Oblivious RAM with Small Storage Overhead

Michael Raskin[1] and Mark Simkin[2]

[1] LaBRI, Université de Bordeaux, raskin@mccme.ru
[2] Aarhus University, simkin@cs.au.dk

Abstract. In this work, we present a new approach to constructing Oblivious RAM (ORAM). Somewhat surprisingly, and despite the large amount of research interest that ORAM has received, all existing ORAM constructions are based on a handful of conceptually different approaches. We believe it is of theoretical and practical interest to explore new ways to construct this primitive.
Our first construction is conceptually extremely simple and has a worst-case bandwidth overhead of $\mathcal{O}\left(\sqrt{N}\frac{\log N}{\log\log N}\right)$, where N is the number of data blocks.
Our main construction, Lookahead ORAM, has a worst-case bandwidth overhead of $\mathcal{O}\left(\sqrt{N}\right)$ and an additive storage overhead of $\sqrt{2N}$, which is the smallest concrete storage overhead among all existing ORAM constructions with sublinear worst-case bandwidth overhead. A small storage overhead is particularly beneficial in outsourced storage settings, where data owners have to pay their storage provider for the amount of storage they consume.
In addition to having a small storage overhead, Lookahead ORAM has perfect correctness, i.e. every operation on the ORAM data structure succeeds with probability 1 and, assuming the client stores some small stash of data locally, an online bandwidth overhead of $\mathcal{O}(1)$ without server-side computation. In comparison with prior work that has sublinear worst-case bandwidth overhead, Lookahead ORAM is asymptotically the most efficient ORAM with perfect correctness.

1 Introduction

Over the past years, more and more sensitive data is stored online. A basic attempt to keep data private while storing it online on an untrusted server is to simply encrypt each data entry. Unfortunately, this is not always sufficient. For instance, Islam et. al. [IKK12] showed that by observing the access patterns induced by encrypted search queries over an encrypted database, an honest-but-curious[3] server storing the database, could learn significant amounts of information about the queries' contents. A more viable approach is to not only encrypt the data, but also hide the access patterns. Oblivious RAM (ORAM) is a cryptographic primitive that allows a client to do exactly this, at the cost of some bandwidth and storage overhead. It enables a client to outsource his data to an untrusted server inside of an ORAM data structure, and then read and write to this dataset without leaking the position that was accessed or the operation that was performed. After ORAM was first introduced by Goldreich and Ostrovsky [Gol87, GO96], much work [OS97, GMOT11, WCS15, MMB15, RFK+15, GMP16, SSS12] has been done to construct ORAM with asymptotically and practically good efficiency measures.

In this paper, we introduce two new ORAM constructions and make the following contributions:

Novel approach to constructing ORAM. We present a fundamentally new approach to constructing ORAM. Somewhat surprisingly, and despite the large amount of research interest that ORAM has received, all existing constructions are based on a handful of conceptually different approaches. We believe it is of theoretical and practical interest to explore new ways to construct this primitive.

Our first construction is extremely simple and *probabilistically* correct in the sense that it works as intended with an overwhelming probability and operations on the ORAM data structure only fail with a negligible probability. It has a worst-case bandwidth overhead of $\mathcal{O}\left(\sqrt{N}\frac{c+\log t+\log N}{\log\left(c+\log t+\log N\right)}\right)$, where N is the

[3] A honest-but-curious server follows the protocol specification, but tries to extract as much private information as possible from the observed communication.

number of data blocks to be stored, t is an upper bound on the number of accesses, and c is the correctness parameter that provides an upper bound of 2^{-c} on the failure probability. To the best of our knowledge, it is one of the conceptually simplest known ORAM constructions to date. The underlying logic is easy to implement and the proof of security is straightforward. Due to its simplicity, we believe it is an interesting open research direction to construct more complex primitives, like garbled RAM [LO13, GHL+14], with practical efficiency based on this ORAM construction.

Our main construction, called Lookahead ORAM, is loosely based on our first construction. It has a worst-case bandwidth overhead of $\mathcal{O}\left(\sqrt{N}\right)$ and is *perfectly* correct, in the sense that every operation on the ORAM data structure succeeds with probability 1. It is the asymptotically most efficient perfectly correct ORAM construction with respect to bandwidth overhead.

Small concrete storage overhead. Lookahead ORAM has the smallest concrete storage overhead among *all* existing ORAM constructions with sublinear worst-case bandwidth overhead. Our construction only incurs an *additive* storage overhead of $\sqrt{2N}$ and in Section 4.5 we show it can be reduced even further at the cost of slightly increasing the constants in the bandwidth overhead. A small storage overhead is particularly beneficial in outsourced storage settings where data owners have to pay their storage provider for the storage they consume. Although constructions with asymptotically smaller bandwidth overhead exist, they usually incur a rather large storage overhead. For example, in the case of the popular Path-ORAM [SvDS+13] construction, which has a bandwidth overhead of only $\mathcal{O}\left(\log^2 N\right)$, the storage overhead for storing N data blocks, is $20N$ large[4]. The more storage efficient (deamortized) square-root ORAM [GMOT11] construction has a storage overhead of $2\left(N + 2\sqrt{N}\right)$, which is still over two times as expensive as our construction, while at the same time being asymptotically slower in terms of bandwidth overhead. Lookahead ORAM has a small storage overhead and at the same time it provides reasonable theoretical and practical efficiency guarantees with respect to its bandwidth overhead. For a comparison of the most relevant existing schemes and our construction with respect to storage overhead, see Table 1.

Constant online bandwidth overhead. In the original work of Goldreich and Ostrovsky [Gol87, GO96], the authors showed, roughly speaking, a lower bound of $\Omega(\log N)$ on the bandwidth overhead of a large class of ORAM constructions. One approach to circumvent the Goldreich and Ostrovsky lower bound was introduced in [BMP11] and then improved upon in [DSS14, RFK+15]. Their main idea was to split the total bandwidth overhead into two parts. The first part, the so called online overhead, is the amount of data that needs to be transmitted between the client and the server to retrieve a desired data element obliviously. The second part, the offline overhead, is the amount of data that needs to be transmitted between the two parties to ensure obliviousness of future accesses. One can think of the offline overhead as background work that, usually, moves around encrypted data elements in the ORAM data structure to ensure the desired obliviousness guarantees. Splitting the total bandwidth overhead this way and then minimizing the online overhead has practical advantages. It allows the client to efficiently retrieve data from the server without much latency during bursts of requests and then do the background work during quieter phases.

In [BMP11], Boneh et al. presented a solution, for a primitive strongly, which is related to ORAM, that has $\mathcal{O}(\log N)$ online and $\mathcal{O}\left(\sqrt{N}\log N\right)$[5] worst-case overhead. In [DSS14, RFK+15] this idea of splitting the total overhead has been further refined, and constructions that achieve an online bandwidth overhead of $\mathcal{O}(1)$ are presented. However, these constructions require some server-side computation during the online phase, which renders these solutions not applicable for "raw" storage providers that do not support these ORAM constructions explicitly. That is, our construction works in combination with arbitrary storage providers

[4] Although the authors experiment with smaller storage overheads in their evaluation section, the theoretical correctness guarantees only hold for the stated storage overhead of $20N$

[5] This worst-case complexity is slightly different from the original paper. The paper has a superlinear worst-case overhead due to an expensive reshuffling phase, but when splitting shuffling over $\sqrt{N}$ accesses, one can achieve the stated complexity

like Dropbox or Google Drive, whereas the constructions from [DSS14, RFK$^+$15] only work with storage providers that explicitly implement their given scheme.

Lookahead ORAM, assuming the client stores a (practically small) position map and a stash that can hold up to $\sqrt{2N}$ data blocks locally, allows the client to obliviously retrieve elements from the ORAM data structure with no bandwidth overhead and no server-side computation in the online phase. That is, in the online phase, the client can directly download the desired elements from the server.

To provide a better feeling for how expensive it is to store the stash and the position map locally, consider a 1GB database with a block size of 1KB. To be able to make use of our online overhead feature, the client would need to store a roughly 1.42MB stash and a 8MB position map locally. If the client chooses to not use the minimizing online overhead feature, it can reduce its persistent storage to $\mathcal{O}(1)$.

Implementation and Evaluation. We implemented Lookahead ORAM in C using libsodium [BLS12] to validate our calculation of specific constants in the Lookahead ORAM construction. We compare its performance to the relevant related work in terms of required server-side storage and bandwidth overhead per access. We show that Lookahead ORAM requires the least storage on the server-side, and is even competitive with respect to the bandwidth overhead per access for a large range of parameters. A more detailed description of our implementation and our performance benchmarks, can be found in Section 6.

Attack on [GMSS16]. We identify a flaw in the recent ORAM construction of [GMSS16] and outline an attack that breaks the claimed obliviousness guarantees in Section A. We have contacted the authors and they have acknowledged our attack.

1.1 Other Related Work

Construction	**Server Storage**	**Bandwidth Online**	**Bandwidth Total**	**Correctness**
Square-root [GMOT11]	$2\left(N+2\sqrt{N}\right)$	$\mathcal{O}(1)$	$\mathcal{O}\left(\sqrt{N}\log N\right)$	perfect
Hierarchical [GMOT11]	$>10N$	–	$\mathcal{O}\left(\log^3 N\right)$	probabilistic
Path-ORAM [SvDS$^+$13]	$20N$	–	$\mathcal{O}\left(\log^2 N\right)$	probabilistic
Parallel Buffers [SSS12]	$3N$	$\mathcal{O}(1)$	$\mathcal{O}\left(\sqrt{N}\right)$	probabilistic
Lookahead ORAM	$N+\sqrt{2N}$	$\mathcal{O}(1)$	$\mathcal{O}\left(\sqrt{N}\right)$	perfect

Fig. 1. Comparison of the most relevant fundamentally different ORAMs and their worst-case performance metrics. For simplicity, we assume that clients store a small stash and the position map locally. Concrete storage overhead is either directly reported from the paper or conservatively estimated. The storage overhead of Lookahead ORAM can be made even smaller than what is reported in the table as is explained in Section 4.5.

In [Gol87, GO96], Goldreich and Ostrovsky proposed two constructions of ORAM, called square-root and hierarchical ORAM, which both have sublinear amortized bandwidth overhead, but suffer from (super)linear worst-case bandwidth overheads. In subsequent work [OS97, GMOT11], it was shown how to deamortize both constructions, i.e. how to make their amortized overhead be their worst-case overhead, at the cost of roughly doubling server's storage overhead.

Another important result given in [Gol87, GO96] was a lower bound of $\Omega(\log N)$ on the number of accesses that are needed to simulate one oblivious access for a large class of ORAM constructions. This bound has been further examined in [BN16]. Constructing an ORAM that matches this lower bound has been of large interest. In a seminal result, Shi et al. [SCSL11] introduced a new approach for constructing ORAM based on binary trees, which has a worst-case overhead of $\mathcal{O}\left(\log^3 N\right)$ and is computationally as well as conceptually simpler than previous works. In Path-ORAM [SvDS$^+$13] this overhead has been further reduced to $\mathcal{O}\left(\log^2 N\right)$. Due

to their simplicity and efficiency, [SCSL11, SvDS+13] have been the basis for many theoretical and practical follow-up works [CP13, WCS15, MMB15, RFK+15, GMP16, DDF+16, SSS12].

In order to achieve practical efficiency and overcome the $\Omega(\log N)$ lower bound, several works have looked at different refinements of the classical ORAM notion in the client server model. Path-PIR [MBC14] uses server-side computations to achieve a practically very small, yet still poly-logarithmic bandwidth overhead. In [AKST14], Apon et al. formally define the notion of Verifiable Oblivious Storage, which generalizes the notion of ORAM by allowing the server to perform computations, and show that the ORAM lower bound does not apply to their setting by providing a scheme with constant overhead per access based on Fully Homomorphic Encryption. In [DDF+16] a scheme, called Onion ORAM, is presented that breaks the lower bound, but only relies on additively homomorphic encryption. In this work we will only focus on the classical notion of ORAM that does *not* allow server-side computation. Apart from being theoretically interesting, this has also practical advantages. An efficient ORAM construction that does not require server-side computations, can be deployed on any "raw" cloud storage provider. This is not the case for solutions that require the server to implement some ORAM specific logic. In Table 1 we compare our solution to the main alternative ORAM approaches that do not requires server-side computation and have sublinear worst-case overheads.

Another approach, called Parallel Oblivious RAM [BCP16, CLT16, SZEA+16], aims at constructing ORAM schemes that support parallel accesses. That is, several client requests can be executed at the same time without losing obliviousness guarantees. We leave at as an interesting open question to extend our solutions to this setting.

Lastly, a recent work by Gordon et al. [GMSS16] presents an ORAM construction that may seem *superficially* similar to ours. However, our work significantly differs from theirs in terms of performance guarantees we achieve, underlying ideas we present, and security we obtain. Their construction, called Matrix-ORAM, arranges the data elements in a fixed number of rows. The size of each row linearly depends on the size of the total database and each row has its own stash. Accesses to their data structure are performed in a conceptually and concretely different manner to ours. The authors claim a logarithmic bandwidth overhead. In contrast, our data structures have a rectangular shape [6], and have a bandwidth overhead of roughly $\mathcal{O}\left(\sqrt{N}\right)$. We discovered a flaw in their construction and present a concrete attack on their scheme. This flaw is discussed in detail in Section A.

2 Preliminaries

On a high level, the ORAM security definition assumes a honest-but-curious server and says that for any two data access sequences, the corresponding access sequences to the ORAM data structure should be indistinguishable. The security definition is taken almost verbatim from [SSS12].

Definition 1. *(Security Definition) Let*

$$((\mathsf{op}_1, \mathsf{a}_1, \mathsf{data}_1), \ldots, (\mathsf{op}_M, \mathsf{a}_M, \mathsf{data}_M))$$

be a data request sequence of length M, where each op_i is either a $\mathsf{read}(\mathsf{a}_i)$ or a $\mathsf{write}(\mathsf{a}_i, \mathsf{data})$ operation. Let $\mathsf{oram}(\overrightarrow{y})$ denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests $\overrightarrow{y}$. An ORAM construction is said to be secure if for any two data request sequences $\overrightarrow{y}$ and $\overrightarrow{z}$ of the same length, their access patterns $\mathsf{oram}(\overrightarrow{y})$ and $\mathsf{oram}(\overrightarrow{z})$ are computationally indistinguishable and the construction is correct in the sense that it returns on input $\overrightarrow{y}$ data that is consistent with $\overrightarrow{y}$ with probability at least $1 - 2^{-c}$. We call c the correctness parameter.

Position Map. All known ORAM schemes need to maintain a position map of size $\mathcal{O}(N)$ that keeps track of the ordering of elements inside the ORAM data structure on the server. For the sake of simplicity we will assume that the client stores the full position map locally. From a practical point of view, this

[6] One may even say they look matrix shaped

seems to be a reasonable assumption in many client-server settings. For example, the position map of a 1GB database containing 1KB blocks is only around 8MB large. From a theoretical point of view, to reduce the client's persistent storage to $\mathcal{O}(1)$, both of our constructions can be combined with the well-known approach of recursively storing the position map in a sequence of smaller ORAMs, which was first introduced in [SCSL11]. Recursively storing the position map on the server increases the number of round-trips per access to $\mathcal{O}(\log N)$, but it does not change the asymptotic bandwidth overheads of our constructions. We explain how to combine our main construction with the recursive approach in more detail in Section 5.

Block Size. If we want to use the recursive ORAM approach mentioned above to store the position map on the server-side, then the data blocks need to be $\Omega(\log N)$ large. In the setting, where the client stores the position map locally, we do not make any assumptions about the data block size. However, for the construction to be useful, the data blocks on the server should be larger than the position map that the client stores locally.

Integrity. The ORAM security definition assumes the server to be honest-but-curious. Similar to previous works [SvDS+13], our construction can be extended to prevent tampering of an actively malicious server by using a Merkle Tree on top of our ORAM data structures.

3 A Simple Matrix Bucket ORAM

In this section, we will present a very simple oblivious RAM construction with reasonable efficiency and a straightforward proof of security. To the best of our knowledge this is one of the, arguably, simplest ORAM constructions known to date. Apart from being interesting on its own, it will also serve a stepping stone towards our main construction by introducing some of the ideas behind our main approach.

Initially we are given an array A of length N of data elements. To initialize our scheme, we create a empty $\sqrt{N} \times \sqrt{N}$ matrix C, in which each matrix cell is a bucket of size w. We randomly (and independently) assign each element from A to a bucket in C. Once all elements from A are distributed among buckets in C, we encrypt each bucket separately, and store the matrix on the server. For the sake of simplicity, we assume that the client stores a position map σ that maps indices of elements from A to columns of C locally.

Init(A)

```
1 C ← initBucketMatrix()
2 σ ← initPosMap()
3 for ℓ = 1 ... N do
4 |   (i, j) ← pickRandBucket()
5 |   C[i, j].put(A[ℓ] ‖ ℓ)
6 |   σ(ℓ) = j
7 C ← encryptBuckets(C)
```

Access(ℓ, x)

```
1  c ← readColumn(C, σ(ℓ))
2  (i, j) ← pickRandBucket()
3  r ← readRow(C, i)
4  r_dec ← decrypt(r)
5  c_dec ← decrypt(c)
6  data ← popData(ℓ, c_dec)
7  putData(x‖ℓ, r_dec, j)
8  σ(ℓ) = j
9  c* ← encrypt(c_dec)
10 r* ← encrypt(r_dec)
11 writeBack(c*, r*)
12 return data
```

Fig. 2. Pseudocode of simple Matrix ORAM construction

To obliviously access some element with index ℓ in A, we access column $\sigma(\ell)$ in C. In addition, we pick a uniformly random bucket (i, j) in the matrix and obtain row i. We find the element with index ℓ in the retrieved column and perform our desired operation (read or write). We then remove element ℓ from its current bucket, put it into bucket (i, j), re-encrypt all retrieved buckets, and write back the retrieved row

and column. Lastly, we update the position map to point to the new column that stores ℓ, i.e., set $\sigma(\ell) = j$. The pseudocode implementing this construction is given in Figure 2

3.1 Security

We prove the following theorem.

Theorem 1. *Let $\mathcal{E} = (\mathsf{gen}, \mathsf{encrypt}, \mathsf{decrypt})$ be an* IND-CPA *secure encryption scheme. Then the construction in Figure 2 is a secure ORAM scheme with $\mathcal{O}\left(\sqrt{N}\frac{c+\log t+\log N}{\log(c+\log t+\log N)}\right)$ bandwidth overhead, where N is the number of data elements, c the correctness parameter, and t is the upper bound on the number of accesses.*

Proof. The key idea of why the proposed scheme is oblivious stems from the basic observation that every column intersects with every row. Intuitively, this means that if we obliviously write an element into some uniformly random position in a row, then, from an adversarial point of view, every column is equally likely a potential candidate for reading that element in a future access. In our scheme, whenever we read an element through a column access, we move it to a new uniformly random bucket and, in particular, a new uniformly random column, through a row access. Importantly, the movement of each element is completely independent of the access history and the other elements residing in the matrix. From these observations it is straightforward to see that the proposed scheme is oblivious.

What remains to show is the relation between the bucket size and the correctness parameter, i.e. we want to pick our buckets sufficiently large such that a bucket overflows with negligible (in the correctness parameter) probability. Towards this goal, we make an observation that simplifies our analysis. Let $\mathsf{Exp}^{N,t}_{\mathsf{move}}$ be the experiment of first throwing N balls into N buckets once and then picking up a random ball from a random bucket and moving it to a new random bucket t times. This experiment expresses the actual movement of data during t many accesses in our oblivious ram construction. Let $\mathsf{Exp}^{N}_{\mathsf{throw}}$ be the experiment of throwing N balls into N initially empty buckets of capacity w. Let $\mathsf{Load}^{>w}_i$ denote the event that bucket with index i at some point in time has more than w many elements in it and $\mathsf{Load}^{>w}$ the event that this happens to any of the buckets. We will analyze the probability of the event of one bucket overflowing in $\mathsf{Exp}^{N}_{\mathsf{throw}}$ and use the following lemma to put $\mathsf{Exp}^{N}_{\mathsf{throw}}$ and $\mathsf{Exp}^{N,t}_{\mathsf{move}}$ into relation.

Lemma 1. *Let $t > 0$, then*

$$\Pr[\mathsf{Load}^{>w}_i | \mathsf{Exp}^{N,t}_{\mathsf{move}}] \leq t \cdot \Pr[\mathsf{Load}^{>w}_i | \mathsf{Exp}^{N}_{\mathsf{throw}}]$$

Proof. Given N balls and N bins, there are N^N different possibilities to distribute the balls among the bins. Let us call each way to distribute the balls a constellation. Let X be one arbitrary but fixed constellation among them and, since all of them are equally probable, we have $\Pr[X|\mathsf{Exp}^{N}_{\mathsf{throw}}] = \frac{1}{N^N}$. Let us now consider constellations, which are one ball move away from X. There are exactly $N^2 - N$ such constellations, because we can select any of the N balls, and pick any of $N-1$ buckets distinct from the current bucket of the selected ball. Selecting a random ball uniformly and moving it to a random bucket yields the original constellation with probability $\frac{1}{N^2}$, and each of the neighbouring constellations with probability $\frac{1}{N^2}$. As all ball moves are reversible, each constellation can be obtained from N^2-N other constellations. The probability of obtainining a constellation after a uniform selection of constellation and a single random ball move is therefore equal to $\frac{N^2-N}{N^2}\frac{1}{N^N} + \frac{1}{N}\frac{1}{N^N} = \frac{1}{N^N}$. The lemma follows by induction over t and then applying the union bound.

Using this lemma it is sufficient to upper bound the probability of a bucket overflowing in the experiment $\mathsf{Exp}^{N}_{\mathsf{throw}}$ and then apply the union bound over all buckets. Let us first look at the probability of some single bucket i overflowing by one element after $\mathsf{Exp}^{N}_{\mathsf{throw}}$, i.e. the probability of a bucket containing (exactly) $z = w + 1$ balls after throwing N balls into N buckets at random once. In the analysis we assume N to be sufficiently large, i.e. N should be large enough for our bucket size w to be at least 8, so that our inequalities work out. In the following calculation we will use two inequalities. First, $\forall x \geq 0$ it holds that $\left(1 - \frac{1}{x}\right)^x \leq e^{-1}$ and, secondly, $\forall 0 \leq k \leq n$ it holds that $\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{en}{k}\right)^k$.

$$
\begin{aligned}
\Pr[\mathsf{Load}_i^z|\mathsf{Exp}_{\mathsf{throw}}^N] &= \binom{N}{z}\left(\frac{1}{N}\right)^z\left(1-\frac{1}{N}\right)^{N-z} \\
&\leq \left(\frac{eN}{z}\right)^z N^{-z}\left(\left(1-\frac{1}{N}\right)^N\right)^{1-\frac{z}{N}} \\
&\leq \left(\frac{eN}{z}\right)^z N^{-z} e^{\frac{z}{N}-1} \\
&= e^z z^{-z} e^{\frac{z}{N}} e^{-1} \\
&= 2^{z(\log e - \log z)+\log e\left(\frac{z}{N}-1\right)} \\
&\leq 2^{z(\log e-\log z)} \\
&\leq 2^{-\frac{1}{2}z\log z}
\end{aligned}
$$

We can provide an upper bound on the event of a single bucket having more than w balls after throwing N balls into N buckets using geometric series as follows

$$
\begin{aligned}
\Pr[\mathsf{Load}_i^{>w}|\mathsf{Exp}_{\mathsf{throw}}^N] &\leq \sum_{z=w+1}^{N} 2^{-\frac{1}{2}z\log z} \\
&= \sum_{z=1}^{N-w} 2^{-\frac{1}{2}(w+z)\log(w+z)} \\
&= \sum_{z=1}^{N-w} 2^{-\frac{1}{2}(w\log(w+z)+z\log(w+z))} \\
&\leq \sum_{z=1}^{N-w} 2^{-\frac{1}{2}(w\log w+z\log z)} \\
&= 2^{-\frac{1}{2}w\log w}\sum_{z=1}^{N-w} 2^{-\frac{1}{2}z\log z} \\
&\leq 2^{-\frac{1}{2}w\log w+1}
\end{aligned}
$$

Applying the union bound over all buckets and using Lemma 1 we obtain

$$
\Pr[\mathsf{Load}^{>w}|\mathsf{Exp}_{\mathsf{move}}^{N,t}] \leq 2^{-\frac{1}{2}w\log w+1}tN
$$

We want to bound this probability of a bad event happening by some correctness parameter c, i.e. we want this probability to be smaller than 2^{-c}.

$$
\begin{aligned}
& 2^{-\frac{1}{2}w\log w+1}tN \leq 2^{-c} \\
\Leftrightarrow\ & -\frac{1}{2}w\log w + 1 + \log t + \log N \leq -c \\
\Leftrightarrow\ & w\log w \geq 2(c+\log t+\log N+1)
\end{aligned}
$$

Hence, the bucket size $w \in \mathcal{O}\left(\frac{c+\log t+\log N}{\log(c+\log t+\log N)}\right)$ and therefore the total bandwidth cost in our construction is $\mathcal{O}\left(\sqrt{N}\frac{c+\log t+\log N}{\log(c+\log t+\log N)}\right)$.

4 Main Construction

In this section we are going to present our main Lookahead ORAM construction. The first difference between our Matrix Bucket ORAM and Lookahead ORAM is that we replace all buckets by cells that can only hold single elements. As a first try to construct a more efficient ORAM we could do the following: Initially, randomly shuffle the data, distribute the data elements among matrix cells, and encrypt each cell separately. To access an element, we retrieve the column corresponding to that element and a row corresponding to a uniformly random cell. After accessing the desired element, we swap the accessed element with the uniformly random cell, re-encrypt both row and column, and write them back into the matrix.

On an intuitive level, one could hope for this to be a secure ORAM construction, since every element will be swapped into a new uniformly random column at every access. Unfortunately this is not the case. The problem is that the distribution of columns into which elements are swapped is not uniformly random when conditioned on the observed row accesses. In particular, the difference with the simple matrix construction is that, here, the accessed element will change the position of another element, i.e the swap partner. It turns out that the server can infer information about the positions of accessed elements whenever we access the same row twice.

Figure 3 illustrates why the straightforward approach of directly swapping the accessed element with an element from a uniformly random cell fails. In the figure, the root node depicts a 2×2 matrix holding four encrypted entries. Initially the server has no knowledge about the arrangement of data elements in the matrix. Let us assume we access two different data elements. With probability non-negligible in the security parameter the following events will occur. On the first access the server observes the second column and second row being accessed. Edges from the root to the first layer show the possible swaps that could have happened, given the observed row and column accesses. $a \leftrightarrow b$ means element a was accessed and swapped with element b. On the second access the server observes the first column and, again, the second row being accessed. The leaf nodes of this tree represent all possible arrangements of elements in the matrix, given the observed access pattern. Dashed boxes indicate the case, where the first accessed element has changed its column. Solid boxes indicate that the first accessed element is in its original column. Counting the leaf nodes it can be seen that the first accessed element will more likely than not have switched columns after the two accesses. Hence, from the server's point of view, the elements are not distributed uniformly at random among the columns and the approach does not provide the desired obliviousness guarantees.

4.1 Intuition for Lookahead ORAM

The main issue with this first approach is that the row accesses reveal too much information. Ideally, we would like to have a swap procedure that allows us to directly access the desired element instead of the whole column and then swap that accessed cell with a new cell without revealing the column or row of that new cell. Observe, that to perform a swap, we have to perform two tasks. We have to remove the accessed element from its cell and put it into the cell of its swap partner. Symmetrically, we have to remove the swap partner from its cell and put it into the cell of the accessed element.

To realize such a swap procedure, we introduce two auxiliary stashes stash_{acc} and stash_{swap}, where stash_{acc} will be storing accessed elements and stash stash_{swap} will store pre-selected swap partners. From a high-level perspective, these stashes will help us to pretend that we immediately swap accessed elements to an unknown new location in the matrix. From the server's point of view, the client will always read both full stashes, and a uniformly random cell in the matrix, since the client behaves as if accessed elements are immediately swapped to their new locations in the matrix. In reality, accessed elements will go to stash_{acc} from where they will be eventually evicted into the cell of their respective swap partner obliviously. Swap partners will be readily waiting in stash_{swap} and upon accessing some element in the matrix, the swap partner will be swapped from stash_{swap} into the accessed cell directly. As an invariant we have that each element is either at its expected location in the matrix or in one of the stashes.

Two issues that need to be addressed are, how do we get swap partners into stash_{swap} before they are used and how to get accessed elements from stash_{acc} into their new cells in the matrix. To solve both these issues, we introduce a (stateful) round-robin column access that will iterate through the columns. Using the

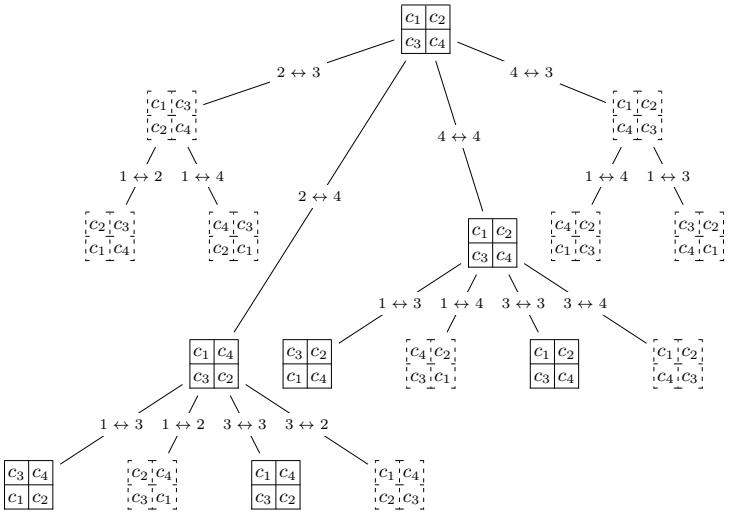

Fig. 3. Illustration of why the naive approach of swapping two random cells via a column and a row access fails. The root node depicts a encrypted 2×2 matrix. The leaf nodes depict all possible arrangements of data elements that are possible after the observed access pattern. Dashed boxes indicate the case, where the first accessed element has changed its column. Solid boxes indicate that the first accessed element is in its original column.

round-robin column access, we perform two "maintenance" tasks. To empty stash_{acc}, we evict all elements from it, whose destination is somewhere in the column of the current round-robin column access. Note that for a matrix of size $\sqrt{N} \times \sqrt{N}$, the round-robin access will have accessed every cell of the matrix in $\sqrt{N}$ steps. This means that no element in stash_{acc} will wait for more than $\sqrt{N}$ steps to be evicted. Since, we add at most one element to the stash per access, this means that stash_{acc} will never contain more than $\sqrt{N}$ elements.

The second task is to ensure that, whenever we use a swap partner from stash_{swap}, the content of the swap partner's cell must already be available in that stash. Observe, that the swap partner's cell is a uniformly random cell in the matrix, which does not depend on the access pattern and can be selected upfront. Assume stash_{swap} contains $\sqrt{N}$ many preselected swap partners in a queue. These swap partners are sufficient for the next $\sqrt{N}$ accesses. Now upon performing an access, one swap partner from the stash will be used, and we pre-select a uniformly random cell that will be the swap partner once all other swap partners from stash_{swap} are used. At this point in time, the content of the pre-selected swap partner is (likely) not in the stash. However, since we have $\sqrt{N} - 1$ many more accesses before it will be used, we can be certain that the round-robin column access will fetch it in time before it will be used. Since our stashes accommodate accessed elements waiting to be evicted and swap partners waiting to be used, the total stash size is $2\sqrt{N}$.

One detail that we have swept under the rug so far, is the case, where the element we want to access is not in the matrix, but somewhere in the stashes. To get an intuitive feeling for the handling of these cases, it is helpful to keep in mind that at every access, we are basically virtually swapping two cells in the matrix. The stashes are just auxiliary data structures that make this process happen. Let (i, j) be the cell in the matrix that is expected to contain the element we are accessing and let the next swap partner from stash_{swap} be some data element v originating from some cell (a, b). If the desired data element is not at position (i, j) in the matrix, but rather in stash_{acc} (waiting to be evicted to some cell (a', b')), then take the next swap partner from stash_{swap} and place its value v into the matrix at position (i, j). From now on the element in stash_{acc} that was expected to be at (i, j) will be waiting to be evicted to (a, b). If the accessed element, expected at location (i, j), is in stash_{swap}, this means that (i, j) is pre-selected as a swap partner for some future access. Therefore the contents of (i, j) are not supposed to be in the matrix, but rather in

Globals

```
C, σ
stash = (stash_swap, stash_acc)
ind_col
```

Init(A)

```
C ← initEmptyMatrix(√N × √N)
(A', σ') ← shuffle(A)
σ ← fillMatrix(A', σ', C)
(stash_swap, stash_acc) ← initStash()
for i = 1 ... √N do
    (i, j) ← pickRandCell()
    enqueue(((i, j), ⊥), stash_swap)
ind_col ← 0
for i = 1 ... √N do
    Background();
```

Access(ℓ, x)

```
v.pos ← σ(ℓ)
(v.val, where) ← ReadVirtual(v.pos)
s ← dequeue(stash_swap)
if x ≠ ⊥ then
    v.val = x
SwapVirtual(v, s, where)
swapInPosMap((i, j), s.pos, σ)
(i', j') ← pickRandCell()
enqueue(((i', j'), ⊥), stash_swap)
Background()
return v
```

SwapVirtual(v, s, where)

```
switch where do
    case matrix
        C[v.pos] = s.val
    case readstash
        remove(v.pos, v.val, stash_acc)
        C[v.pos] = s.val
    case swapstash
        t ← find(v.pos, stash_swap)
        replaceVal(t, s.val, stash_swap)
put(s.pos, v.val, stash_acc)
```

Background()

```
c = readColumn(ind_col, C)
for t ∈ stash_acc do
    if t.pos.j = ind_col then
        c[t.pos.i] == t.value
        remove((i, j), stash_acc)
for t ∈ stash_swap do
    if t.pos.j == ind_col then
        if c[t.pos.i] ≠ ⊥ then
            t.value = c[t.pos.i]
            c[t.pos.i] = ⊥
writeColumn(j, c, C)
ind_col ← ind_col + 1 mod √N
```

ReadVirtual((i, j))

```
if C[i, j] ≠ ⊥ then
    return (C[i, j], matrix)
v = lookup((i, j), stash_acc)
if v ≠ ⊥ then
    return (v, readstash)
v = lookup((i, j), stash_swap)
return (v, swapstash)
```

Fig. 4. Pseudocode of Lookahead ORAM

the stash_{swap}. In this case, we find (i, j) in the stash_{swap}, put the value of (i, j) into stash_{acc} to be evicted into (a, b), and replace the value of (i, j) in stash_{swap} with v.

4.2 Formal Description

Given this intuition about how our construction work, we are now ready to formally present our construction in Figure 4. Let C be the matrix containing the encrypted data entries and σ be the position map that maps array indices to matrix positions in C. We implement stash_{swap} as a queue and stash_{acc} as an map from positions to values. $\mathrm{Init}(A)$ initializes the ORAM data structure, by permuting the elements and storing them in an encrypted matrix C. Initially, both stashes are created empty. stash_{swap}, is filled up with random elements from the encrypted matrix. To write value x or just read at position ℓ in array A inside the ORAM data structure, we use $\mathsf{Access}(\ell, \mathsf{x})$, which makes use of the ReadVirtual, WriteVirtual, and Background subroutines. $\mathsf{ReadVirtual}(\mathsf{i}, \mathsf{j})$ reads the stash and the matrix cell $C[i, j]$ to find the data element that is expected to be at position (i, j) inside the matrix C. $\mathsf{SwapVirtual}(\mathsf{v}, \mathsf{s}, \mathsf{where})$ simulates a swap of accessed value $v.val$ at position $v.pos$ with pre-selected swap partner s with value $s.val$ from position $s.pos$. $\mathsf{Background}()$ implements the round-robin column access, which takes care of flushing elements out of stash_{acc} and fetching elements into stash_{swap}.

4.3 Security

Theorem 2. *Let $\mathcal{E} = (\mathsf{gen}, \mathsf{encrypt}, \mathsf{decrypt})$ be an* IND-CPA *secure encryption scheme. Then the construction in Figure 4 is a secure ORAM scheme with perfect correctness and $\mathcal{O}\left(\sqrt{N}\right)$ bandwidth overhead, where N is the number of data elements.*

Proof. Instead of directly arguing about the security of our proposed construction, we will rather argue about the security of an idealized version, which leaks the same amount of information about the access pattern, but is easier to analyze. As previously explained, from a high-level perspective, our construction directly accesses the desired element in the matrix and then swaps it with a random cell. The swap is immediately applied to the position map and we always directly access the cell in the matrix, which should contain a desired element according to the position map. The stash and the round-robin column accesses are there to enable us to virtually swap the accessed element into a new cell without leaking anything about that new location.

$\mathsf{IdealizedAccess}_1(\sigma, \ell)$

1 $(i, j) \leftarrow \sigma(\ell)$
2 $(i', j') \leftarrow$ `pickRandCell(`C`)`
3 $v \leftarrow$ `readCell(`i, j, C`)`
4 $C \leftarrow$ `readMatrix(`C`)`
5 `swapInPosMap(`$(i, j), (i', j'), \sigma$`)`
6 `swapInPosMap(`$(i, j), (i', j'), C$`)`
7 **return** v, σ

$\mathsf{IdealizedAccess}_2(\sigma, \ell)$

1 $(i, j) \leftarrow \sigma(\ell)$
2 $(i', j') \leftarrow$ `pickRandCell(`C`)`
3 $v \leftarrow$ `readCell(`i, j, C`)`
4 $C \leftarrow$ `readMatrix(`C`)`
5 `swapInPosMap(`$(i, j), (i', j'), \sigma$`)`
6 $(C, \sigma) \leftarrow$ `fullReshuffle(`C, σ`)`; **return** v, σ

Fig. 5. Idealized access procedures

Since both the stash and the the round-robin column access are always executed independently of the access pattern, they leak no information. Hence, instead of analyzing our construction directly, we can analyze a construction with the idealized access procedure $\mathsf{IdealizedAccess}_1$ depicted in Figure 5 on the left. The initialization procedure corresponding to $\mathsf{IdealizedAccess}_1$ is a straightforward adaption of our main construction and is not stated explicitly. In $\mathsf{IdealizedAccess}_1$, we directly access the cell that contains our data

element and, next, we retrieve the full matrix to perform the swap operation locally. From an efficiency point of view this is clearly a useless construction, but with respect to obliviousness both our main construction and this idealized version thereof leak the same amount of information about the access pattern. More formally, the success probability of any distinguisher D, distinguishing two data access sequences, is the same in our main construction and in the construction with $\mathsf{IdealizedAccess}_1$.

Lemma 2. *Let* oram *be the main construction from Figure 4 and* oram_1^* *the construction using the access procedure* $\mathsf{IdealizedAccess}_1$*. Then, for any distinguisher* D*, for any two data request sequences* $\overrightarrow{y}$ *and* $\overrightarrow{z}$ *we have*

$$\begin{aligned} &|\Pr[\mathsf{D}(\mathsf{oram}(\overrightarrow{y})] - \Pr[\mathsf{D}(\mathsf{oram}(\overrightarrow{z}))]| \\ &= |\Pr[\mathsf{D}(\mathsf{oram}_1^*(\overrightarrow{y}))] - \Pr[\mathsf{D}(\mathsf{oram}_1^*(\overrightarrow{z}))]| \end{aligned}$$

There are two components that are observable by the server in both our real construction and the idealized access that can leak information about the access pattern. The first component is the swap logic that moves accessed elements to a new position. The second component is the direct accesses to the desired elements, i.e. we need to show that conditioned on previously observed accesses, each new access will fetch a uniformly random cell in the matrix.

Towards showing the first part, let v_k for $1 \leq k \leq N$ be some arbitrary data elements. Let $\mathsf{Exp}_{\mathsf{swap}}^{N,t}$ be the experiment of, initially, distributing the N data elements v_k in a matrix C with N cells uniformly at random and then for t steps repeatedly swapping the contents of two uniformly random cells.

Lemma 3. *Let* C *be a matrix of size* $\sqrt{N} \times \sqrt{N}$ *and let* $v_k \in \mathcal{V}$ *for* $1 \leq k \leq N$ *be arbitrary values from some value space* $\mathcal{V}$*. Then* C*, after running experiment* $\mathsf{Exp}_{\mathsf{swap}}^{N,t}$*, is a uniformly random permutation of the data elements* v_k*.*

Proof. Initially distributing the data elements v_k uniformly at random in the matrix C corresponds to a uniformly random permutation. For $\mathsf{Exp}_{\mathsf{swap}}^{N,1}$, i.e. distributing the elements and then swapping two uniformly random cells once, the statement holds, since we apply a random permutation of two elements to a uniformly random permutation. The statement for $t > 1$ follows by induction over t.

To conclude the security proof it remains to show that even conditioned on the previously observed accesses the distribution of data elements in C is uniformly random. Assume the server observes an access to position (i, j) in the matrix to fetch some data element v_k. The accessed element is going to be swapped into every position in the matrix with equal probability. Since each element is equally likely, with a probability of $\frac{1}{N}$, to be selected as a swap partner, every element is equally likely to end up in (i, j). From the accessed cells, no other information about the access pattern is leaked. Hence, from the server's perspective all distributions of access patterns are equally likely, no matter what the actual data access sequence is.

For the sake of clarity, let us look at the slightly modified access procedure $\mathsf{IdealizedAccess}_2$ depicted in Figure 5 on the right. In $\mathsf{IdealizedAccess}_2$ we do not just swap two cells locally, but we fully reshuffle the whole matrix. Due to the full reshuffle, each access is completely independent of the previously observed access pattern. It is straightforward to see that $\mathsf{IdealizedAccess}_2$ is a secure ORAM construction. Since in both $\mathsf{IdealizedAccess}_1$ and $\mathsf{IdealizedAccess}_2$ the access patterns are distributed uniformly at random, $\mathsf{IdealizedAccess}_2$ leaks as much information as $\mathsf{IdealizedAccess}_1$.

4.4 Online Overhead

For some practical applications it may be of interest to split the total bandwidth overhead into an online and an offline overhead. The point of this is to minimize the online bandwidth, which represents the amount of data that needs to be transmitted, when a client requests an element, and then do some background work, the offline overhead, to ensure the security of the ORAM, when no data requests are actively pending. This way we can minimize the practical latency of user requests despite the inherent lower bound on the overhead shown in [Gol87, GO96].

Looking at our main construction, it is straightforward to split the total bandwidth overhead into online and offline overhead. In the online phase, assuming we download the stash once, we can directly access the desired elements in the matrix on the server without any overhead. In the offline phase we need to do the remaining work, i.e. perform the round-robin column access, fill up the stash of pre-selected swap partners and flush out elements that need to go back into the matrix.

While storing the stash locally can theoretically compromise data integrity in case of a client device failure, the only part of the stash that cannot be randomly reinitialized is the cache of recently accessed elements. If the server is is significantly more reliable than the client device, recently accessed elements can be written to a separate server-side buffer of size $\sqrt{N}$ in a round-robin manner. Each element will be stored in this buffer for $\sqrt{N}$ operations which is enough for the background operations to write it to its long-term server-side storage position.

It is also possible to allow multiple online accesses in a single burst. The simplest way to do it increases local storage requirements by twice the size of the maximum allowed burst length. Note that our implementation faithfully simulates the oram_1^* construction as long as the background work is performed at least $\sqrt{N}$ times between committing to a swap partner and its use, and background work is also performed at least $\sqrt{N}$ times between accessing the element and evicting it from the recently accessed element stash. If there is additional stash space of size b for the potential swap partners and the same amount of space for the recently accessed elements, we only need to have at least $\sqrt{N}$ background work operations performed during every interval when $b+\sqrt{N}$ access operations are performed. Note that it is possible to perform some part of the background work, handle an additional burst and then continue the background work as long as enough background work is done to prevent exhaustion of the swap partners or overflow of the recently accessed elements stash.

The same analysis shows that there is a trade off between stash size and bandwidth overhead. For example, if we have a stash twice as large as needed, we can afford doing only half the background processing step after each access.

4.5 Trading Off Bandwidth and Storage Overhead

For our main construction, the server's storage overhead comes from the two auxiliary stashes that it needs to store in addition to the encrypted data elements. For a matrix C, where the number of columns equals the number of rows, this results in an additive storage overhead of $2\sqrt{N}$. More generally, by considering an arbitrary rectangle C with H rows and W columns, we can trade off the concrete storage and bandwidth overhead costs of our construction. The number of columns W affects the time it takes the round-robin column access to iterate over the whole rectangle C and thus it also affects the stashes which have to be of size W each. The number of rows H, affects the size of the column that we need to download at each access.

For example, by setting $H = 2\sqrt{N}$ and $W = \frac{1}{2}\sqrt{N}$, we can, in comparison to a matrix shaped C, directly reduce the additive storage overhead to $\sqrt{N}$ and maintain a bandwidth overhead of $3\sqrt{N}+1$. By setting $H = \sqrt{2N}$ and $W = \sqrt{\frac{N}{2}}$, we get a bandwidth overhead of $2\sqrt{2}\sqrt{N}+1$ and a storage overhead of $\sqrt{2N}$.

5 Recursive Position Map

So far we have assumed that the full position map is stored explicitly on the client side. Following the approach of [SCSL11], we show that our construction can be modified to only require $\mathcal{O}(1)$ client-side storage at the cost of $\mathcal{O}(\log N)$ rounds of interaction per access between the client and the server. To store the position map on the server side, we will create a sequence of ORAM data structures that increase in size, which represent the position map. This means that now the server stores a sequence of position map ORAMs and an ORAM data structure that contains the actual data. To access an element at a certain index, the client will use the position map ORAMs to determine, which index it should query in the ORAM data structure that stores the actual data.

Theorem 3. *Consider an arbitrary ORAM construction with a client-side position map. Assume there exist such constants K and C that for N entries with a block size of $D \geqslant 4 \log N$, the ORAM construction has a multiplicative bandwidth overhead of $C\sqrt{N}$ and total storage of $(N + K\sqrt{N})D$ bits.*

Such an ORAM can be converted into an ORAM with a server-side position map with a multiplicative bandwidth overhead of $2C\sqrt{N} + 4$ and total storage $ND + 2KD\sqrt{N} + 2N \log N$ bits.

Proof. The proof goes by induction over N. In the base case, for size $N \leqslant 4$, we use a linear ORAM, which simply reads and writes all the blocks for each access. This ORAM has a bandwidth overhead of 4 and no storage overhead. For an arbitrary N, we use the induction hypothesis to replace an ORAM with client-side position map with an ORAM that stores the position map on the server-side. We encode 4 blocks addresses in the position map into one storage block inside the ORAM data structure. Using a block size of $4 \log N$ results in an ORAM data structure that stores $\frac{N}{4}$ blocks, storing 4 addresses each.

Now we apply the induction hypothesis. Since we want to store $\frac{N}{4}$ blocks with $4 \log N$ bits per block, by induction hypothesis our position map ORAM has a total storage cost of $N \log N + 2K(4 \log N)\sqrt{\frac{N}{4}} + 2\frac{N}{4} \log N$ bits. Adding the storage costs of our main ORAM, the total storage cost in bits will be

$$
\begin{aligned}
&N \log N + 2K(4 \log N)\sqrt{\frac{N}{4}} + 2\frac{N}{4} \log N + ND + KD\sqrt{N} \\
&= ND + K(4 \log N)\sqrt{N} + KD\sqrt{N} + \frac{3}{2} N \log N \\
&< ND + 2KD\sqrt{N} + 2N \log N
\end{aligned}
$$

In our construction, every access to the main ORAM requires one access to the position map. Note that we do not need any additional position map accesses for the background work that moves data in our out of the stash. The position map access adds its bandwidth cost of $2C\sqrt{\frac{N}{4}} + 4$ to the main ORAM bandwidth cost of $C\sqrt{N}$. The total overhead is $C\sqrt{N} + 2C\sqrt{\frac{N}{4}} + 4 = 2C\sqrt{N} + 4$.

This completes the proof of the inductive step.

6 Evaluation

To verify that our calculations of the concrete performance costs of Lookahead ORAM are correct and complete, we implemented a prototype thereof in C and measured storage and bandwidth costs for different parameter settings. Our implementation uses libsodium [BLS12] for the underlying encryption. Each block is encrypted using Salsa20 in combination with MACs to provide authenticated encryption. Each encrypted block is 40 bytes larger than the corresponding plaintext data block. Both the stash of accessed elements and the stash of pre-selected swap partners are implemented as arrays. Each entry in the stash has an additional 20-byte header containing the blocks status, its logical index, and its corresponding position on the server.

When encrypting N data blocks of size B each, the server's total storage is $N \times (B+40)$. The corresponding position map is $8N$, and the stash $2\sqrt{N} \times (B+20)$ bytes large. Whenever the client performs an operation on the ORAM data structure, it needs to download $40+(B+40)\times(\sqrt{N}+1)$ and upload $80+(B+40)\times(\sqrt{N}+1)$ bytes. During initialization of our ORAM, we fill the storage with zeros and then fill the stash with pre-selected swap partners. During this initalization, we upload $44 + (B + 40) \times N$ bytes to the server. All of the above formulas have been calculated based on our theoretical construction and validated empirically based on our concrete implementation. In the following, the data block size is fixed to 1024 bytes.

Storage overhead. We compare our storage overhead on the server-side to the storage overheads of the most relevant related work. For our comparison, measure the total storage requirements on the server-side

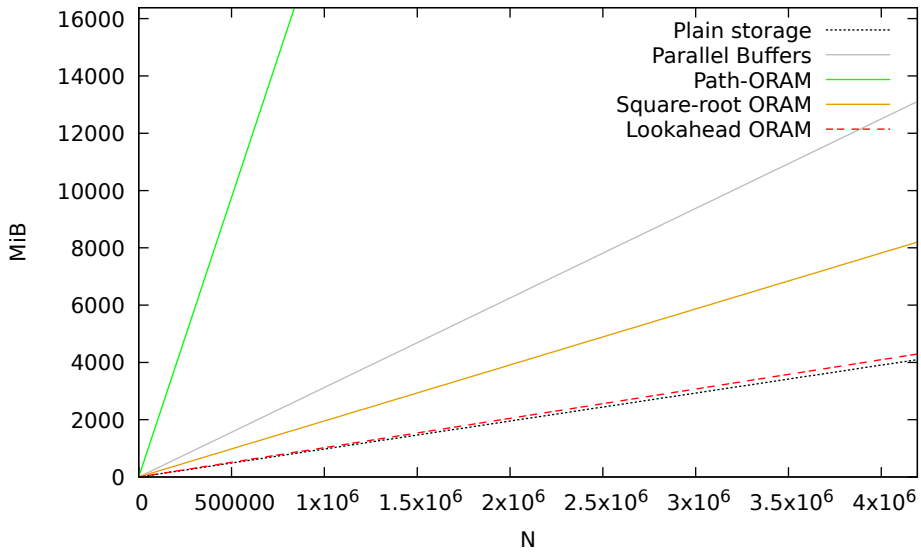

Fig. 6. Comparison of the storage overheads of different ORAM constructions. The x-axis shows different amounts of data blocks. The data block size is fixed to 1024 bytes. The y-axis plots the total required storage on the server-side in MiB. For Lookahead ORAM, the values are taken from empirical measurements of our implementation. For the other schemes, the values are computed based on the concrete formulas and constants that are reported in the respective papers.

for varying N. The data points for Lookahead ORAM were obtained through experiments. For the schemes we compare ourselves to, we computed the data points based on the formulas (including constants) given in the respective papers. It should be noted that in [SvDS+13] the authors obtain provable security for Path-ORAM with a storage overhead of $20N$, but evaluate their scheme on smaller parameter settings. Since we are interested in provable security and correctness guarantees, we compare our scheme to theirs with a storage overhead of $20N$. The results are depicted in Figure 6.

Bandwidth overhead. We compare ourselves to Path-ORAM, which is known to be the most efficient construction in terms of asymptotic and practical bandwidth overhead. For our comparison we use their self-reported bandwidth overhead of $10 \log N$. The results of our comparison are depicted in Figure 7. As expected from the asymptotic behaviour of Path-ORAM and Lookahead ORAM, we can see that Path-ORAM becomes more efficient for large values of N. However, for values of $N < 3000$, Lookahead ORAM is more efficient in terms of concrete bandwidth overhead.

Acknowledgements We thank the anonymous reviewers for their useful feedback. We thank Jesper Buus Nielsen and Ivan Damgård for initial discussions that led us to this work. This project received funding from the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme und grant agreement No 669255 (MPCPRO), the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583 (SODA), and the French National Research Agency (ANR project GraphEn / ANR-15-CE40-0009)

References

AKST14. Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. pages 131–148, 2014.

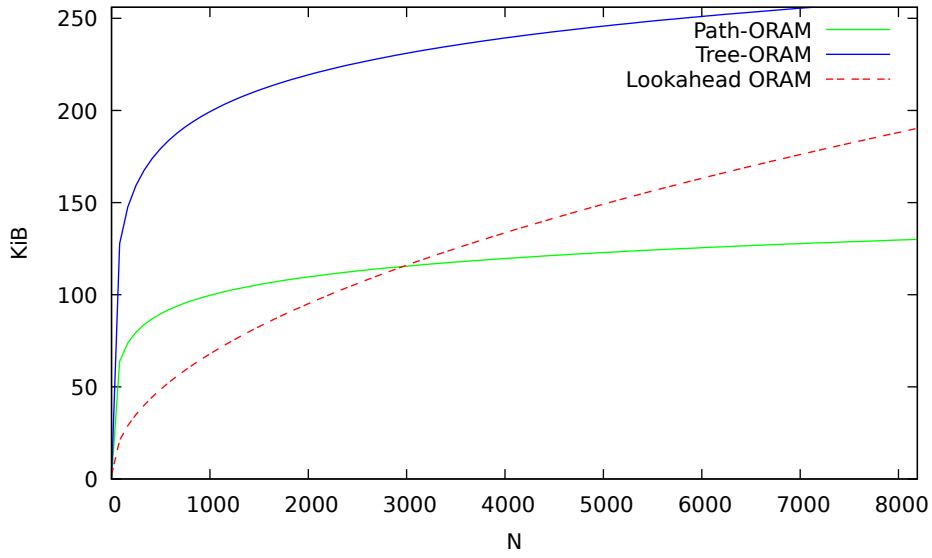

Fig. 7. Comparison of the bandwidth overhead of Lookahead ORAM and Path-ORAM. The x-axis shows different amounts of data blocks. The data block size is fixed to 1024 bytes. The y-axis shows the total amount of transmitted data per access in MiB.

BCP16. Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. pages 175–204, 2016.

BLS12. Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. pages 159–176, 2012.

BMP11. Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious ram practical. 2011.

BN16. Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? pages 357–368, 2016.

CLT16. Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: Improved efficiency and generic constructions. pages 205–234, 2016.

CP13. Kai-Min Chung and Rafael Pass. A simple ORAM. Cryptology ePrint Archive, Report 2013/243, 2013. `http://eprint.iacr.org/2013/243`.

DDF+16. Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. pages 145–174, 2016.

DSS14. Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst oram: Minimizing oram response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, 2014.

GHL+14. Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. pages 405–422, 2014.

GMOT11. Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100. ACM, 2011.

GMP16. Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. pages 563–592, 2016.

GMSS16. Steven Gordon, Atsuko Miyaji, Chunhua Su, and Karin Sumongkayothin. M-ORAM: A matrix ORAM with log N bandwidth cost. pages 3–15, 2016.

GO96. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

Gol87. Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. pages 182–194, 1987.

IKK12. Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. 2012.

LO13. Steve Lu and Rafail Ostrovsky. How to garble RAM programs. pages 719–734, 2013.
MBC14. Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining ORAM and PIR. 2014.
MMB15. Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. Constant communication ORAM with small blocksize. pages 862–873, 2015.
OS97. Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). pages 294–303, 1997.
RFK+15. Ling Ren, Christopher 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)*, pages 415–430, Washington, D.C., 2015. USENIX Association.
SCSL11. Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. pages 197–214, 2011.
SSS12. Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. 2012.
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. pages 299–310, 2013.
SZEA+16. Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 198–217. IEEE, 2016.
WCS15. Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. pages 850–861, 2015.

A Attack on [GMSS16]

In a recent work by Gordon et al. [GMSS16] the authors present an ORAM, called M-ORAM. Their construction has a flaw and does not provide obliviousness. In the following, we give a high-level overview of their scheme and sketch our attack that breaks their obliviousness claims.

The construction partitions the server-side storage into a fixed number of rows and a number of columns that depends on the dataset's size. Every cell in their rectangular storage layout holds one data element. Additionally, every row has its own separate constant-sized stash.

Initially, all data elements are present in the storage rectangle in a randomly permuted order and the stashes are empty. Simply speaking an access is performed by accessing one element in each row of their data structure. In one of the rows the desired element is accessed and in all other rows a uniformly random cell is selected. More precisely, the authors claim that to achieve obliviousness not all "dummy" cells are selected uniformly at random, instead some of them are random cells from the previous access. After retrieving one cell from each row, the client shuffles the cells and puts one cell into each stash. The client picks one random block from each stash and sends it back to the server as the new content of the retrieved cells.

Let $x_1, \ldots, x_N$ be some data elements stored in the ORAM data structure, then the access sequences $(\mathsf{read}(x_1), \mathsf{read}(x_2), \mathsf{read}(x_1))$ and $(\mathsf{read}(x_1), \mathsf{read}(x_2), \mathsf{read}(x_3))$ can be distinguished with a success probability that is non-negligible in the security parameter. From a high-level perspective, every access selects a subset of cells from the data structure and every two subsets corresponding to two consecutive accesses intersect at some random cells. For three accesses the proposed approach breaks down. Looking at our first access sequence, the proposed construction has a slightly higher bias of the first and third access subset intersecting, since we are accessing the same element.

We have contacted the authors and they have acknowledge our attack.