A preliminary version of this paper appears in the proceedings of the *22nd International Conference on Information and Communications Security (ICICS 2020)*, DOI: 10.1007/978-3-030-61078-4_18. This is the full version.

Client-oblivious OPRAM

Gareth T. Davies[1] Christian Janson[2] Daniel P. Martin[3]

[1] Bergische Universität Wuppertal, Wuppertal, Germany
`davies@uni-wuppertal.de`
[2] Cryptoplexity, Technische Universität Darmstadt, Darmstadt, Germany
`christian.janson@cryptoplexity.de`
[3] The Alan Turing Institute, London, UK
`dmartin@turing.ac.uk`

Abstract. Oblivious Parallel RAM (OPRAM) enables multiple clients to synchronously make read and write accesses to shared memory (more generally, any data-store) whilst hiding the access patterns from the owner/provider of that shared memory. Prior work is best suited to the setting of multiple processors (or cores) within a single client device, and consequently there are shortcomings when applying that work to the *multi-client* setting where distinct client devices may not trust each other, or may simply wish to minimise – for legal reasons or otherwise – the volume of data that is leaked to other client devices. In prior constructions, obliviousness from the storage provider is achieved by passing accesses between the clients in one or more sorting networks, both before and after the logical access is made to the shared memory: this process inherently leaks the contents of the accesses to those other clients.

In this paper we address this issue by introducing the notion of *client obliviousness* for OPRAM, which asks that clients should only learn as much as is necessary for the scheme to function correctly. We provide an instantiation using established tools, with careful analysis to show that our new notion and regular OPRAM security are met. This introduces several subtleties which were not previously apparent, and we further discuss the implications of using the OPRAM model in the context of outsourced storage.

Keywords. Oblivious parallel RAM · Client-obliviousness · Outsourced storage security

Contents

1 Introduction

Oblivious RAM is a cryptographic primitive that enables a client to store and retrieve blocks of data on an untrusted storage medium. The beauty of this primitive is that a client can do this in such a way that *no information* about their *access pattern* is revealed to the storage server beyond the total number of accesses. This primitive dates back to the seminal work by Goldreich and Ostrovsky [GO96, Gol87]. ORAM has been extensively studied, both in terms of advanced capabilities and stronger security models [PR10, SCSL11, CLP14, GGH$^+$13].

In this paper we consider the problem of hiding the access pattern when *multiple clients* concurrently read from and write to an untrusted storage server. This is a fundamental problem in the realm of protecting outsourced storage and verification of outsourced computations – Boyle *et al.* [BCP16] defined and constructed *Oblivious Parallel RAM* (OPRAM) and subsequent works have mainly focused on improving the efficiency of realized schemes [NWI$^+$15, CLT16, CCC$^+$16, NK16, SZA$^+$16, CS17, CCS17, CGLS17, CNS18].

The OPRAM literature to date is most suited for the situation where the clients are co-located, such as when each client represents (possibly a core of) a processor in the same computer. Trust between the clients is required because the clients pass their accesses to each other and sort based on the access locations (in order to deal with access conflicts): this process stops the server learning from which client a given access originated. In some situations the clients may be restricted by legal systems or organisational policy and thus want their memory accesses to remain as private as possible from the server (as provided by OPRAM) but also if possible from the other clients. If the clients are processors – or more generically, devices – that are based in disparate geographic locations and are accessing some central (storage) service, then not only is inter-client communication an issue of cost, but also a concern regarding both the privacy and legal implications of multi-jurisdiction data sharing. In short, the low-latency and pairwise-secure channels assumed by previous descriptions of OPRAM may not be realistic in practice.

Motivating Scenario

Consider an organisation with operating facilities in several distinct locations, with numerous legal requirements for each jurisdiction meaning that a strict access control regime and audit trail is required for the data that flows between the clients, and data that is stored on a central storage server. This organisation wishes to store data in such a way that all facilities can append their latest reports at regular intervals, but access the other facilities' data only when necessary (and perhaps only when approved following legal procedure). To do this, a storage provider is tasked with holding the database, but an oblivious RAM protocol is used to hide access patterns, and since the regular update procedure is at a predictable time, oblivious parallel RAM is used to ensure that the identity of the facility updating an entry is hidden from the storage server.

How does this scenario fit with the security model for OPRAM in the literature? Do existing constructions facilitate mechanisms for reducing the volume of data that is leaked to each client as part of the protocol? In this context, there may exist other central entities that are used by the organisation to assist with enhancing privacy for the clients as part of the protocol – if this is the case, then what are the trade-offs regarding efficiency and trust by using such entities? It is these questions that we approach in this work.

Contributions

In this paper we introduce an additional security property for OPRAM schemes, which we call *client obliviousness* (CO), which informally states that the clients should learn as little as possible about the *other* clients' accesses. Numerous subtleties consequently arise, and we address the minimal leakage in (a large class of) OPRAM schemes and the effects of techniques such as client anonymisation and storage-space partitioning. We provide an instantiation that is functionally equivalent to the subtree-OPRAM scheme of Chen *et al.* [CLT16], yet to obtain security in the strong CO sense, the constituent parts are almost all replaced with other primitives (from the cryptographic literature). We conclude by providing a thorough discussion of the implications of our approach, and potential extensions and modifications.

Threat Model

The system consists of a set of *users*, a storage database $\mathcal{S}$, and a routing entity R. The users encrypt data using symmetric encryption and store it with $\mathcal{S}$, and $\mathcal{S}$ is assumed to try to learn as much as it can from correct execution of the protocol, i.e. is honest-but-curious (HbC). Any collusion between one user and $\mathcal{S}$ leads to total loss of any security – this is inherent in ORAM and OPRAM schemes in which the client/all clients have shared ownership of the stored data. The router R is to carry some of the management burden, yet it should not learn which users are making which accesses, and is also assumed to be HbC. We use 'semi-trusted' to refer to the combination of user anonymity and HbC that we desire from R.[1]

Related Work

Oblivious RAM is a very active research area and was initially introduced by Goldreich and Ostrovsky [GO96]. In this paper, we consider the parallel version of ORAM first formalised by Boyle, Chung and Pass (BCP) [BCP16] in work that built upon several earlier ideas [GM11, WST12, LPM+13]. Their OPRAM formulation requires considerable inter-client communication in order to synchronise before and after accesses to the data storage occur. In particular, clients coordinate with each other in an oblivious aggregation phase to ensure that no two clients access the same block simultaneously, and if two (or more) wish to write to the same block, some regime defines which client proceeds. Chen, Lin and Tessaro (CLT) [CLT16] provided a more efficient OPRAM construction, named Subtree-OPRAM, based on an extension of the well-known Path-ORAM [SvS+13] protocol: we will build upon this construction later on. CLT also provided a generic construction from ORAM to OPRAM, with slightly worse complexity than Subtree-OPRAM. Other works have subsequently given further optimised OPRAM schemes [CS17, CCS17, CGLS17, NK16, CNS18].

One area in which curious/malicious clients have been considered is the realm of *Multi-client ORAM* (MC-ORAM) [FWC+12], where a number of *distinct* data owners (clients) use some central data store and can delegate read and write access to other users for their files. (Recall that in OPRAM the entire database is necessarily shared between all clients, so there is no concept of file ownership.) Security of access control in the MC-ORAM context has been studied by Maffei *et al.* [MMRS15, MMRS17] (hereafter MMRS) and their aim is to model the capabilities of adversarial clients who wish to learn i) which clients are making read requests and ii) any information about write requests to data that the adversary does not have access to. We investigate a subtly different scenario that is motivated by OPRAM. Consider a database that is collectively owned by a number of clients who share key material, and is partitioned such that all clients can perform accesses on only a subset of the database – an explicit property of many OPRAM schemes in the literature. If client A wishes to read an entry in the database, it will (usually, if the eventual position is not in its partition) be directed to the partition accessor for that data item, client B, who will make the lookup. Client B can see the value being written or read, this is essential to the proper operation of the system, however, they should not learn the identity of client A. So far this is captured by the definitions of MMRS, however, in our system architecture, following OPRAM constructions in the literature, the

[1]This abstraction – using a device that would normally exist in the system topology of our application scenario – takes inspiration from prior work on verifiable computation [AJCC16], where a simple and minimally-trusted entity acts as a key distribution centre for the clients.

new location of the data item after it is read could be in another partition, and in fact it may well be with high probability. As far as MMRS are concerned this means that data ownership is transferred (to randomly chosen other clients), something their model can't support. We reiterate that this per-timestep partitioning is a natural method for achieving OPRAM since it aids both obliviousness and efficiency.

Chen and Popa [CP20] target hiding file metadata in outsourced storage using multi-client ORAM and two servers that use multi-party computation. Their work hides user identities from the servers, which MMRS do not, however, the malicious clients they consider are essentially the same as in MMRS.

TaoStore [SZA+16] is an extension of Path-ORAM to the asynchronous setting, achieved by employing a trusted proxy; their aim is not to provide an OPRAM construction, but rather to deal with continuous and asynchronous requests to the storage server by one or more clients in the presence of an adversary that learns timing information of the accesses. The security model presented by the authors is cast as a game-based notion: we follow this approach, however, this is approximately where the similarities between their problem setting and ours end. Their proxy is considerably more trusted than the router we wish to employ – the paper's focus is to bundle concurrent reads and eviction operations efficiently and not to hide any information from a client obliviousness perspective. We note however, that employing a trusted proxy can give strong guarantees of client obliviousness, and there may exist scenarios slightly outside of our target problem setting for which this – or a combination of this approach and ours – is a more appropriate solution.

Chakraborti and Sion [CS16, CS19] study efficiency in parallel accesses to ORAM architectures. In the process, they consider the information leakage to each client inferred by the global set of accesses, but their threat model is considerably weaker than our notion of client obliviousness: no attempts are made to stop observation of accesses of the other clients. As mentioned before, works regarding multi-client ORAM schemes fall into a similar regime. Recall that in this setting, multiple clients have their *own* data, however, stored in a single ORAM, where each client is free to *share* parts of their data with other clients. Franz *et al.* [FWC+12] initiated the study of multi-client ORAM by introducing the concept of delegated ORAM. Karvelas, Peter and Katzenbeisser [KPK16] introduced Blurry-ORAM, a multi-client extension of Path-ORAM that tried to hide the access patterns *for their own data* from the storage server as well as other clients. Clients owning only some data and sharing with other clients requires sharing and revocation algorithms: these concerns are not relevant to the OPRAM scenario.

Organization of the Paper

In Section 2 we review the necessary background regarding ORAM and its parallel version OPRAM. In Section 3, we turn our attention to defining client-oblivious OPRAM, and in Section 4 we provide a construction with associated security proof that demonstrates how to achieve this notion. We conclude the paper in Section 5. Additional preliminaries and a brief discussion on non-interactivity can be found in the Appendix.

2 Preliminaries

We begin by setting the scene for Oblivious RAM [GO96] and its parallel analogous definition as put forward by BCP [BCP16]. Throughout this section, we follow the notation from both BCP and CLT.

2.1 Notation and Abstraction Level

For vector $\mathbf{x}$, let $\mathbf{x}[i]$ indicate the i-th component of $\mathbf{x}$, and for integer n let $[n]$ be the set $\{1, ..., n\}$. We will at times define a vector as the concatenation of vectors: in this case consider the result as a matrix with a vector in each column. If $\mathbf{L}$ is a matrix, we use $\mathbf{L}_{i,j}$ to specify the entry in the i-th row and j-th column, and we will give context where necessary to identify which component serves which purpose. $\Delta(\cdot,\cdot)$ is the statistical distance between two distributions.

Since our work is mainly applicable to the setting of outsourced storage, we follow CLT's approach and notation for casting O(P)RAM in terms of clients, servers and accesses, rather than as abstract (parallel) RAM program compilers – the formulations are in our setting equivalent. An oblivious RAM compiler essentially turns the *logical* accesses to the storage medium into a sequence of *actual* accesses, in such a way that the logical requests are hidden if the honest-but-curious server only sees the actual sequence of accesses. In the multi-client setting, the adversary sees the transcripts of communication among the clients (in addition to the communication between each client and the server).

Fix N', the number of cells (each of size B') of the (external) database, and m, the number of clients. Interactions between any client and the server's storage (i.e. actual accesses) are of the form $\mathsf{Acc}(op, \mathbf{a}, v)$ where $op \in \{\text{read}, \text{write}\}$, $\mathbf{a} \in [N']$, and v is either in $\{0,1\}^{B'}$ (for writes) or $\perp$ (for reads). An oblivious parallel RAM (scheme/compiler) $\mathcal{O} = \{\mathcal{C}_i\}_{i\in[m]}$ takes as input security parameter λ, storage size parameter N, and block size B, and proceeds in a sequence of T rounds, which represent the synchronous accesses of the m clients. The logical accesses, which can be regarded as 'pre-compiled', are defined as above except for being in the correct spaces: $\mathbf{a} \in [N]$ and $v \in \{0,1\}^B \cup \{\perp\}$. For all $i \in [m]$, denote the logical operations of client $\mathcal{C}_i$ as

$$\mathbf{y}_i = \left(\mathsf{Acc}(op_{i,r}, \mathbf{a}_{i,r}, v_{i,r})\right)_{r\in[T]}.$$

Then, collect these operations using $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_m)$. In the interactive OPRAM protocol that is produced by the compiler from the parameters and these logical accesses, the clients can communicate with each other (direct, point-to-point) and make 'actual' accesses to the server $\mathcal{S}(N', B')$. In our construction later, we will additionally allow clients to further interact with a routing entity R. In each round, each $\mathcal{C}_i$ will output (intuitively: receive) some output $\mathsf{val}_{i,r}$ and update its local state. If two or more parties wish to access the same location in a given round, we term this an *access collision*. Similarly, if two or more parties wish to write to the same location in a given round, we term this a *write collision*.

We follow CLT in writing the server as $\mathcal{S}(N', B')$ where N' is a function of N, and B' is a function of B (and the security parameter) – in all existing schemes the relationship for block size expansion represents encryption: $B' = B + O(\lambda)$.

2.2 Write-Conflict Resolution

BCP and CLT followed the concurrent-read-concurrent-write (CRCW) approach, explicitly insisting that in the event of a write collision, the client with the lowest identifier will be the one that gets to go ahead and write. We do not make such a restriction, and leave the write-conflict *regime* ($\mathtt{Reg}$) as a system parameter. Fich, Ragde and Wigderson [FRW84] detailed a number of possible regimes, including:

- PRIORITY [Gol78] (as used by BCP and CLT): clients have assigned identifiers, and priority is given to e.g. the client with the lowest identifier;

- ARBITRARY [Vis83]: An arbitrary processor is allowed to write;

- COMMON [Kuc82]: Simultaneous writes to a location are allowed as long as the clients are writing the same data;

- COLLISION: No client gets to write, and the special symbol $\perp_c$ is written to the memory location.

Further, we also note that the concurrent-read-exclusive-write model (at most one client is allowed to write to a location in each time step) described by Fich *et al.* and introduced by Fortune and Wyllie [FW78] may also be appropriate for our setting, though this is a simplification that reduces a number of the challenges that we tackle later on. In the scenario that motivates our work, the entity (or group of users) tasked with access control would define which regime is in place for a subset of the rounds, or for the lifetime of the system.

2.3 Oblivious Parallel RAM

For an OPRAM compiler to be meaningful and useful, it must be *correct* and *oblivious*. We again follow CLT in this regard. We need to introduce the write-conflict regime $\texttt{Reg}$ as a parameter of the algorithms used to determine these two properties. We write $\mathcal{O}(\mathbf{y})$ as the executed compilation for logical (sequence of) accesses $\mathbf{y}$. Inspired by CLT, we define

$$\mathsf{ACP}_{\mathcal{O}}(\lambda, N, B, \texttt{Reg}, \mathbf{y}) = (\mathsf{ACP}_1, \dots, \mathsf{ACP}_T)$$

as the collection of communication patterns for each round, representing the transcript of communication between the clients, (between the clients and the third party, if it exists,) and between clients and the server. Intuitively, a scheme provides obliviousness if an adversary given this information cannot infer anything about $\mathbf{y}$ (other than the number of accesses). Similarly, we can define $\mathsf{ACP}_{i,r}$ as the communication pattern for client $\mathcal{C}_i$ in round r. Further, write the outputs for client i as $\mathbf{val}_i = (\mathsf{val}_{i,1}, \dots, \mathsf{val}_{i,T})$ and all outputs as

$$\mathsf{Out}_{\mathcal{O}}(\lambda, N, B, \texttt{Reg}, \mathbf{y}) = (\mathbf{val}_1, \dots, \mathbf{val}_m).$$

For an OPRAM compiler $\mathcal{O}$, outputs $\mathbf{z} = \mathsf{Out}_{\mathcal{O}}(\lambda, N, B, \texttt{Reg}, \mathbf{y})$ are correct with respect to (parallel access) sequence $\mathbf{y}$ if for each command $\mathsf{Acc}(op_{i,r}, \mathbf{a}_{i,r}, v_{i,r})$ of $\mathbf{y}$, the output $\mathbf{val}_{i,r}$ in $\mathbf{z}$ is either the most recently written data in $\mathbf{a}_i$ or $\perp$ if the location is yet to be written to. Further, it must be that write regime $\texttt{Reg}$ has been successfully implemented in the execution. Again following CLT, define $\mathsf{Correct}$ as a predicate that takes as input $(\mathbf{y}, \mathbf{z})$ and returns 1 if the outputs $\mathbf{z}$ are correct with respect to $\mathbf{y}$, and 0 otherwise.

Definition 2.1 (OPRAM [CLT16])**.** *An OPRAM (scheme/compiler) $\mathcal{O}$ provides correctness and obliviousness if, for all N, B, T and fixed $\texttt{Reg}$, there exists a negligible function $\mu\colon \mathbb{N} \to \mathbb{R}$ such that for every $\lambda \in \mathbb{N}$, and for every two (parallel sequences) $\mathbf{y}$ and $\mathbf{y}'$ of length T:*

- Correctness:
$$\text{Prob}[\,\mathsf{Correct}(\mathbf{y}, \mathsf{Out}_{\mathcal{O}}(\lambda, N, B, \texttt{Reg}, \mathbf{y})) = 1] \geq 1 - \mu(\lambda),$$
- Obliviousness:
$$\Delta(\mathsf{ACP}_{\mathcal{O}}(\lambda, N, B, \texttt{Reg}, \mathbf{y}), \mathsf{ACP}_{\mathcal{O}}(\lambda, N, B, \texttt{Reg}, \mathbf{y}')) \leq \mu(\lambda).$$

While the work of BCP considered general programs where not all of the m processors need to be active at each time step, we follow the approach of CLT, who consider the situation with all processors responsible for a partition of the storage *and* all participating in each time step (we discuss later the ability for protocols to provide dummy read requests for each client not wishing to make a genuine access). They reference Stefanov *et al.* [SSS12] as the source of the partitioning technique. The BCP approach can still be regarded as using partitioning, however their approach insists that this is in a sense dynamic for each time step: the protocol chooses a representative for each data access.

2.4 Constructing OPRAM

To achieve OPRAM, the compiler appears to need to perform the following steps, with each communication step among clients having fixed topology (and being independent of inputs) and each access to the storage medium being protected by 'regular' ORAM security:

- Identify repeated (colliding) requests, and create dummy requests if any collisions exist. This step was termed the *CPU co-ordination phase* by BCP;
- Perform read accesses for the real and dummy requests, and transmit these values to the requesters;
- Put back or overwrite values in their newly assigned locations, and flush (or equivalent).

The challenge in the third step is to ensure that not only are these writes oblivious from the server, but also the parallel writing procedure does not cause a failure of the ORAM protocol being simulated: this could either occur if a client's local storage becomes too big, or if a client cannot write back to a location (or bucket) that is already full.

To give an idea of the mechanisms required to provide parallelism in a memory-oblivious manner, we now briefly present the ideas behind two constructions from the OPRAM literature, by BCP and CLT. We refer to these works for complete and formal definitions of the concepts described, and also for full exposition of the sub-protocols that are combined to build the OPRAM compilers.

The BCP Construction

First, we briefly detail the construction of BCP [[BCP16], § 3]. The compiler builds OPRAM in a tree-based manner, using oblivious sorting networks [AKS83]. We refer to the three steps above and give general intuition of the inter-client sub-protocols.

The main technical tool to solve CPU (client) co-ordination, the first bullet point above, is an input-independent *oblivious aggregation* procedure allowing CPUs to learn whether two or more want to access the same position, and in the event of a write, decide who gets to proceed (all others do a dummy read and know they're doing a dummy read). First, the clients sort based on the data item they want to access, so that all conflicting accesses are adjacent, then among conflicting clients, the one with the lowest index learns all information about the conflicts (so that it knows who to transmit to later). Then, once the actual accesses have been made, *oblivious multi-cast* allows the designated accessor to send back requested values to the (other) desiring clients. Doing put-backs in a parallel manner requires a different approach to regular tree-based ORAMs, since the root node would overflow – data is thus placed into level $\log m$ of the tree: the trick here is for the clients to pass the data using a (data-independent) routing network that always has $\log m$ rounds, such that at the end each client is responsible for one node at the write-back level of the tree. Simultaneous flushing (pushing data items 'down' towards the leaves as far as they can go, for a random path for each client) requires aggregation, and the client identifier again provides the marker for which client should perform flushing on nodes (this is particularly likely higher up the tree).

The CLT Construction

We focus on the Subtree-O(P)RAM [[CLT16], § 3] construction. Subtree-ORAM is a generalisation of Path ORAM [SvS$^+$13] such that $m \geq 1$ accesses can be made simultaneously: to do this, path fetching, write-back and flushing are all done on an entire subtree to eliminate sequentiality. As in BCP, fake reads must be created to disallow trivially observable data access collisions. Subtree-OPRAM is an instantiation of Subtree-ORAM but with m clients making one access each per round: thus (i) the clients must obliviously agree on the fake reads and write-back locations; (ii) the individual stashes of the clients emulate the single stash in the single-user version; and (iii) parallel flushing emulates the single-client flushing procedure. The idea is to start with a partitioned tree-based ORAM, with each client independently managing its own subtree. To make this work, a partition map is included with the regular position map. To write-back and flush, instead of sequentially placing the new locations back in the tree, the items in the stash are combined with the read locations for all clients and are essentially permuted: each item is assigned a new position and then placed on the furthest down bucket on its path in the previously-read 'global' subtree – this ensures that the re-writing of the accessed subtree (for this timestep) is done completely in one go.

To give more detail, we briefly summarise Path-ORAM and then provide a description of the two-step transformation to Subtree-OPRAM. To implement to storage space for N data blocks, the storage space is represented as a complete binary tree with depth $O(\log N)$. Each node in the tree is a "bucket" that contains a fixed number of $Z = O(1)$ encrypted blocks. In order to hide the access pattern, each data block is assigned to a random path ℓ and stored in some bucket along this path. After each access, the assignment needs to refreshed to a new path ℓ'_i. Each client also keeps an additional (small) memory of overflowing data blocks, known as the stash. Now for Subtree-OPRAM, we assume that $m = 2^l$ and the top $\log m$ levels of the tree are removed (in fact, moved to the stash) turning it into a forest of m complete binary trees $\mathcal{T}_1, \ldots, \mathcal{T}_m$ (each of depth $\log N - \log m$) where each client (partition accessor) is responsible to manage one of these trees. In parallel, each client does the following:

1. Using the global position map, client $\mathcal{C}_i$ finds the path ℓ_i assigned to $\mathbf{a}_i$ and delegates the job of accessing the data item to the responsible client $\mathcal{C}_j$ managing $\mathcal{T}_j$ in which ℓ_i is contained.

2. After retrieving all paths that correspond to requests in the client's partition, these paths form a subtree of $\mathcal{T}_j$. The client can now find the data items either in the subtree or the local stash, and send back the items to the requesting clients.

3. After receiving a data item that it requested, $\mathcal{C}_i$ assigns to $\mathbf{a}_i$ a new path ℓ'_i and delegates the job of writing back to the client responsible for the tree containing ℓ'_i. In the event of a write, the new data to be written is including in the transmission to the partition owner of the new position ℓ'_i.

4. Finally each client runs the subtree-flushing procedure locally on their obtained subtree and own stash to write the subtree back into their partition $\mathcal{T}_i$.

The set of oblivious inter-client communication protocols that enable this procedure are mostly inspired by those given by BCP. (We reiterate that oblivious here means from the perspective of a network eavesdropper that sees ciphertexts.)

2.5 System Assumptions

Here we clarify our setting and briefly discuss some of the choices we have made. We assume a group of m clients who will interact with some central data store ('server') that is capable of storing N' fixed-size data items ('blocks'), plus a router R. We assume R to be a very simple device and we minimise the trust assumptions placed upon it as much as is possible.

The task of R is to prepare the received client access requests in a well-formed manner which includes, e.g., to remove repetitions in accessing the same data items. Note that this routing entity could be an *elected* group of the set of users (with the election occurring in a separate pre-processing phase), or run using multi-party computation between two or more of the users. However, we prefer to aid readability by explicitly assuming this routing entity to be a separate one.[2] All existing ORAM schemes assume at a minimum that plaintext data blocks are encrypted, and it is the ciphertexts that are subject to ORAM operations. All clients possess the (symmetric) key material used to encrypt the data blocks, plus the system parameters (including e.g. encryption algorithms) necessary to implement the compiled OPRAM protocol (and thus interact with the database hosted by the server). The encryption mechanism is assumed to provide semantic security, and the constructions will apply further primitives to the plaintexts

[2]The router can, if required, (i) enable a fully non-interactive system architecture, where the clients only communicate with R and not each other; and/or (ii) assist with the audit trail, in the motivating example of restrictive legislation. In Appendix B we give a brief discussion on non-interactivity in OPRAM schemes.

and ciphertexts involved. We assume that there are pairwise secure communication channels between all m clients, and between the clients and R, however we do not assume that the cost of this communication is free or negligible.

In prior work the clients are given identifiers $\{1, \dots, m\}$ (or more generally elements of some identifier space $\mathcal{ID}$) and the write-conflict regime $\mathtt{Reg}$ is fixed as PRIORITY as defined in Section 2.2. Defining this regime as a parameter means that we also need to make the mechanism for choosing (unique) client identifiers as the designer's prerogative. If identifiers are fixed and known amongst the clients (e.g. the identifier is the location of the client) but $\mathtt{Reg}$ uses some hierarchical mechanism then an adversary may be able to calculate its position in the hierarchy using its requests. In this sense, a random allocation of (unique) identifiers is the simplest setting, but we wish to additionally build protocols that defend against such side channels.

We will only consider tree-based O(P)RAMs in this work, and as such we will often use the terminology (paths, nodes etc.) to reflect this. Two important components of O(P)RAM schemes are the *position map*, that maps positions for the logical accesses $\mathbf{a} \in [N]$ to locations in the storage medium $\mathbf{a}' \in [N']$, and the local *stash*. In our construction we assume that R holds and updates the position map – this is to make the protocol simpler and reduce the challenges invoked by synchronisation. Prior work (such as Path ORAM [SvS+13]) has shown how to recursively store the position map in another ORAM, and while this appears possible in our setting it is not clear if the extra communication rounds required to securely realise this would benefit what is, for the most part, a proof-of-concept. Since our construction functionally emulates the Subtree-OPRAM protocol of CLT, the analysis of stash is inherited from their work.

We have already mentioned the *partitioning* existing in prior work: the storage medium's data locations are (approximately equally) divided into $\frac{N'}{m}$ entries and each of the clients is responsible for making accesses in just one partition. This implies the existence of some *partition map*, where the allocation may either be fixed for all T rounds, or be dynamic. If the storage medium itself is geographically divided then it would certainly make sense for the partitions to be fixed, however we leave the decision for this to the implementer.

3 Client-oblivious OPRAM

Deploying current instantiations of OPRAM would mean that in the (fixed-topology) shuffling phase, that decides which client should be responsible for writing to which location in the database, the records of each client are by design passed between a large number of the other clients. This may be undesirable, and it will often be preferable that an instantiation would limit the sharing between the clients as much as possible. We discuss the unavoidable leakage and give a security model that captures this scenario.

3.1 Inevitable Leakage in OPRAM

Given the system assumptions detailed in Section 2.5, we now indicate what it is possible to hide, and what information must necessarily pass to clients in any protocol that achieves the OPRAM definition (Def. 2.1).

We have fixed that each client can only read and write to one partition, on behalf of the other clients. In each round, parallel requests need to be managed before and after the actual access, and ensuring (at most) two a priori-fixed representatives for each actual access (the read, then the write-back) fulfils this role. This in itself makes client obliviousness without a trusted proxy (à la TaoStore [SZA+16]) more challenging: we seek to minimise the impact. If a client makes a request to a position that is not in their partition, then some other client doing the read will observe the data in this position, and the client writing back will necessarily see the prior content of the cell or the new data being written (though it should not be able to distinguish these cases). This leakage is unavoidable, and even if it was protected in one time-step using some encryption mechanism, the partition-accessing client could of course just read that data item in the next time-step. More important from our point of view is that the identity of the client that originally made the access should be hidden from the partition accessor. We must thus split the ORAM access process into two steps: first the clients read m data items from the storage, and then those data items plus potentially some other items are written back (or overwritten, in the case of writes) and flushed into position.

To mitigate some of the data leakage we wish to avoid, an anonymisation step could occur before any data sharing between the participating clients takes place. The goal here is to hide client identities, as much as is possible without inhibiting functionality, from other clients, but also from any routing entity or other third party. In doing this, the OPRAM protocol's ability to remove repeated entries and return the retrieved values to the correct clients invokes many challenges. For a given round, in the event that multiple clients wish to access the same data item, fake *read* accesses must be created such that requests for a total of m positions are eventually passed into what can be thought of as the non-parallel component of the OPRAM compiler.

Intuitively, we consider a security game in which an adversary tries to learn or infer some information that was not passed via its (set of) corrupted client(s). The adversary $\mathcal{A}$ provides some parameters: the number of clients, the size of the database, and the number of 'rounds' (time steps) of the program (sequence of accesses). Then, the challenger constructs a program based on these parameters, with random data for writes. The compiler then runs, turning this program into an interactive OPRAM protocol. For each round, $\mathcal{A}$ receives the transcript of all communication that it has elected to see, as defined by a corruption strategy it provides. Then, $\mathcal{A}$ must submit its output of an access that it believes was made: a client, a data position and a round. Since $\mathcal{A}$ will see accesses for its corrupted clients in their partitions, we normalize $\mathcal{A}$'s advantage by the number of uncorrupted clients in the round it gave, as output.

Given these concerns, we wish to design schemes that give the following protections simultaneously:

- The entity hosting the server should not be able to infer anything beyond the number of accesses, i.e. regular ORAM security;
- The entity hosting the server should not be able to distinguish parallel requests (i.e. multiple requests to the same position, compared with the same number of requests to distinct positions), i.e. regular OPRAM security;
- The protocol should be *client oblivious*: For any access that a client did not make itself, it should not learn:
  - the originating client
  - the position being read, for positions outside of its partition

A possible extension to client obliviousness is to also capture the data being written, however: (1) formalising this in a definition is very challenging, and (2) our construction does not cover this and cannot be easily extended to do so.

We now give a definition for client obliviousness which captures these properties: clients cannot learn information about other clients' accesses, beyond the inevitable leaks discussed above. In Section 3.3 we discuss some potential extensions to our definition.

3.2 Client Obliviousness for OPRAM

We cast CO as a game-based notion: this allows more fine-grained corruption of clients, however this necessitates care regarding win conditions. Our construction uses public-key primitives and so we require a computational adversary, moving away from the statistical security definitions in many areas of the

```
$\mathbf{Exp}^{\mathsf{CO\text{-}OPRAM}}_{\mathcal{O},\,\mathcal{A}}(\lambda)$:
1 :  $(m, T, N, \mathsf{CorrStrat}) \leftarrow \mathcal{A}$
2 :  for $i = 1, \ldots, m$ do
3 :    for $j = 1, \ldots, T$ do
4 :      $op_{i,j} \leftarrow_{\$} \{\text{read}, \text{write}\}$
5 :        if $op_{i,j} = \text{write}$ do
6 :          $v_{i,j} \leftarrow_{\$} \{0,1\}^B$
7 :        else $v_{i,j} \leftarrow \perp$
8 :      $\mathbf{a}_{i,j} \leftarrow_{\$} \{1, \ldots, N\}$
9 :      $\mathbf{y}_{i,j} \leftarrow \mathsf{Acc}(op_{i,j}, \mathbf{a}_{i,j}, v_{i,j})$
10 :   $\mathbf{y}_i \leftarrow \{\mathbf{y}_{i,1}, \ldots \mathbf{y}_{i,T}\}$
11 : $\mathbf{y} \leftarrow \{\mathbf{y}_1, \ldots, \mathbf{y}_m\}$
12 : $(\mathsf{ACP}_{1,1}, \ldots, \mathsf{ACP}_{m,T}) \leftarrow \mathcal{O}(\mathbf{y})$
13 : for $i = 1, \ldots m$ do
14 :   if $\mathsf{CorrStrat}[i] \neq \perp$ then
15 :     $\forall j \geq \mathsf{CorrStrat}[i]$ do
16 :       $\mathsf{Trn}_{\mathcal{A}} \leftarrow \mathsf{Trn}_{\mathcal{A}} \cup \{\mathsf{ACP}_{i,j}\}$
17 : $(id, \mathbf{a}, r) \leftarrow \mathcal{A}(\mathsf{Trn}_{\mathcal{A}})$
18 : if $\exists \mathsf{Acc}(\cdot, \mathbf{a}_{id,r}, \cdot) \in \mathbf{y}$ then
19 :   return 1
20 : else return 0
```

Figure 1: Client Obliviousness security experiment.

O(P)RAM literature. Our game-based security experiment for CO is given in Fig. 1. The idea is that an adversary submits a set of parameters, which specifies the number of clients and rounds. Then the game will, for each access, choose read or write, choose a location, and if a write choose some data. The adversary specifies its corruption strategy, e.g. client 7 from round 3 onwards, client 2 from round 6 onwards: this is a vector $\mathsf{CorrStrat}$ of m elements, where entries are either a round number $\{1, \ldots, T\}$ or $\perp$. In doing so, the adversary specifies the points from which it sees a 'decrypted' version of each client's transcript. Finally, it outputs a triple: a client identifier, a position in the ORAM and a round identifier. If the adversary had corrupted that client before that round, then it trivially loses.

Since we assume that all clients are active in all rounds, if the adversary has corrupted a client and does not see any accesses to its partition then it learns that none of the clients accessed its data items. Further, since a client has to read (in cleartext) requests in its own partition, the adversary can just corrupt one client, wait until it is asked to make a request in its own partition (on average one per round) then output that data item with a random other client identifier, and win with probability upper-bounded[3] by $\frac{1}{m-1}$. This means we must normalise the success probability by the number of uncorrupted clients in the round that was output by the adversary: CC_r (number of corrupted clients in round r) is calculated by incrementing a counter once for every entry in $\mathsf{CorrStrat}$ smaller than or equal to r.

Definition 3.1 (Client Obliviousness (game-based)). *$\mathcal{O}$ is a* Client Oblivious Oblivious Parallel RAM (CO-OPRAM) *compiler if there exists no adversary with non-negligible advantage in the following sense:*

$$\mathbf{Adv}^{\mathsf{CO\text{-}OPRAM}}_{\mathcal{O},\,\mathcal{A}}(\lambda) = \left|\text{Prob}\left[\mathbf{Exp}^{\mathsf{CO\text{-}OPRAM}}_{\mathcal{O},\,\mathcal{A}}(\lambda) = 1\right] - \frac{1}{m - \mathsf{CC}_r}\right|,$$

where experiment $\mathbf{Exp}^{\mathsf{CO\text{-}OPRAM}}_{\mathcal{O},\,\mathcal{A}}(\lambda)$ is given in Fig. 1 and CC_r is defined as above.

3.3 Extensions to Client Obliviousness Model

In this section, we discuss some potential extensions to this representation of client obliviousness.

A hierarchy of adversarial power exists for game-based definitions of client obliviousness (CO). Specifically, an adversary can:

[3]Note that any protocol achieving regular OPRAM security needs to (at a minimum) produce one fake read every time that an access collision occurs. If the protocol hides which reads are real and which are fake from the reading clients, and the probability of access collisions is high, then this probability may be much smaller.

- have access to 'decrypted accesses' of one client
- have access to 'decrypted accesses' of k clients, for $k \in [m]$
- (Section 3.2) have access to 'decrypted accesses' of k clients, plus can specify from which point onwards it wants to corrupt each one (thus being able to win by outputting one of these clients before corruption time)
- adaptively corrupt clients as the game progresses

We believe that the corruption power that we presented in Section 3.2 is the most interesting and natural, and discuss here how it is possible to mitigate against any additional powers.

Intuitively the client obliviousness game requires that the adversary can not distinguish which of the other clients made memory accesses to its portion of the memory. We now discuss some variants of this definition.

The definition that we present is in a sense analogous to *one-wayness*: the adversary is required to produce a triple where for any two values being fixed there is only one valid entry in the third value. One potential modification is to move towards an **indistinguishability**-based notion, however there does not appear to be any direct analogue to our game, since the adversary could not provide two programs. A similar argument follows for real-or-random notions: it is not clear what either execution would consist of. It is possible to ask subtly different questions, e.g. define security if the adversary can not tell if two clients made an access to the same memory location, however the relationship between such a definition and ours is not immediate.

As mentioned earlier, corruption in our definition is static: the adversary provides one vector indicating its strategy. The setting of **adaptive corruptions** does not appear to be much more difficult to provide security in, since the adversary has no power over the logical accesses being executed. However it is not clear if there is a separation result here (a protocol that is secure in the static setting but insecure in the adaptive setting), mainly because we are only aware of one construction (in Section 4) that meets the static CO notion.

We assume that the clients, the server and any additional parties operate according to the protocol, and even after the adversary corrupts a client the adversary can then perform computations based on the information it receives via the transcripts. Extending to **malicious clients** that can arbitrarily deviate would require some additional assumption that this behaviour retains correctness. It may be possible (and efficient) for the challenger to check this for some protocols however in general this may be very challenging, and any security reductions would need to take this into account – further it is not apparent if this strengthening is necessarily well motivated in our motivating scenario.

Security properties that capture the 'information learned' by some adversary are often presented using a **simulation-based definition**. In our setting this does not appear to help, since the simulator's task of extracting the adversary's inputs is very challenging. Providing security proofs for schemes in this framework is left as an open problem.

4 Construction

Removing the *inter-client communication* between entities in a parallel ORAM scenario where multiple clients access data will often be a desirable property in the realm of the system architecture that we consider. However, current OPRAM schemes inherently require the clients to communicate with each other in order to access the requested data item within the ORAM. Achieving the above notion of client-obliviousness is not a straightforward task and hence, our system assumptions (described in Section 2.5) have been made to enable a clear exposition here.

In this section, we provide a detailed overview of our construction which (functionally) emulates the Subtree-OPRAM protocol of CLT (see Section 2.4), with major modifications to the sub-protocols to ensure client obliviousness. We now set the scene and begin with m clients $\mathcal{C}_1, \dots, \mathcal{C}_m$ and routing entity R, where each client is "responsible" for accessing data within one distinct part (partition) of the ORAM (data storage), even on behalf of the other clients. We assume that all m clients share a secret symmetric key for a semantically-secure encryption scheme, with which the data blocks are encrypted. Note that the router does not possess this key. Each client now wishes to execute an operation (either read or write) to a data item within the ORAM. These clients wish to store N items (each of size B, for N a power of two) on a server by using N' cells with the assistance of R. Following our motivating scenarios for this type of protocol, some type of router or equivalent infrastructure will already exist, and we will simply make use of it. The protocol (specifically, R) will implement some write-conflict regime $\mathtt{Reg}$ (see Section 2.2).

Our protocol follows Subtree-OPRAM in organising the server storage as a forest of m complete binary trees $\mathcal{T}_1, \dots, \mathcal{T}_m$ of depth $\log N - \log m$ where each node in each separate tree contains a bucket of blocks in which data items can be stored. As usual, we identify a path with a leaf in the tree. By $\mathsf{Pos.Map}$, we denote the position map that maps the locations $\mathbf{a} \in [N]$ to the leaves in the server storage. Each client $\mathcal{C}_i$ is responsible for handling a partition of the ORAM, namely the corresponding tree $\mathcal{T}_i$. This means that $\mathcal{C}_i$ executes reads and writes to all leaves (i.e. paths) that belong to this tree, and each client needs to locally manage a stash Stash_i to store overflowing blocks whose path belongs to $\mathcal{T}_i$. Note that the top $\log m$ levels of the tree that have been initially removed are incorporated into the stashes of the clients. This means that the tree – the combination of the subtrees and the shared stash for upper levels – is a complete binary tree with no 'overlap' between the partitions. Further, the union of all client stashes emulates the single stash in the Subtree-ORAM protocol. R has read and write access to the position (and thus partition) map[4]. Note however that it is necessary for each client to have read access to the partition map for its own partition, in order to make reads and flush appropriately: consequently we consider this minimal requirement as a system assumption..

4.1 Client-oblivious OPRAM construction

We will use in the following a public-key encryption scheme $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and a (one-time) symmetric encryption scheme $\mathsf{SKE} = (\mathsf{KG}, \mathsf{E}, \mathsf{D})$. We also assume the existence of a TOR-style onion-routing network to anonymously route accesses between the clients such that the router R does not learn the origin of each request. Our exposition here asks that the clients themselves use a (size m) TOR-style network, however this is not inherent, and the clients could use any (larger) network as long as all clients act as exit nodes. Indeed an external onion routing may often be preferable, as it does not assume some fixed identification system of the clients. For a formal treatment of TOR, we refer to [DMS04, DS18].

We now describe the execution of the protocol in a given round. Each of the m clients and R initially run Gen to generate a key-pair $(pk_{\mathcal{C}_i}, sk_{\mathcal{C}_i})_{i \in [m]}$ and $(pk_{\mathsf{R}}, sk_{\mathsf{R}})$, respectively. Additionally, each client generates a one-time symmetric key $\tilde{k}_{\mathcal{C}_i}$. We will also require the router R to sample fake key material of equal length as the one-time symmetric keys – this requires that the size of the output of KG is a constant. Each client $\mathcal{C}_i$ produces/provides a logical access request of the form $\mathsf{Acc}(op_i, \mathbf{a}_i, v_i)$ and the m clients proceed in parallel to process the m logical accesses:

1. **Preparing the access request in an anonymised fashion.** The aim of this first phase is to *anonymise* each client's access to the ORAM from the other clients as well as the router R. We achieve this through a combination of a TOR-style mechanism sending the requests to the router and also using random client identifiers to hide the client's identity. We assume that all clients are

[4]We make this assumption to more closely represent our motivating scenarios; in the event that the full position map is known to all clients then Step 6 in our construction can be replaced by a broadcast.

active TOR nodes and that the initial setup has taken place before the start of the protocol. Each client chooses a random identifier from the identifier space via $id_{\mathcal{C}_i} \leftarrow_{\$} \mathcal{ID}$ and generates a one-time symmetric key via $\tilde{k}_{\mathcal{C}_i} \leftarrow_{\$} \mathsf{KG}(1^\lambda)$. The purpose of this key is to encrypt the retrieved data item and then simply broadcast the ciphertext. Since this ciphertext is encrypted under the client's one-time symmetric key, only this requesting client can successfully decrypt it. The client chooses a random route of three TOR relays $\mathsf{tor}_1, \mathsf{tor}_2, \mathsf{tor}_3 \in [m]$ and establishment of this circuit will result in the generation of symmetric keys $k_{\mathsf{tor}_1}, k_{\mathsf{tor}_2}, k_{\mathsf{tor}_3}$ for each of the nodes. Next it prepares the onion encryption as

$$\mathsf{E}_{k_{\mathsf{tor}_1}}\left(\mathsf{E}_{k_{\mathsf{tor}_2}}\left(\mathsf{E}_{k_{\mathsf{tor}_3}}(\mathsf{R}\|\mathsf{Enc}_{pk_{\mathsf{R}}}(id_{\mathcal{C}_i}\|\tilde{k}_{\mathcal{C}_i}\|\mathsf{Acc}(op_i, \mathbf{a}_i, v_i)))\right)\right)$$

and sends this ciphertext through the chosen route: each node decrypts one layer after another until the inner (public-key) encryption arrives at R.

2. **Digesting the access requests.** After having received m ciphertexts, R first decrypts all of them and checks whether there are any collisions between the client identifiers. If so, it will abort and send each client $\perp$ to indicate that the protocol failed. Otherwise, the router continues, and first must handle access collisions. In the event of any access collision, i.e. $\tilde{m}$ clients ($\tilde{m} \geq 2$) wishing to access a location, R must create $\tilde{m}-1$ fake reads by selecting a random location $\mathbf{a} \leftarrow_{\$} [N]$, setting $op = \text{read}$ and $v = \perp$. In the event of a write collision, i.e. $\tilde{m}'$ clients ($\tilde{m}' \geq 2$) wishing to write to a location, R must enforce $\mathtt{Reg}$ to decide which clients (if any) get to write. In summary, R will turn the m logical access requests that it decrypted into m actual accesses, and appending to each access a record of the client (identifiers), if any, that actually requested the location. Using the position map, R can determine which accesses need to be executed by which partition accessor, i.e., it determines the path ℓ_i to which each request corresponds. In more detail: for each received request, R simply fetches the information about the path from the position map, i.e., $\ell_i = \mathsf{Pos.Map}(\mathbf{a}_i)$. Then it sets $\mathbf{a}'_i = \mathbf{a}_i$, and immediately refreshes the position map $\mathsf{Pos.Map}(\mathbf{a}_i)$ to a new randomly assigned path $\ell'_i \leftarrow_{\$} [N]$. This new path might fall into another client's partition. Hence, we add also the information to which client the block needs to be re-routed later since the clients themselves do not have access to the full position map. Note that for write requests, the behaviour here still applies: the client expects to receive the *old* data item in return, so the router is also required to fetch the corresponding path. If the determined path belongs to tree $\mathcal{T}_j$ then this means that R needs to prepare an access request to the partition accessor $\mathcal{C}_j$. In order to keep it oblivious from the partition accessor how many clients wish to access the data item in $\mathcal{T}_j$, we force the router to include m many one-time symmetric keys into the request. Here we distinguish between *valid keys*, i.e., keys that were initially sent by the requesting client, and *fake keys* which are generated by R to simply keep the partition accessor busy without learning how many clients really wish to access this particular data item. We distinguish three cases to clarify our approach. The first case is that only *one* client wishes to access a particular data item in the handled partition of $\mathcal{C}_j$. This means that R has received a "proper" one-time key generated from $\mathcal{C}_i$, namely $\tilde{k}_{\mathcal{C}_i}$. For the remaining $m-1$ keys, the router simply samples $m-1$ many fake keys $fk_1, \ldots, fk_{m-1}$ from a key space $\mathcal{K}$ ensuring that they are all of equal length to $\tilde{k}_{\mathcal{C}_i}$. Being equipped with those keys, R now prepares the access instruction for the partition accessor, i.e., $(op_i, \mathbf{a}'_i, v_i, \ell_i, \ell'_i\|\mathcal{C}_k, \pi(\tilde{k}_{\mathcal{C}_i}\|fk_1\|\ldots\|fk_{m-1}))$ where π permutes the keys such that the partition accessor does not know which key is valid. The second case deals with the more general version of the above where we have an *access collision* for a data item, i.e., multiple clients wish to access the *same* data item. This boils down to the router to include as many real keys as the number of clients who requested the same location plus filling up the remaining key material with fake keys until a total of m keys are included in the request. The third case is when the read operation is fake, so no clients made this logical access request. In this case, R must simply produce m fake keys. Finally, R encrypts

the access instruction for a data item under the partition accessor's public key, i.e.

$$\mathsf{Enc}_{pk_{\mathcal{C}_j}}\left(op_i, \mathbf{a}'_i, v_i, \ell_i, \ell'_i \| \mathcal{C}_k, \pi\left(\{\tilde{k}_{\mathcal{C}_p}\}_{p\in[\tilde{m}]} \| \{fk_q\}_{q\in[\hat{m}]}\right)\right)$$

where we decompose the number of clients as $m = \tilde{m} + \hat{m}$ while $\tilde{m}$ corresponds to the number of clients making a real access and $\hat{m}$ corresponds to the number of fake keys that need to be included. Finally, the router sends the resulting ciphertext to $\mathcal{C}_j$. Note that the transmission of this ciphertext does not need to happen in a Tor-style manner since each partition accessor knows from the protocol that they will receive prepared (encrypted) accesses by the router.

3. **Accessing the paths.** Each client $\mathcal{C}_j$, $j \in [m]$, now receives a set of ciphertexts including the accesses it must make. The client starts with decrypting them using its secret key $sk_{\mathcal{C}_j}$. In the next step, all the requested paths are retrieved and batched[5] as a set S_j. Then, proceed as follows:

   i The partition accessor retrieves all paths in S_j which form a subtree $\mathcal{T}_{S_j}$.

   ii For each request $(op_i, \mathbf{a}'_i, \ell_i)$, the partition accessor finds the block $\mathbf{a}'_i$ in either $\mathcal{T}_{S_j}$ or Stash_j with data item $\bar{v_i}$ and keeps it locally, and deletes it either in the tree or stash.

4. **Multicast data items.** After having retrieved the requested data item $\bar{v_i}$, the partition accessor prepares m encryptions of the data item using the m keys received from R. After having prepared all ciphertexts, the partition accessor simply broadcasts all of them. Note that only the ciphertexts that were generated with a valid one-time key can be decrypted by the respective requester: the decryption of all other ciphertexts will fail with overwhelming probability since they have been generated using fake keys.

5. **Retrieve data items.** Client $\mathcal{C}_i$ now fetches all ciphertexts and starts trial decrypting them all using one-time key $\tilde{k}_{\mathcal{C}_i}$. (As soon as one ciphertext has successfully decrypted $\mathcal{C}_i$ can stop, since only one data item was requested.)

6. **Re-route blocks.** Each partition accessor has also received the information in Step (2) to which path ℓ'_i the data item needs to be routed. Since the client does not have access to the full position map, the router has initially provided the identity of the client $\mathcal{C}_k$ to whom the blocks must be sent. For each retrieval made in step (3), the partition-owning client prepares $m-1$ encryptions[6]

$$\mathsf{Enc}_{pk_{\mathcal{C}_i}}(\mathtt{msg}), \text{ where } \mathtt{msg} = \left\{ \begin{array}{ll} (\ell'_i, \mathbf{a}'_i, \tilde{v_i}) & \text{for } i = k, \\ \mathtt{str} & \text{otherwise.} \end{array} \right\}$$

   where $\tilde{v_i} = v_i$ if $op_i =$ write, and $\tilde{v_i} = \bar{v_i}$ if $op_i =$ read, and $\mathtt{str}$ is some fixed string of length equal to $\ell'_i || \mathbf{a}'_i || \tilde{v_i}$. Then, the client sends these $m-1$ equal-length ciphertexts to the respective public key holders.

7. **Flush subtree and write-back.** Each client $\mathcal{C}_k$ tries to decrypt each of the m ciphertexts received in step (6), to learn which newly assigned paths must be written back to in its partition. After successfully obtaining this information, each client runs the flushing procedure on all real-read paths and the stash. Finally, the client writes back subtree $\mathcal{T}_{S_k}$. If at any point the Stash contains too many blocks then the procedure outputs "overflow".

[5]Note that it is possible that many clients have requested multiple data items held by one partition accessor. Therefore we batch all data item in a set.

[6]Reminder that clients may read multiple paths (in a round) in their partition, or none. Further, the new location of the path may be in the same partition as it was read from, and in this case the client only generates 'dummy' encryptions of `str`.

4.2 Analysis of Our CO-OPRAM Protocol

In this section, we provide details of why the construction given in Section 4.1 is correct and satisfies obliviousness. Since our construction is essentially built to emulate Subtree-OPRAM – and crucially all sub-protocols are *functionally* the same – the main arguments of CLT regarding correctness and stash analysis simply apply also to our scheme. Obliviousness is more tricky, as we have introduced new components that mimic the operation of the sub-protocols: we just need to argue that these components leak only as much as the original scheme. A crucial component of this is fixing the topology of the communication: the communication pattern seen by the OPRAM obliviousness adversary in each sub-protocol should be independent of the inputs. As CLT observe, this means that they are oblivious in a very strong sense, and unfortunately we cannot inherit this in our scheme. The 'vulnerable' communications in our protocol are as follows, indicating in which step of the construction the communication occurs:

(1.) The (onion-encrypted) messages sent from clients to R;

(2.) The access instructions sent to the (reading) partition accessors;

(4.) Multicasting the results;

(6.) Re-routing from reading partition accessors to writing partition accessors.

The first step involves three ciphertexts per request, sent and forwarded by two random clients. Since at this stage there is no link between these pre-processed requests and the accesses to be made, these requests appear to be independent of the inputs from the perspective of an adversary seeing only ciphertexts. For the second set of messages, which are again of fixed size, the adversary learns nothing other than what it is about to learn from the subsequent path reads (assuming that the partition map is known to the adversary). In the multicast stage, the messages sent by each partition accessor are again of fixed size, and assuming security of the one-time encryptions this is again a fixed communication pattern. Finally, the re-routing mechanism relies on the strength of the PKE scheme, the fact that messages are fixed size and the fact that these clients write back anyway.

Client Obliviousness

Finally, we need to argue that our construction satisfies client obliviousness. This is based on the strength of the one-time symmetric scheme $\mathsf{OT\text{-}SKE}$ that we employ in step 4, the PKE scheme PKE used in steps 1 and 6, and the TOR-style encryption[7] $\mathsf{MT\text{-}SKE}$ used in steps 1 and 6 (we simply assume security of the the block-encryption scheme used for the data items).

Fig. 2 details the transcript for client i in a single round of the protocol. As the proof proceeds, we indicate which lines the challenger modifies in each game.

Theorem 4.1. *Let $\mathcal{O}$ be the OPRAM protocol given in Section 4.1, built using* $\mathsf{OT\text{-}SKE}$*,* PKE *and* $\mathsf{MT\text{-}SKE}$*. For any adversary $\mathcal{A}$ against the client obliviousness* ($\mathsf{CO\text{-}OPRAM}$) *of $\mathcal{O}$, there exist adversaries $\mathcal{B}$, $\mathcal{B}'$ and $\mathcal{B}''$ against the one-time symmetric encryption scheme, the public-key encryption scheme, and the TOR-style symmetric encryption scheme, respectively, such that*

$$\mathbf{Adv}^{\mathsf{CO\text{-}OPRAM}}_{\mathcal{O},\ \mathcal{A}}(\lambda) \leq \mathbf{Adv}^{\mu\mathsf{ind\text{-}ote}}_{\mathsf{OT\text{-}SKE},\ \mathcal{B}}(\lambda) + \mathbf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE},\ \mathcal{B}'}(\lambda) + \mathbf{Adv}^{\mu\mathsf{ind\text{-}mte}}_{\mathsf{MT\text{-}SKE},\ \mathcal{B}''}(\lambda).$$

[7]Note that in the TOR-style encryption that transmits the user accesses to R: if the adversary has corrupted all nodes in the TOR circuit for an access by an uncorrupted client then the adversary will be able to win the CO game – this is why we need to normalize the win probability in Definition 3.1.

```
Transcript for client $i$:
1 : $\mathsf{Acc}(op_i, \mathbf{a}_i, v_i)$
2 : $id_{\mathcal{C}_i} \leftarrow_\$ \mathcal{ID}$; $\tilde{k}_{\mathcal{C}_i} \leftarrow_\$ \mathsf{KG}(1^\lambda)$
3 : $\mathsf{tor}_1, \mathsf{tor}_2, \mathsf{tor}_3 \leftarrow_\$ [m]$
4 : $C_a \leftarrow \mathsf{Enc}_{pk_\mathsf{R}}(id_{\mathcal{C}_i} \| \tilde{k}_{\mathcal{C}_i} \| \mathsf{Acc}(op_i, \mathbf{a}_i, v_i))$
5 : $ct_b \leftarrow \mathsf{E}_{k_{\mathsf{tor}_3}}(\mathsf{R} \| C_a)$
6 : $ct_c \leftarrow \mathsf{E}_{k_{\mathsf{tor}_2}}(ct_b)$
7 : $ct_d \leftarrow \mathsf{E}_{k_{\mathsf{tor}_1}}(ct_c)$
8 : Receive $s \in \{0, \ldots, 3m\}$ TOR ciphertexts.
9 : For ciphertexts $\in \{1, \ldots, s\}$, decrypt, and route to designated client or $\mathsf{R}$
10 : Receive $t \in \{0, \ldots, m\}$ access requests from $\mathsf{R}$, decrypt using $sk_{\mathcal{C}_i}$
11 : For each request $\in \{1, \ldots, t\}$, read path, find block data, store locally
12 : For each read path, encrypt block data $\bar{v_i}$ under all $m$ keys
13 : Receive $m$ ciphertexts
14 : Decrypt all under $\tilde{k}_{\mathcal{C}_i}$
15 : For each read path, encrypt $(\ell'_i, \mathbf{a}'_i, \tilde{v_i})$ under $pk_{\mathcal{C}_k}$ and str under all other public keys
16 : Receive $m - 1$ ciphertexts, decrypt with $sk_{\mathcal{C}_i}$
17 : For each valid writeback request received, write $\tilde{v_i}$ to $\ell'_i$
```

Figure 2: Transcript for client i in exection of OPRAM protocol given in Section 4.1.

Proof. Note that in the CO-OPRAM game, in each round, the adversary is given transcripts for the clients that it corrupts, and nothing for those it has not. $\mathcal{A}$ produces $(id, \mathbf{a}, r)$ and wins if client id actually sent a logical request for location $\mathbf{a}$ in round r, and $\mathcal{A}$ had not corrupted client id before or during round r. We seek to assess what information the adversary can glean from its transcripts alone, since this is all the adversary actually gets. In particular, if it learns the plaintext sent in Step (4) via the one-time symmetric encryption scheme, for any clients that it has not corrupted, then this would directly lead to the ability to produce a winning output. (If it could decrypt any of the messages sent by R to uncorrupted clients in Step (2) then this would also result in a win, however this is not part of the transcripts except for corrupted clients.)

We require multi-user indistinguishability for symmetric encryption (μind-ote), which mimics the encryption under one-time keys in the multicast step of our construction, and the multi-time variant (μind-mte) for the TOR-style encryption. We additionally also require the standard notion of real-or-random ind-cpa security for a public-key encryption scheme in the anonymisation step of our construction. Definitions and security experiments for these games are given in Appendix A.1 and A.2.

In order to show that our proposed construction achieves our property of client obliviousness, we apply the game-hopping technique. The games are specified as follows:

- **Game 0.** This game simply corresponds to the original CO-OPRAM game.
- **Game 1.** Same game as Game 0 except that the challenger now replaces all ciphertexts that were generated in Step (4) of the construction, and hence are part of the transcript, with encryptions of random messages, *except* for the ones that the clients should be able to decrypt.
- **Game 2.** Same game as Game 1 except that the ciphertexts that were generated in Step (1) are now modified, and hence different in the transcript which the adversary receives. Here we replace

the innermost encryption of the TOR-style encryption mechanism, that is the public-key encryption under the public key of R, with the encryption of a random message, with the restriction that the client assigned to k_{tor_3} is not corrupted.

- **Game 3.** Same as Game 2 except the challenger swaps the appropriate elements of the TOR-style mechanism to encryptions of random messages.

We start with analysing the hop from Game 0 to Game 1. The difference between both games only relies on the fact that the encryptions generated in Step (4) of the construction are replaced with encryptions of random messages (except for the ones that are generated for a corrupted client). Let $\mathcal{A}$ be the adversary playing against the $\mathsf{CO\text{-}OPRAM}$ game. We build a reduction $\mathcal{B}$ playing against the $\mu\mathsf{ind\text{-}ote}$ game and running $\mathcal{A}$ as a sub-routine. On input of the security parameter, $\mathcal{A}$ begins with specifying its corruption strategy, as well as the remaining parameters such as number of clients and number of rounds. $\mathcal{B}$ simulates everything appropriately following the protocol specification, i.e., it runs lines 2 to 12 in the $\mathsf{CO\text{-}OPRAM}$ game for the compiler $\mathcal{O}$. After obtaining the appropriate data items, the reduction needs to add to the transcript a set of ciphertexts (corresponding to line 12 in Fig. 2) generated from a symmetric encryption scheme. Since $\mathcal{B}$ cannot generate them itself, it needs to embed, i.e. pass them to its own symmetric encryption oracle E. Depending on the bit b, this encryption oracle either returns a genuine encryption in case $b = 1$, or an encryption of a random message otherwise. The obtained ciphertexts are added to the transcript and given to the adversary. Eventually, $\mathcal{A}$ outputs its guess of client identifier, position in the ORAM and round identifier. If this is correct then $\mathcal{B}$ outputs $b' = 1$, else $b' = 0$.

Reduction $\mathcal{B}$ is efficient since all simulated steps are performed in polynomial-time. Depending on the bit b chosen by the challenger for the encryption oracle E, $\mathcal{B}$ either simulates Game 0 or Game 1. Next we analyse the probabilities for $\mathcal{A}$ winning in the respective games. Observe that the probability of $\mathcal{A}$ winning in Game 0 corresponds of course to the probability of winning the $\mathsf{CO\text{-}OPRAM}$ game. Analysing the probability of Game 1 gives that the adversary can win this game if it wins $\mathsf{CO\text{-}OPRAM}$ game and also can distinguish what encryptions it has received from the reduction in its transcript. Hence a simple calculation of probabilities gives us that $|\Pr[G_0 \Rightarrow 1] - \Pr[G_1 \Rightarrow 1]| \leq \mathbf{Adv}^{\mu\mathsf{ind\text{-}ote}}_{\mathsf{SKE},\,\mathcal{B}}(\lambda)$.

In the next step, we need to analyse the game hop from Game 1 to Game 2. The main difference basically lies in the fact that we now swap genuine public-key encryptions from Step (1) to public-key encryptions of random messages. For this, we again build a reduction $\mathcal{B}'$ that depending on the bit b of its encryption oracle Enc either simulates Game 1 or Game 2. $\mathcal{B}'$ simulates everything appropriately following the protocol specification, i.e., it runs lines 2 to 12 in the $\mathsf{CO\text{-}OPRAM}$ game for the compiler $\mathcal{O}$. The main changes occur in the transcript. In more detail, $\mathcal{B}'$ executes lines 1 to 3 as detailed in Fig. 2. When it comes to line 4 – and only where the TOR exit node (client assigned to k_{tor_3}) is uncorrupted – the reduction sends $(id_{\mathcal{C}_i} \| \tilde{k}_{\mathcal{C}_i} \| \mathsf{Acc}(op_i, \mathbf{a}_i, v_i))$ to its encryption oracle Enc. In case the bit for the oracle corresponds to $b = 1$, then the oracle will output an genuine encryption of the provided string. Otherwise, it simply outputs an encryption of a random string. Next the reduction continues with prepring the TOR-style onion encryptions in lines 5 to 7 in Fig. 2. The obtained ciphertexts are added to the transcript and given to the adversary. Eventually, $\mathcal{A}$ outputs its guess of client identifier, position in the ORAM and round identifier. If this is correct then $\mathcal{B}'$ outputs $b' = 1$, else $b' = 0$.

Reduction $\mathcal{B}'$ is efficient since all simulated steps are performed in polynomial-time. Depending on the bit b chosen by the challenger for the (public key) encryption oracle Enc, $\mathcal{B}'$ either simulates Game 1 or Game 2. Following a similar analysis as before, $|\Pr[G_1 \Rightarrow 1] - \Pr[G_2 \Rightarrow 1]| \leq \mathbf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE},\,\mathcal{B}'}(\lambda)$.

Let us now move on to analyse the game hop between Game 2 and Game 3. The main difference is that we now replace the TOR-style onion encryption[8] under the appropriate input messages to random

[8]We assume that this TOR-style encryption mechanism provides onion encryption with labels, allowing clients (decrypting nodes) to know whether to stop (if they are an exit node) or continue (and to which node to send the inner ciphertext).

messages (in lines 5 to 7). For this, we again build a reduction $\mathcal{B}''$ that depending on the bit b of its encryption oracle E either simulates Game 2 or Game 3. $\mathcal{B}''$ simulates everything appropriately following the protocol specification, i.e., it runs lines 2 to 12 in the $\mathsf{CO\text{-}OPRAM}$ game for the compiler $\mathcal{O}$. Reduction $\mathcal{B}''$ now executes lines 1 to 4 as specified in the transcript. When it comes to lines 5 to 7, it passes the requests to its encryption oracle for any TOR circuit in which an adversarial client is not the exit node (assigned to k_{tor_3}). In case the bit $b = 1$ then the oracle returns genuine encryptions, otherwise it returns encryptions under random messages. The obtained ciphertexts are added to the transcript and given to the adversary. Eventually, $\mathcal{A}$ outputs its guess of client identifier, position in the ORAM and round identifier. If this is correct then $\mathcal{B}''$ outputs $b' = 1$, else $b' = 0$.

Reduction $\mathcal{B}''$ is efficient since all simulated steps are performed in polynomial-time. Depending on the bit b chosen by the challenger for the encryption oracle E in $\mathsf{MT\text{-}SKE}$, $\mathcal{B}''$ either simulates Game 2 or Game 3. Following a similar analysis as before, $|\Pr[G_2 \Rightarrow 1] - \Pr[G_3 \Rightarrow 1]| \leq \mathbf{Adv}^{\mu\mathsf{ind\text{-}mte}}_{\mathsf{MT\text{-}SKE},\ \mathcal{B}''}(\lambda)$.

Finally, we analyse the probability of the adversary winning in Game 3. Here the adversary has no information that enables it to win the $\mathsf{CO\text{-}OPRAM}$ game: in a given round, the only remaining 'real' data belongs to accesses of clients that the adversary has corrupted. We conclude that its win probability in the experiment is normalized to zero, i.e. $\Pr[G_3 \Rightarrow 1] = \text{Prob}\left[\mathbf{Exp}^{\text{Game 3}}_{\mathcal{O},\ \mathcal{A}}(\lambda) = 1\right] = \frac{1}{m - \mathsf{CC}_r}$ and thus $\mathbf{Adv}^{\text{Game 3}}_{\mathcal{O},\ \mathcal{A}}(\lambda) = 0$, for m and CC_r as defined in Section 3.2. Summing up the above gives the expected bound. $\square$

4.3 Comparison to CLT construction

We now compare our construction given in Section 4.1 more closely with the construction of CLT. As stated earlier, our protocol functionally emulates the operation of subtree-OPRAM, however, the sub-protocols performed by the clients are replaced by the steps above, with the addition of the router R. Recall that the Subtree-OPRAM protocol of CLT requires to run a sub-protocol $\mathsf{OblivElect}$ where a representative (for a particular data item $\mathbf{a}$) between all m clients is elected – this client is not necessarily the partition accessor where respective accesses have been made. The representative receives the data items from the partition accessor and is responsible to distribute them along the requesting clients. This process is highly interactive and defeats the purpose of our client-oblivious notion. Steps (1) and (2) of our construction (perfectly) emulate $\mathsf{OblivElect}$ with the help of R, since they are functionally the same, and the communication pattern is of fixed topology (one fixed-size message from each client to R, and one fixed-size message from R to each accessing client). Generation of fake reads is done in a similar manner as CLT – though it is R, rather than the non-representatives that choose the locations of the fake reads to be made – and naturally the reads themselves are done in the same way. To multicast we cannot use an unencrypted sorting network, however, our step (4) allows only the requesting clients to retrieve their requests and is thus functionally the same as $\mathsf{OblivMulticast}$. Similarly we cannot use $\mathsf{OblivRoute}$, however, in our step (6) we again use a fixed topology mechanism so that a client receives the items that it must write to its partition. Flushing and writing back is exactly as in Subtree-OPRAM.

4.4 Extension to the Scheme

4.4.1 Dynamic Partitioning

In the OPRAM literature, each client gets allocated a portion of space they are in charge of read/writing to/from. These works normally discuss a scheme for when all clients are activated and say that it trivially extends. Here, we discuss some of the subtleties that arise in this model, and the best way to approach this. There are essentially two approaches; the data is shared once between all clients, or the data is reshared at *each* timestep only between the active clients.

In the former setting clients may have to access data even if they are not active themselves in that timestep. If only these clients request data, then this will reveal information about where data was accessed from. Thus all clients *must* request data each time step regardless of if they are active.

In the latter setting, only the activated clients will have to perform any actions in the given timestep. This is more desirable for the setting that we are considering where we have mutually distrustful clients, who may only be online for short periods of time. The disadvantage of this is that the clients who are active in a timestep must initially separate out the space evenly between them.

Incorporating a revocation mechanism into the procedure may be a very valuable property in practical applications. This is the simplest case of dynamic partitioning: in the event of revocation, the storage medium needs to be re-divided amongst the remaining clients. We assumed that $\frac{N}{m}$ is an integer, so even beginning to perform revocation in an elegant way appears challenging.

4.4.2 Write-back Content Hiding

Note that in our construction, for a given access, the client performing the read in step (5) learns the data to be written and the (new) position it is going to be written to. To hide this information from the reading client, R could encrypt the new data to be written under the one-time key of the writing client (the partition accessor of the new location ℓ'), and the reading client could then send this ciphertext to all other clients, such that only the writing client can decrypt. Doing so of course increases the communication between clients, adding m^2 messages per round (but removing the need for the TOR-style message transmission in step (6)). Removing this altogether and incorporating some sort of functional encryption mechanism (or possibly even just using homomorphic encryption) appears promising: for accesses that are reads, the writing partition accessor will always learn what is to be written but they could combine what they get from the reading partition accessor with an encryption of zero that is provided by R, such that they do not learn if the data they are writing is fresh or not. However doing this for writes means the 'encryption combiner' would have to completely disregard what was sent by the reading partition accessor: this functionality essentially has two encrypted inputs plus a flag indicating read or write, and the flag (which is hidden to the decryptor) essentially determines which input is decrypted.

5 Conclusions and Future Work

In this paper we have presented a new security notion for OPRAM schemes, namely client-obliviousness for clients. This notion captures the sensitivity of non-essential information that is passed among clients in OPRAM schemes, and may be desirable when the clients are not physically collocated and do not trust each other.

We have discussed the usefulness of our game-based exposition of this novel property, and our construction is built using well-established primitives with a minimally-trusted routing entity R. Our construction is a proof of concept, and consequently we leave further optimisations as open problems, including enabling dynamic assignment of storage partitions.

Another obvious avenue for improvement is to mimic the role of the router R using either a consortium of the clients – perhaps in combination with the storage server – or even all of the clients. Doing this in such a way that does not require (variants of) the sub-protocols of BCP and CLT appears to be very challenging, since this entity sees and sorts based on plaintext accesses. If the storage server could provide what is essentially a 'bulletin board' to which the clients could post the multicast ciphertexts in step (4), this would save one round of inter-client communication – formalising and securely realising (a mechanism similar to) this is also left as an open problem.

Another possible extension one could consider is to strengthen the model to 'malicious security', i.e. allowing the router R to arbitrarily deviate from the protocol. We think that this is possible using generic yet very inefficient techniques from the multi-party computation literature.

Acknowledgments

We thank James Alderman for initial discussions during the early stages of this project. We would also like to thank anonymous reviewers for valuable comments in improving earlier versions of this work.

Gareth T. Davies has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreement 802823. Christian Janson has been co-funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297. Parts of this work were conducted while Gareth T. Davies was employed by NTNU Trondheim, funded by the Norges Forskningsrådet (NFR) project 'Cryptographic Tools for Cloud Security'. Additionally, we would like to thank the NFR and this project for funding a visit of Christian Janson to NTNU Trondheim.

References

[AJCC16] James Alderman, Christian Janson, Carlos Cid, and Jason Crampton. Hybrid publicly verifiable computation. In Kazue Sako, editor, *Topics in Cryptology – CT-RSA 2016*, volume 9610 of *Lecture Notes in Computer Science*, pages 147–163, San Francisco, CA, USA, February 29 – March 4, 2016. Springer, Heidelberg, Germany. `doi:10.1007/978-3-319-29485-8_9`. (Cited on page 4.)

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *15th Annual ACM Symposium on Theory of Computing*, pages 1–9, Boston, MA, USA, April 25–27, 1983. ACM Press. `doi:10.1145/800061.808726`. (Cited on page 8.)

[BCP16] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A: 13th Theory of Cryptography Conference, Part II*, volume 9563 of *Lecture Notes in Computer Science*, pages 175–204, Tel Aviv, Israel, January 10–13, 2016. Springer, Heidelberg, Germany. `doi:10.1007/978-3-662-49099-0_7`. (Cited on pages 3, 4, 5, 8, and 28.)

[CCC$^+$16] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Cryptography for parallel RAM from indistinguishability obfuscation. In Madhu Sudan, editor, *ITCS 2016: 7th Conference on Innovations in Theoretical Computer Science*, pages 179–190, Cambridge, MA, USA, January 14–16, 2016. Association for Computing Machinery. `doi:10.1145/2840728.2840769`. (Cited on page 3.)

[CCS17] T.-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel RAM. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 567–597, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany. `doi:10.1007/978-3-319-70694-8_20`. (Cited on pages 3 and 4.)

[CGLS17] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 660–690, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany. `doi:10.1007/978-3-319-70694-8_23`. (Cited on pages 3 and 4.)

[CLP14] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 62–81, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany. doi:10.1007/978-3-662-45608-8_4. (Cited on page 3.)

[CLT16] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: Improved efficiency and generic constructions. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A: 13th Theory of Cryptography Conference, Part II*, volume 9563 of *Lecture Notes in Computer Science*, pages 205–234, Tel Aviv, Israel, January 10–13, 2016. Springer, Heidelberg, Germany. doi:10.1007/978-3-662-49099-0_8. (Cited on pages 3, 4, 7, 8, and 28.)

[CNS18] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018: 16th Theory of Cryptography Conference, Part II*, volume 11240 of *Lecture Notes in Computer Science*, pages 636–668, Panaji, India, November 11–14, 2018. Springer, Heidelberg, Germany. doi:10.1007/978-3-030-03810-6_23. (Cited on pages 3 and 4.)

[CP20] Weikeng Chen and Raluca Ada Popa. Metal: A metadata-hiding file-sharing system. In *ISOC Network and Distributed System Security Symposium – NDSS*, San Diego, CA, USA, February 23-26 2020. The Internet Society. doi:10.14722/ndss.2020.24095. (Cited on page 5.)

[CS16] Anrin Chakraborti and Radu Sion. POSTER: ConcurORAM: High-throughput parallel multi-client ORAM. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 1754–1756, Vienna, Austria, October 24–28, 2016. ACM Press. doi:10.1145/2976749.2989062. (Cited on page 5.)

[CS17] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part II*, volume 10678 of *Lecture Notes in Computer Science*, pages 72–107, Baltimore, MD, USA, November 12–15, 2017. Springer, Heidelberg, Germany. doi:10.1007/978-3-319-70503-3_3. (Cited on pages 3, 4, and 28.)

[CS19] Anrin Chakraborti and Radu Sion. ConcurORAM: High-throughput stateless parallel multi-client ORAM. In *ISOC Network and Distributed System Security Symposium – NDSS 2019*, San Diego, CA, USA, February 24-27, 2019. The Internet Society. (Cited on page 5.)

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In Matt Blaze, editor, *USENIX Security 2004: 13th USENIX Security Symposium*, pages 303–320, San Diego, CA, USA, August 9–13, 2004. USENIX Association. (Cited on page 14.)

[DS18] Jean Paul Degabriele and Martijn Stam. Untagging tor: A formal treatment of onion encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 259–293, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany. doi:10.1007/978-3-319-78372-7_9. (Cited on page 14.)

[FRW84] Faith E. Fich, Prabhakar Ragde, and Avi Wigderson. Relations between concurrent-write models of parallel computation. In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *3rd ACM Symposium Annual on Principles of Distributed Computing*, pages 179–189, Vancouver, BC, Canada, August 27–29, 1984. Association for Computing Machinery. doi:10.1145/800222.806745. (Cited on page 6.)

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *ACM Symposium on Theory of Computing*, pages 114–118. ACM, 1978. doi:10.1145/800133.804339. (Cited on page 6.)

[FWC$^+$12] Martin Franz, Peter Williams, Bogdan Carbunar, Stefan Katzenbeisser, Andreas Peter, Radu Sion, and Miroslava Sotáková. Oblivious outsourced storage with delegation. In George Danezis, editor, *FC 2011: 15th International Conference on Financial Cryptography and Data Security*, volume 7035 of *Lecture Notes in Computer Science*, pages 127–140, Gros Islet, St. Lucia, February 28 – March 4, 2012. Springer, Heidelberg, Germany. (Cited on pages 4 and 5.)

[GGH$^+$13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In Emiliano De Cristofaro and Matthew K. Wright, editors, *PETS 2013: 13th International Symposium on Privacy Enhancing Technologies*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18, Bloomington, IN, USA, July 10–12, 2013. Springer, Heidelberg, Germany. doi:10.1007/978-3-642-39077-7_1. (Cited on page 3.)

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011: 38th International Colloquium on Automata, Languages and Programming, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 576–587, Zurich, Switzerland, July 4–8, 2011. Springer, Heidelberg, Germany. doi:10.1007/978-3-642-22012-8_46. (Cited on page 4.)

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996. doi:10.1145/233551.233553. (Cited on pages 3, 4, and 5.)

[Gol78] Leslie M. Goldschlager. A unified approach to models of synchronous parallel machines. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *ACM Symposium on Theory of Computing*, pages 89–94. ACM, 1978. doi:10.1145/800133.804336. (Cited on page 6.)

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 182–194, New York City, NY, USA, May 25–27, 1987. ACM Press. doi:10.1145/28395.28416. (Cited on page 3.)

[KPK16] N. P. Karvelas, Andreas Peter, and Stefan Katzenbeisser. Blurry-ORAM: A multi-client oblivious storage architecture. Cryptology ePrint Archive, Report 2016/1077, 2016. http://eprint.iacr.org/2016/1077. (Cited on page 5.)

[Kuc82] Ludek Kucera. Parallel computation and conflicts in memory access. *Inf. Process. Lett.*, 14(2):93–96, 1982. doi:10.1016/0020-0190(82)90093-X. (Cited on page 6.)

[LPM$^+$13] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: ensuring private access to large-scale data in the data center. In *FAST*, pages 199–214. USENIX, 2013. (Cited on page 4.)

[MMRS15] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Privacy and access control for outsourced personal records. In *2015 IEEE Symposium on Security and Privacy*, pages 341–358, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press. doi:10.1109/SP.2015.28. (Cited on page 4.)

[MMRS17] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Maliciously secure multi-client ORAM. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17: 15th International Conference on Applied Cryptography and Network Security*, volume 10355 of *Lecture Notes in Computer Science*, pages 645–664, Kanazawa, Japan, July 10–12, 2017. Springer, Heidelberg, Germany. doi:10.1007/978-3-319-61204-1_32. (Cited on page 4.)

[NK16] Kartik Nayak and Jonathan Katz. An oblivious parallel RAM with $O(\log^2 N)$ parallel runtime blowup. Cryptology ePrint Archive, Report 2016/1141, 2016. http://eprint.iacr.org/2016/1141. (Cited on pages 3, 4, and 28.)

[NWI+15] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy*, pages 377–394, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press. doi:10.1109/SP.2015.30. (Cited on page 3.)

[PR10] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 502–519, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany. doi:10.1007/978-3-642-14623-7_27. (Cited on page 3.)

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214, Seoul, South Korea, December 4–8, 2011. Springer, Heidelberg, Germany. doi:10.1007/978-3-642-25385-0_11. (Cited on page 3.)

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *ISOC Network and Distributed System Security Symposium – NDSS 2012*, San Diego, CA, USA, February 5–8, 2012. The Internet Society. (Cited on page 7.)

[SvS+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 2013: 20th Conference on Computer and Communications Security*, pages 299–310, Berlin, Germany, November 4–8, 2013. ACM Press. doi:10.1145/2508859.2516660. (Cited on pages 4, 8, and 10.)

[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. doi:10.1109/SP.2016.20. (Cited on pages 3, 5, and 10.)

[Vis83] Uzi Vishkin. Implementation of simultaneous memory address access in models that forbid it. *J. Algorithms*, 4(1):45–50, 1983. doi:10.1016/0196-6774(83)90033-0. (Cited on page 6.)

[WST12] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: a parallel oblivious file system. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on*

Computer and Communications Security, pages 977–988, Raleigh, NC, USA, October 16–18, 2012. ACM Press. doi:10.1145/2382196.2382299. (Cited on page 4.)

A Additional Preliminaries

In this part of the appendix we provide any additional definitions which we require to construct our scheme, but which are not core to the paper.

A.1 Public-key Encryption Scheme

In the following we review the notion of a public-key encryption scheme.

Definition A.1. *A* public-key encryption scheme $\mathsf{PKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *consists of the following three algorithms:*

- $(pk, sk) \leftarrow_{\$} \mathsf{Gen}(1^\lambda)$: *this randomised* key generation *algorithm takes as input the common key the security parameter and returns a public and secret key pair* (pk, sk)*.*
- $C \leftarrow_{\$} \mathsf{Enc}(pk, m)$: *this randomised* encryption *algorithm takes as input the public key* pk *and a message* m *outputting a ciphertext* C*.*
- $d \leftarrow \mathsf{Dec}(sk, C)$: *this deterministic* decryption *algorithm takes as input the secret key* sk *and a ciphertext* C *outputting a string* d*. In case that decryption was successful then* d *corresponds to the message* m*, and otherwise* $\perp \notin \{0,1\}^*$*.*

Definition A.2. *A public-key encryption scheme* PKE *is said to be* correct*, if for all security parameters* λ*, all messages* $m \in \mathcal{M}_{\mathsf{PKE}}$*, and all key pairs* $(pk, sk) \leftarrow_{\$} \mathsf{Gen}(1^\lambda)$*, it holds that*

$$\text{Prob}\left[\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m\right] = 1.$$

Security for a public-key encryption scheme is defined in terms of indistinguishability of encrypted messages. The formulation corresponds to the well-known *real-or-random* paradigm and the formal details are displayed in Figure 3. Intuitively, the game proceeds as follows. After generating the key pair, the challenger chooses a bit b at random which parameterises the encryption oracle. Depending on this bit b, the oracle either encrypts the "real" message m which the adversary has given or a "random" message . After receiving back one ciphertext, the adversary's goal is to guess which message has been encrypted.

Definition A.3. *For the security game displayed in Figure 3, we define the advantage of the adversary as*

$$\mathbf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE},\ \mathcal{A}}(\lambda) = \text{Prob}\left[\mathbf{Exp}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE},\ \mathcal{A}}(\lambda) = 1\right] - \frac{1}{2}.$$

A public-key encryption scheme PKE *is said to be* IND-CPA secure *if for any adversary the above advantage is negligible in the security parameter.*

A.2 Symmetric-key Encryption Scheme

In the following we review the notion of a symmetric-key encryption scheme.

Definition A.4. *A* symmetric-key encryption scheme $\mathsf{SKE} = (\mathsf{KG}, \mathsf{E}, \mathsf{D})$ *consists of the following three algorithms:*

```
$\mathbf{Exp}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE},\ \mathcal{A}}(\lambda)$:
11 : $(pk, sk) \leftarrow_\$ \mathsf{Gen}(1^\lambda)$
12 : $b \leftarrow_\$ \{0,1\}$
13 : $b' \leftarrow_\$ \mathcal{A}^{\mathsf{Enc}_b(\cdot)}(1^\lambda, pk)$
14 : if $b' = b$
15 :     return 1
16 : else
17 :     return 0

$\mathsf{Enc}_b(m)$:
21 : $m_1 \leftarrow m$
22 : $m_0 \leftarrow_\$ \mathcal{M}_{\mathsf{PKE}}$
23 : $C \leftarrow_\$ \mathsf{Enc}(pk, m_b)$
24 : return $C$
```

Figure 3: Security notion of indistinguishability under chosen-plaintext attack (IND-CPA) for a public-key encryption scheme.

- $k \leftarrow_\$ \mathsf{KG}(1^\lambda)$: *this randomised* key generation *algorithm takes as input the common security parameter and returns a key k.*
- $ct \leftarrow_\$ \mathsf{E}(k, m)$: *this randomised* encryption *algorithm takes as input the key k and a message m outputting a ciphertext ct.*
- $d \leftarrow \mathsf{D}(k, ct)$: *this deterministic* decryption *algorithm takes as input the key k and a ciphertext ct outputting a string d. In case that decryption was successful then d corresponds to the message m, and otherwise $\perp$.*

Definition A.5. *A symmetric-key encryption scheme* SKE *is said to be* correct*, if for all security parameters λ, all messages $m \in \mathcal{M}_{\mathsf{SKE}}$, and all keys $k \leftarrow_\$ \mathsf{KG}(1^\lambda)$, it holds that*

$$\text{Prob}[\,\mathsf{D}(k, \mathsf{E}(k, m)) = m] = 1.$$

IND-CPA security for symmetric-key encryption can be given in a similar fashion to the game given in Figure 3. The only difference is that the key generation algorithm simply outputs a symmetric key k and hence the encryption oracle also receives this key. Note that the adversary is only given the security parameter. We need multi-user indistinguishability security for one-time symmetric encryption – this experiment is defined in Figure 4. The μ represents the multiple users, and $\mathcal{M}_{\mathsf{SKE}}[\cdot]$ is shorthand for elements of the message space that are the same size as the input value. In this game, the adversary has access to an oracle which takes as input an index and a message, and the adversary can query this oracle once for each index. The oracle will provide in return an encryption of either the input message (under the key associated with the index) or a random string, where this choice is made once for all oracle queries.

Definition A.6. *For the security game displayed in Figure 4, we define the* $\mu\mathsf{ind\text{-}xx}$ *advantage (for* $\mathsf{xx} \in \{\mathsf{ote}, \mathsf{mte}\}$*) of an adversary $\mathcal{A}$ as*

$$\mathbf{Adv}^{\mu\mathsf{ind\text{-}xx}}_{\mathsf{SKE},\ \mathcal{A}}(\lambda) = \text{Prob}\left[\mathbf{Exp}^{\mu\mathsf{ind\text{-}xx}}_{\mathsf{SKE},\ \mathcal{A}}(\lambda) = 1\right] - \frac{1}{2}.$$

A symmetric-key encryption scheme SKE *is said to be* $\mu\mathsf{ind\text{-}xx}$ *secure if for any adversary the above advantage is negligible in the security parameter.*

B Non-Interactivity

As mentioned in the discussion of the motivating scenario in the Introduction, we aim to focus on the case of OPRAM operating between some storage medium and many distinct client processors. In this

```
$\mathbf{Exp}^{\mu\text{ind-xx}}_{\mathsf{SKE},\ \mathcal{A}}(\lambda)$:
11 :  $b \leftarrow_\$ \{0,1\}$
12 :  for $i \in \{1,\dots,\mu\}$ do
13 :      $\tilde{k}_i \leftarrow_\$ \mathsf{KG}(1^\lambda)$
14 :      $\boxed{\text{flag}_i \leftarrow 0}$
15 :  $b' \leftarrow_\$ \mathcal{A}^{\mathsf{E}_b(\cdot,\cdot)}(1^\lambda)$
16 :  if $b' = b$
17 :      return $1$
18 :  else
19 :      return $0$

$\mathsf{E}_b(i,m)$:
21 :  $\boxed{\textbf{if } \text{flag}_i = 1 \textbf{ then}}$
22 :      $\boxed{\textbf{return } \perp}$
23 :  $\boxed{\text{flag}_i \leftarrow 1}$
24 :  $m_\$ \leftarrow_\$ \mathcal{M}_{\mathsf{SKE}}[|m|]$
25 :  if $b = 1$ then
26 :      return $\mathsf{E}(\tilde{k}_i, m)$
27 :  else
28 :      return $\mathsf{E}(\tilde{k}_i, m_\$)$
```

Figure 4: Security experiment for multi-user indistinguishability of one-time symmetric encryption (μind-ote) (boxed code included) and multi-time symmetric encryption μind-mte (boxed code excluded).

context, reducing the concrete communication complexity is paramount. This definition is stated as a possible design goal rather than a specific contribution of our work. We formally define what it means for an OPRAM compiler to enforce the *non-interactivity* of the clients. Note that BCP [BCP16] already hinted in Remark 1 that this property is desirable. Informally, the definition states that no two clients are required to communicate with each other during the execution of the program.

Definition B.1 (Non-Interactive OPRAM)**.** *Let* $\mathbf{y}$ *be a sequence of parallel accesses.* $\mathcal{O}$ *is* non-interactive *if all messages in* $\mathcal{O}(\mathbf{y})$ *are either sent or received by the server* $\mathcal{S}(N', B')$*.*

It is fairly straightforward to see that the constructions of OPRAM in the literature [BCP16, CLT16, CS17, NK16] require interaction between the clients. However, they can (trivially) be made non-interactive using the following proposition.

Proposition B.2. *Any OPRAM compiler* $\mathcal{O}$ *can be converted into a non-interactive OPRAM compiler* $\mathcal{O}'$ *without changing the time complexity.*

Proof. Any message sent from one client to another can be replaced by a pair of messages, one from the sender to the server and then from the server to the recipient. This can at most double the number of communication rounds and thus does not change the overall time complexity. □

In the presence of the routing entity R, this definition of non-interactivity essentially means that the instructions that are created by $\mathcal{O}$, including those sent from one client to R and vice versa, are sent via the server. This in itself may be undesirable depending on geographical proximity and system topology, and we stress here that the utility of this definition and construction technique is very dependent on the application scenario in question.