17] TrueCrypt. (discontinued). 2017. http://www.truecrypt.org/.

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 850–861, Denver, CO, USA, October 12–16, 2015. ACM Press.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 215–226, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.

[ZWR+16] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy*, pages 218–234, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

A Insecurity of DataLair

Let N be the number of logical blocks. DetaLair sets $2N$ to the number of physical blocks so that the number of free physical blocks is always N. In DataLair [CCS17, Section IV], every ORAMWrite considers two disjoint sets of k items:

- Free set S_0: A set of k blocks chosen randomly among the N free physical blocks.
- Random set S_1: A set of k blocks chosen randomly among the entire $2N$ physical blocks.

To make sure that S_0 and S_1 are disjoint, some elements may be removed and addional steps of sampling may be done. Based on the two sets, the ORAM writes a data block as follows:

```
ORAMWrite($d$): // $d$ is a data block
1. Insert $d$ in stash
2. Create two sets $S_0$ and $S_1$ as described above.
3. Choose $k$ blocks $U = \{u_1, \ldots, u_k\}$ as follows:
   For $i = 1$ to $k$:
     $b_i \leftarrow \{0, 1\}$, and fetch (and remove) $u_i$ from $S_{b_i}$.
     If $b_i = 0$ and stash is not empty:
       Take out a data item from stash and write it in $u_i$.
     Otherwise, reencrypt $u_i$.
```

We assume $N > 2k$ and $k \geq 3$. The actual scheme chooses a large N and $k = 3$.

Insecurity of the scheme We note that the access pattern of a single ORAMWrite is hidden. However, that alone is not sufficient to show write obliviousness. In particular, security breaks down when one considers *multiple ORAMWrite operations*.

Observe that the above algorithm is more likely to choose a free block than a non-free block; with probability 1/2, a chosen block will be from S_0 and thereby always free, and with probability 1/2, a chosen block will be from S_1 and thereby sometimes free. This tendency towards choosing free blocks leaks information. To clarify our point, consider the following two sequences of logical writes:

$$\mathrm{seq}^0 = (init, w_0, w_0, w_2), \qquad \mathrm{seq}^1 = (init, w_0, w_1, w_2)$$

Here, w_i denotes writing data to a logical address i, and *init* is a sequence of operations that makes the ORAM have exactly N free blocks.[2]

Let $U_i = (u_{i,1}, \ldots, u_{i,k})$ be the set of chosen blocks from the ith ORAMWrite after the *init* sequence. Let d_ℓ be the data in *logical block* ℓ.

Then, in seq^0, physical block $\gamma \in U_1$ containing d_0 will be probably freed up thanks to the second w_0, and the last w_2 may be able to choose γ as a free block. However, in seq^1, the block γ cannot be freed up by w_1, since γ contains d_0! So, the last w_2 can choose γ only as a non-free block. Due to the different probablity weights in choosing free blocks vs. non-free blocks, U_1 and U_3 are more likely to overlap in seq^0 than in seq^1, and security breaks down.

To clarify our point, we give an attack. Given an access pattern (U_1, U_2, U_3), the adversary tries to tell if it is from seq^0 or seq^1. Consider the following events:

- X: $u_{1,1} \notin U_2$, $\quad$ Y: $u_{1,1} \in U_3$, $\quad$ E: $X \wedge Y$

The adversary works as follows:
Ouptut 0 if E takes place; otherwise output a random bit.

Let $p^b = \Pr[E]$ from seq^b. We show that $p^0 - p^1$ is non-negligible, which proves that the adversary is a good distinguisher.

Let $F_i(u)$ denote a predicate indicating whether a physical block u was free when the ith ORAMWrite starts. Note that whether $u_{1,1}$ belongs U_3 ultimately depends on $F_3(u_{1,1})$. In particular, for any $u_{1,1}$, we have

$$\begin{aligned} q_y &= Pr[Y \mid F_3^b(u_{1,1})] = \frac{1}{2} \cdot \frac{k}{N} + \frac{1}{2} \cdot \frac{k}{2N-k} \\ q_n &= Pr[Y \mid \neg F_3^b(u_{1,1})] = \frac{1}{2} \cdot \frac{k}{2N-k} \end{aligned}$$

[2] Their ORAM seems to be initialized with exactly N free blocks, in which case $init$ contains no operation. If that's the case, we can set the $init$ sequence as follows:

$$init = (w_0, \ldots, w_{N-1}, \underbrace{w_0, \ldots, w_0}_{\lambda \text{ times}}),$$

where λ is the security parameter. Note that after the $init$ sequence, the ORAM will have exactly N non-free physical blocks and N free physical blocks with probability least $1 - \mathrm{negl}(\lambda)$. So, we can safely ignore this negligible probability, and proceed our argument assuming that the ORAM has exactly N free blocks after the $init$ sequence.

The following table shows how $F_3(u_{1,1})$ depends on the previous events. In the table, $D_2(u) \in \{f, d^*_0, \ldots, d^*_{N-1}, d_0\}$ denotes a random variable indicating which logical block a physical block u contains when the second ORAMWrite starts. If the value is f, it means the block is free, and d^*_ℓ is the initial data for the logical block ℓ that the ORAM initialization procedure used. The value d_0 denotes the data block used in the first w_0 operation in seq^0 and seq^1. Let $S_i(\ell)$ denote a predicate indicating whether a logical block ℓ is in the stash when the ith ORAMWrite starts. In addition, FreeSet$_i$ denotes a predicate indicating whether the ith ORAMWrite found a physical block in the free set S_0 (thereby succesfully writing the input logical block in the free physical block).

case	$D_2(u_{1,1})$	$S_2(d_0)$	FreeSet$_2$	$F_3(u_{1,1})$	$F_3(u_{1,1})$
			(cond. on X)	(seq^0)	(seq^1)
$c1$	f	x	x	1	1
$c2$	d_0	0	0	0	0
$\boxed{c3}$	d_0	0	1	1	0
$c4$	$d^*_{\geq 0}$	x	0	0	0
$c5$	d^*_0	1	1	1	1
$\boxed{c6}$	d^*_1	x	1	0	0 or 1
$c7$	$d^*_{\geq 2}$	x	1	0	0

For example, in case $c3$,

- $D_2(u_{1,1}) = d_0$: When the second ORAMWrite begins, the physical block $u_{1,1}$ contains the logical block d_0.
- $S_2(d_0) = 0$: When the second ORAMWrite begins, the stash is empty.
- $(\text{FreeSet}_2|X) = 1$: The second ORAMWrite found at least one block in the free set S_0.
- For seq^0, the second write is w_0. From FreeSet$_2 = 1$, a new d_0 from w_0 will be written in a free block, and $u_{1,1}$ containing the old d_0 is freed.
- For seq^1, the second write is w_1. From FreeSet$_2 = 1$, a new d_1 from w_1 will be written in a free block, but $u_{1,1}$ containing d_0 is not affected.

Note that the first ORAMWrite in both seq^0 and seq^1 is the same with w_0, so $D_2(u_{1,1})$ and $S_2(d_0)$ is identically distributed for both seq^0 and seq^1. Moreover, observe that the distribution of X depends on only $u_{1,1}$ because the ORAMWrite samples U_2 at random. Finally, Pr[FreeSet$_2$] is always the same, since the number of free blocks in the second ORAMWrite is always the same with N.

Based on the table and the above observation, we have the following:

$$\begin{aligned}
p^0 - p^1 &\geq \Pr[c3 \wedge X](q_y - q_n) - \Pr[c6 \wedge X](q_y - q_n) \\
&\geq \frac{k}{2N} \cdot \big(\Pr[c3 \wedge X] - \Pr[c6 \wedge X]\big) \\
&= \frac{k}{2N} \cdot \Pr[X] \cdot \big(\Pr[c3 \wedge X] - \Pr[c6 \wedge X]\big) \\
&\geq \frac{k}{4N} \cdot \big(\Pr[c3|X] - \Pr[c6|X]\big)
\end{aligned}$$

Now, let's first calculate the lowerbound on $\Pr[c3|X]$. If the first ORAMWrite chooses at least one block from the free set and writes d_0 in $u_{1,1}$, it must be $D_2(u_{1,1}) = d_0$ and $S_2(d_0) = 0$. Therefore,

$$\Pr[D_2(u_{1,1}) = d_0, S_2(d_0) = 0] \geq \frac{1}{2} \cdot \frac{1}{k}.$$

Moreover, at least probability $\frac{1}{2}$, the second ORAMWrite will find a block from the freeset, which implies that

$$\Pr[c3|X] \geq \frac{1}{4k}.$$

To calculate the upperbound on $\Pr[c6|X]$, observe that $D_2(u_{1,1}) = d_1^*$ implies that $u_{1,1}$ contained d_1^* even before the first ORAMWrite w_0. Therefore, we have

$$\Pr[c6|X] \leq \Pr[\ u_{1,1} \text{ has } d_1^* \text{ before the 1st ORAMWrite}] = \frac{1}{2N}.$$

Therefore, we have

$$p^0 - p^1 \geq \frac{k}{4N} \cdot \left(\frac{1}{4k} - \frac{1}{2N}\right) = \frac{N-2k}{4N^2}.$$