OBLIVP2P: An Oblivious Peer-to-Peer Content Sharing System

Yaoqi Jia[1*]   Tarik Moataz[2*]   Shruti Tople[1*]   Prateek Saxena[1]

[1]*National University of Singapore*
{*jiayaoqi, shruti90, prateeks*}*@comp.nus.edu.sg*
[2]*Colorado State University and Telecom Bretagne*
tarik.moataz@colostate.edu

Abstract

Peer-to-peer (P2P) systems are predominantly used to distribute trust, increase availability and improve performance. A number of content-sharing P2P systems, for file-sharing applications (e.g., BitTorrent and Storj) and more recent peer-assisted CDNs (e.g., Akamai Netsession), are finding wide deployment. A major security concern with content-sharing P2P systems is the risk of long-term *traffic analysis* — a widely accepted challenge with few known solutions.

In this paper, we propose a new approach to protecting against persistent, global traffic analysis in P2P content-sharing systems. Our approach advocates for hiding data access patterns, making P2P systems *oblivious*. We propose OBLIVP2P— a construction for a scalable distributed ORAM protocol, usable in a real P2P setting. Our protocol achieves the following results. First, we show that our construction retains the (linear) scalability of the original P2P network w.r.t the number of peers. Second, our experiments simulating about 16,384 peers on 15 Deterlab nodes can process up to 7 requests of 512KB each per second, suggesting usability in moderately latency-sensitive applications as-is. The bottlenecks remaining are purely computational (not bandwidth). Third, our experiments confirm that in our construction, no centralized infrastructure is a bottleneck — essentially, ensuring that the network and computational overheads can be completely offloaded to the P2P network. Finally, our construction is highly parallelizable, which implies that remaining computational bottlenecks can be drastically reduced if OBLIVP2P is deployed on a network with many real machines.

1 Introduction

Content sharing peer-to-peer (P2P) systems, especially P2P file-sharing applications such as BitTorrent [1], Storj [2] and Freenet [3] are popular among users for sharing files on the Internet. More recently, peer-assisted CDNs such as Akamai Netsession [4] and Squirrel [5] are gaining wide adoption to offload web CDN traffic to clients. The convenient access to various resources attract millions of users to join P2P networks, e.g., BitTorrent has over 150 million active users per month [6] and its file-sharing service contributes 3.35% of all worldwide bandwidth [7]. However, the majority of such P2P applications are susceptible to long-term traffic analysis through global monitoring; especially, analyzing the *pattern* of communication between a sender and a receiver to infer information about the users. For example, many copyright enforcement organizations such as IFPI, RIAA, MPAA, government agencies like NSA and ISP's are reported to globally monitor BitTorrent traffic to identify illegal actors. Monitoring of BitTorrent traffic has shown to reveal the data requested and sent by the peers in the network [8–10]. Unfortunately, while detecting copyright infringements is useful, the same global monitoring is applicable to *any* user of the P2P network, and can therefore collect benign users' data. Thus, users of such P2P systems are at a risk of leaking private information such as the resources they upload or download.

To hide their online traces, users today employ anonymous networks as a solution to conceal their digital identities or data access habits. Currently, anonymous networks include Mix networks [11–13], and Onion routing/Tor-based systems [14–17], as well as other P2P anonymity systems [18–23]. Such systems allow the user to be *anonymous*, so that the user is unidentifiable within a set of users [24].

Although above solutions provide an anonymity guarantee, they are vulnerable to long-term traffic pattern analysis attacks, which is an important threat for P2P systems like BitTorrent [25–30]. Researchers have demonstrated attacks targeting BitTorrent users on top of Tor that reveal information related to the resources uploaded or downloaded [31, 32]. Such attacks raise the question - *is anonymizing users the right defense against traffic*

---

*Lead authors are alphabetically ordered.

pattern analysis in P2P content sharing systems?

In this paper, we investigate a new approach to solve the problem of persistent analysis of data communication patterns. We advocate that data / resource access pattern hiding is an important and necessary step to thwart leakage of users data in P2P systems. To this end, we present a first candidate solution, OBLIVP2P— an oblivious protocol for peer-to-peer content sharing systems. Hiding data access patterns or making them *oblivious* unlinks user's identity from her online traces, thereby defending against long-term traffic monitoring.

1.1 Approach

For hiding data access patterns between a trusted CPU and an untrusted memory, Goldreich and Ostrovsky proposed the concept of an Oblivious RAM (ORAM) [33]. We envision providing similar obliviousness guarantees in P2P systems, and therefore select ORAM as a starting point for our solution. To the best of our knowledge, OBLIVP2P is the first work that adapts ORAM to accesses in a P2P setting. However, directly employing ORAM to hide access patterns in a P2P system is challenging. We outline two key challenges in designing an oblivious and a scalable P2P protocol using ORAM.

Obliviousness. The first challenge arises due to the difference in the setting of a standard ORAM as compared to a P2P content sharing system. Classical ORAM solutions consists of a single client which securely accesses an untrusted storage (server), wherein the client is eventually the owner and the only user of the data in the memory. In contrast, P2P systems consist of a set of trusted trackers managing the network, and multiple data owners (peers) in the network. Each peer acts both as a client as well as a server in the network i.e., a peer can either request for a data or respond to other peer's request with the data stored on its machine. Hence, adversarial peers present in the network can see the plaintext and learn the data requested by other peers, a threat that does not exists in the traditional ORAM model where only encrypted data is seen by the servers.

Scalability. The second challenge lies in seeking an oblivious P2P system that 1) the throughput scales linearly with the number of peers in the network, 2) has no centralized bottleneck and 3) can be parallelized with an overall acceptable throughput. In standard ORAM solutions, the (possibly distributed) server is responsible for serving all the data access requests from a client one-by-one. In contrast, P2P systems operate on a large-scale with multiple peers (clients) requesting resources from each other simultaneously without overloading a particular entity. To retain scalability of P2P systems, it is necessary to ensure that requests can be served by distributing the communication and computation overhead.

Solution Overview. We start with a toy construction (OBLIVP2P-0) which directly adapts ORAM to a P2P setting, and then present our main contribution which is a more efficient solution (OBLIVP2P-1).

Centralized Protocol (OBLIVP2P-0): Our centralized protocol or OBLIVP2P-0, is a direct adaptation of ORAM in a P2P system. The peers in the network behave both like distributed storage servers as well as clients. They request a centralized, trusted tracker to access a particular resource. The tracker performs all the ORAM operations to fetch the resource from the network and returns it to the requesting peer. However, this variant of OBLIVP2P protocol has limited scalability as it assigns heavy computation to the tracker, making it a bottleneck.

Distributed Protocol (OBLIVP2P-1): As our main contribution, we present OBLIVP2P-1 which provides both obliviousness and scalability properties in a tracker-based P2P system. To attain scalability, the key idea is to avoid any single entity (say the tracker) as a bottleneck. This requires distributing all the ORAM operations for fetching and sharing of resources among the peers in the network, while still maintaining obliviousness guarantees. To realize such a distributed protocol, our main building block, which we call *Oblivious Selection* (OblivSel), is a novel combination of private information retrieval with recent advances in ORAM. Oblivious Selection gives us a scalable way to securely distribute the load of the tracker. Our construction is proven secure in the honest-but-curious adversary model. Constructions and proofs for arbitrarily malicious fraction of peers is slated for future work.

1.2 System and Results

We provide a prototype implementation of both OBLIVP2P-0 and OBLIVP2P-1 protocols in Python. Our source code is available online [34]. We experimentally evaluate our implementation on DeterLab testbed with 15 servers simulating up to 2^{14} peers in the network. Our experiments demonstrate that OBLIVP2P-0 is limited in scalability with the tracker as the main bottleneck. The throughput for OBLIVP2P-1, in contrast, scales linearly with increase in the number of peers in the network. It attains an overall throughput of 3.19 MBps for a network of 2^{14} peers that corresponds to 7 requests per second for a block size of 512 KB. By design, OBLIVP2P-1 is embarrassingly parallelizable over the computational capacity available in a real P2P network. Further, our protocol exhibits no bottleneck on a single entity in experiment, thereby confirming that the network and the computational overhead can be completely offloaded to

the P2P network.

Contributions. We summarize our contributions below:

- **Problem Formulation.** We formulate the problem of making data access pattern oblivious in P2P systems. This is a necessary and important step in building defenses against long-term traffic analysis.
- **New Protocols.** We propose OBLIVP2P— a first candidate for an oblivious peer-to-peer protocol in content sharing systems. Our main building block is a primitive which we refer to as oblivious selection that makes a novel use of recent advances in Oblivious RAM combined with private information retrieval techniques.
- **System Implementation & Evaluation.** Our prototype implementation is available online [34]. We experimentally evaluate our protocol to measure the overall throughput of our system, latency for accessing resources and the impact of optimizations on the system throughput.

2 Problem

Many P2P applications are not designed with security in mind, making them vulnerable to traffic pattern analysis. We consider BitTorrent as our primary case study. However, the problem we discuss is broadly applicable to other P2P file sharing systems like Gnutella [35], Freenet [3] and Storj [2] or peer-assisted CDNs such as Akamai Netsession [4], Squirrel [5] and APAC [36].

2.1 BitTorrent: A P2P Protocol

The BitTorrent protocol allows sharing of large files between users by dividing it into blocks and distributing it among the peers. It has a dynamic network, made up of a number of nodes that join the network and volunteer themselves as peers. Each peer holds data blocks in its local storage and acts both as a client / requester and server / sender simultaneously. There exists a tracker that tracks which peers are downloading / uploading which file and saves the state of the network. It keeps information regarding the position or the IP addresses of peers holding each resource but does not store any real data blocks. A peer requests the tracker for a particular resource and the tracker responds with a set of IP addresses of peers holding the resource. The requester then communicates with these IP addresses to download the blocks of the desired resource. The peers interact with each other using a P2P protocol[1]. The requester concatenates all the blocks received to construct the entire resource.

---

[1] We want to emphasize that there are other models of P2P networks without tracker based on DHT that we are not addressing in this work.

2.2 Threat Model

In our threat model, we consider the tracker as a trusted party and peers as passive honest-but-curious adversaries i.e., the peers are expected to correctly follow the protocol without deviating from it to learn any extra information. In P2P systems including CDNs (content delivery networks) and BitTorrent, passive monitoring is already a significant threat on its own. We consider the following two types of adversaries:

Global Passive Adversary. Since BitTorrent traffic is public, there exist tools like Global BitTorrent Monitor [37] or BitStalker [38] that support accurate and efficient monitoring of BitTorrent. Previous research has shown that any BitTorrent user can be logged within a span of 3 hours, revealing his digital identity and the content downloaded [39]. Further, the adversary can log the communication history of the network traffic to perform offline analysis at a later stage. Hence, we consider it rational to assume the presence of a global adversary with the capability to observe long term traffic in the network.

Passive Colluding Peers. Some of the peers in the P2P network can be controlled by the global adversary. They can further collude to exchange data with other adversarial peers in the system. While colluding these "sybil" peers can share information such as observed / served requests and the contents stored at their local storage. Their goal is to collectively glean information about other peers in the network. A formal definition of passive colluding peers is as follows:

Definition 2.1. *(Passive Colluding peers) We say that a peer P_i passively colludes with peer P_j if both peers share their views without any modification, where a view consists of: a transcript of the sequence of all accesses made by P_i, a partial or total copy of peer's private storage, and a transcript of the access pattern induced by the sequence of accesses. We denote by $\mathscr{C}(P_i)$ the set of colluding peers with P_i.*

Note that from the above definition, we have a symmetric relation such that if $P_i \in \mathscr{C}(P_j)$ for $i \neq j$, then $P_j \in \mathscr{C}(P_i)$. It follows that if $P_i \notin \mathscr{C}(P_j)$, then $\mathscr{C}(P_i)$ and $\mathscr{C}(P_j)$ are disjoint.

Our protocol tolerates a fraction of c adversarial peers in the network such that $c \in O(N^\varepsilon)$, where N is the total number of peers in the network and $\varepsilon < 1$. Although the P2P network undergoes churn, we assume the fraction of adversarial peers c remains within the asymptotic bounds of $O(N^\varepsilon)$. Our choice of the upper bound for c ensures an exponentially small advantage to the attacker; for an application that can tolerate higher attacker's advantage, a larger malicious fraction can be allowed.

2.3 Insufficiency of Existing Approaches

Existing techniques propose anonymizing users to prevent traffic pattern analysis attacks. However, these solutions are not sufficient to protect against a global adversary with long term access to communication patterns.

Unlinkability Techniques (e.g. Mixnet). Existing anonymity approaches "unlink" the sender from the receiver (see survey [40]). Chaum proposed the first anonymous network called mix network [11], which shuffles messages from multiple senders using a chain of proxy servers and sends them to the receiver. Another recent system called Riposte guarantees traffic analysis resistance by unlinking a sender from its message [41]. However, all these systems are prone to attack if an adversary can observe multiple request rounds in the network.

For example, consider that *Alice* continuously communicates with *Bob* using a mixnet service. A global adversary observes this communication for a couple of rounds, and records the recipient set in each round. Let the senders' set consists of $S_1 = \{Alice, a, b, c\}$ and $S_2 = \{a', b', Alice, c'\}$, and the recipients' set consists of $R_1 = \{x, y, z, Bob\}$ and $R_2 = \{x', y', Bob, z'\}$ for rounds 1 and 2 respectively. The attacker can then infer the link between sender and receiver by intersecting $S_1 \cap S_2 = \{Alice\}$ and $R_1 \cap R_2 = \{Bob\}$. The attacker learns that Alice is communicating with Bob, and thus breaks the unlinkability. This attack is called the *intersection, hitting set* or *statistical disclosure attack* [25, 26]. Overall, one time unlinkability is not a sufficient level of defense when the adversary can observe traffic for arbitrary rounds.

Path Non-Correlation (e.g. Onion routing). Another approach for guaranteeing anonymity is to route the message from a path such that the sender and the receiver cannot be correlated by a subset of passive adversarial nodes. Onion-routing based systems like Tor enable anonymous communication by using a sequence of relays as intermediate nodes (called circuit) to forward traffic [15, 42]. However, Tor cannot provide sender anonymity when the attacker can see both the ends of the communication, or if a global adversary observes the entire network. Hence, if an attacker controls the entry and the exit peer then the adversarial peers can determine the recipient identity with which the initiator peer is communicating [27–30]. This is a well-known attack called the *end-to-end correlation attack* or *traffic confirmation attack* [43, 44].

2.4 Problem Statement

Our goal is to design a P2P protocol that prevents linking a user to a requested resource using traffic pattern analysis. Section 2.3 shows how previous anonymity based solutions are susceptible to attacks in our threat model. In this work, we address this problem from a new viewpoint, by making the communication pattern oblivious in the network. We advocate that hiding data / resource access pattern is a necessary and important step in designing traffic pattern analysis resistant P2P systems.

In a P2P system such as BitTorrent, a user accesses a particular resource by either downloading (Fetch) or uploading (Upload) it to the network. We propose to build an oblivious P2P content sharing protocol (OBLIVP2P) that hides the data access patterns of users in the network. We formally define an Oblivious P2P protocol as follows:

Definition 2.2. *(Oblivious P2P): Let* $(P_1, \cdots, P_n)$ *and* $\mathscr{T}$ *be respectively a set of n peers and a tracker in a* P2P *system. We denote by* $\overrightarrow{x_i} = (x_{i,1}, \cdots, x_{i,M})$ *a sequence of M accesses made by peer* P_i *such that* $x_{i,j} = (\mathsf{op}_{i,j}, \mathsf{fid}_{i,j}, \mathsf{file}_{i,j})$ *where* $\mathsf{op}_{i,j} = \{\mathsf{Upload}, \mathsf{Fetch}\}$, $\mathsf{fid}_{i,j}$ *is the filename being accessed, and* $\mathsf{file}_{i,j}$ *is the set of blocks being written in the network if* $\mathsf{op}_{i,j} = \mathsf{Upload}$.

We denote by $\mathscr{A}(\overrightarrow{x_i})$ *the access pattern induced by the access sequence* $\overrightarrow{x_i}$ *of peer* P_i. *The access pattern is composed of the memory arrays of all peers accessed while running the sequence* $\overrightarrow{x_i}$. *We say that a* P2P *is* oblivious *if for any two equal-length access sequences* $\overrightarrow{x_i}$ *and* $\overrightarrow{x_j}$ *by two peers* P_i *and* P_j *such that*

- $P_j \notin \mathscr{C}(P_i)$
- $\forall k \in [M]$: $x_{i,k} = \mathsf{Fetch} \Leftrightarrow x_{j,k} = \mathsf{Fetch} \wedge x_{i,k} = \mathsf{Upload} \Leftrightarrow x_{j,k} = \mathsf{Upload}$
- $\forall k \in [M], |\mathsf{file}_{i,k}| = |\mathsf{file}_{j,k}|$

are indistinguishable for all probabilistic poly-time adversaries except for $\mathscr{C}(P_i)$, $\mathscr{C}(P_j)$, *and tracker* $\mathscr{T}$.

Scope. OBLIVP2P guarantees resistance against persistent communication traffic analysis i.e., observing the path of communication and thereby linking a sender to a particular resource. OBLIVP2P does *not* prevent against:

a) *Active Tampering*: An adversarial peer can tamper, alter and deviate from the protocol to learn extra information. Admittedly, this can have an impact on obliviousness, correctness and availability of the network.

b) *Side Channels*: An adversary can monitor any peer in the system to infer its usage's habits via side channels: the number of requests, time of activity, and total number of uploads. In addition, an adversary can always infer the total file size that any peer is downloading or uploading to the P2P network. Literature shows that some attacks such as website fingerprinting can be based on the length of file requested by peers [45].

c) *Orthogonal Attacks*: Other attacks in P2P file sharing

systems consist of threats such as poisoning of files by uploading corrupted, fake or misleading content [46] or denial of service attacks [47]. However, these attacks do not focus on learning private information about the peers and hence are orthogonal to our problem.

Admittedly, our assumption about honest-but-curious is less than ideal and simplifies analysis. We hope that our construction spurs future work on tackling the active or arbitrary malicious adversaries. Emerging trusted computing primitives (e.g., Intel SGX [48]) or cryptographic measures [49] are promising directions to investigate. Lastly, OBLIVP2P should not be confused with traditional anonymous systems where a user is anonymous among a set of users. OBLIVP2P does not guarantee sender or receiver anonymity, but hides data access patterns of the users.

3 Our Approach

As a defense against traffic pattern analysis, we guarantee oblivious access patterns in P2P systems. We consider Oblivious RAM as a starting point.

3.1 Background: Tree-Based ORAM

Oblivious RAM, introduced by Goldreich and Ostrovsky [33], is a cryptographic primitive that prevents an adversary from inferring any information via the memory access pattern. Tree-based ORAM introduced by Shi et al. [50] offers a poly-logarithmic overhead which is further reduced due to improvements suggested in the follow up works [51–56]. In particular, we use Ring ORAM, [52], one of the latest improvements for tree-based ORAM in our protocol. In Ring ORAM, to store N data blocks, the memory is organized in a (roughly) $\log N$-height full binary tree, where each node contains z real blocks and s dummy blocks. Whenever a block is accessed in the tree, it is associated to a new randomly selected leaf identifier called, tag. The client stores this association in a position map PosMap along with a private storage (stash). To read and write to the untrusted memory, the client performs an Access followed by an Evict operation described *at a high level* as follows:

- Access(adr): Given address *adr*, the client fetches the leaf identifier tag from PosMap. Given tag, the client downloads one block per every node in the path $\mathscr{P}(\text{tag})$ that starts from the root and ends with the leaf tag. The client decrypts the retrieved blocks, and retrieves the desired block. This block is appended to the stash.
- Evict(A, v): After A accesses, the client selects a path $\mathscr{P}(v)$ based on a deterministic reverse lexicographic order, downloads the path, decrypts it and appends it to the stash. The client runs the
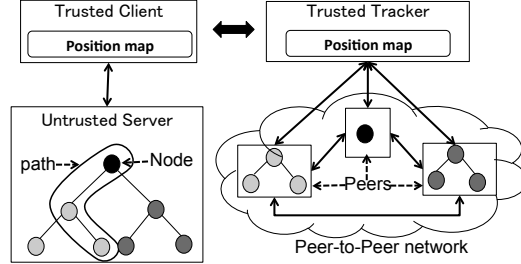

Figure 1: Mapping of a client / server ORAM model to a P2P system

least common ancestor algorithm to sort the blocks as in [51]. Finally, the client freshly encrypts the blocks and writes them back to the nodes in the path.

The stash is upper bounded by $O(\log N)$. The overall bandwidth may reach $\simeq 2.5 \log N$, for N blocks stored. In Ring ORAM, eviction happens periodically after a controllable parameter $A = 2z$ accesses where z is the number of blocks in each bucket [52].

3.2 Mapping an ORAM to a P2P setting

We start from a traditional ORAM in a client / server model where the client is trusted and the server is not, and simulate it on a tracker / peers setting. In particular, we consider that the server's memory is organized in a tree structure, and we delegate every node in the tree to a peer. That is, a full binary tree of N leaves is now distributed among $N_p = 2N - 1$ peers (refer to Figure 1). In practice, many nodes can be delegated to many peers based on the storage capacity of each peer.

Contrary to the client / server setting where the client is the only one who can fetch, modify or add a block, in P2P, the peers can also request and add new blocks. In addition, the peers are volatile, i.e, many peers can join or leave the network. Moreover, from a security perspective, the network peers do not trust each other, and an adversarial peer can always be interested in finding out the block being retrieved by other peers. To avoid this, the tracker instructs the peers in a P2P system to save encrypted blocks in their local memory (different from the conventional BitTorrent model). Our construction ensures that the peer neither has the keys necessary to decrypt its storage nor can it collude with other adversarial peers to recover it. In this setting, we first present a strawman approach that guarantees our security goal but is restricted in terms of scalability.

3.3 OBLIVP2P-0 : Centralized Protocol

Almost all ORAM constructions are in a client / server setting and not designed for a P2P setting. A simple approach is to map the role of the trusted client in an

ORAM setting (refer to Figure 1) to the trusted tracker in a P2P system. The client in ORAM is simulated by the trusted tracker (storing the position map, private keys and the stash) and the server by the untrusted peers (storing the encrypted blocks). With such a mapping from an ORAM model to a P2P setting, a peer (initiator) can request for a resource to the tracker. To access a particular resource, the tracker fetches the blocks from a path in the tree and decrypts them to get the desired block. It then returns the requested resource to the initiator peer. This simple *plug-&-play* construction satisfies all our P2P security requirements.

In OBLIVP2P-0, the trusted tracker behaves as the client in traditional ORAM model. Whenever a peer requests a block, the tracker performs all the ORAM access work, and then sends the plaintext block to the initiator. The tracker downloads the path composed of a logarithmic number of nodes, writes back the path with a fresh re-encryption before routing the block to the initiator. As long as the tracker is trusted, this ensures the obliviousness property of peers' accesses, as stated by definition 2.2.

Upload algorithm. To upload a file, the peer divides it into data blocks and sends the blocks to the tracker. The tracker appends the block to the stash stored locally while generating new random tags. The tracker updates accordingly TagMap, and FileMap (refer to Table 1).

Fetch algorithm. To fetch a file, the peer sends the file identifier, as an instance a filename, to the tracker. The tracker fetches from the FileMap and TagMap the corresponding blocks and sends requests to the corresponding peers to retrieve the blocks, following the Ring ORAM Access protocol. For every retrieved block, the tracker sends the plaintext block to the requesting peer.

Sync algorithm. The synchronization happens after every $A \simeq 2z$ accesses [52] (e.g., nearly 8 accesses) at which point the tracker evicts the stash.

Tracker as Bottleneck. In OBLIVP2P-0, the tracker has to transmit / encrypt a logarithmic number of blocks on every access. The tracker requires a bandwidth of $O(\log N \cdot B)$ where B is the block size and the computation cost of $O(\log N \cdot E)$ where E is time for encrypting / decrypting a block. Moreover, our evaluation in Section 5 shows that the eviction step is network-intensive. In a P2P setting with large number of accesses per second, the tracker creates a bottleneck in the network.

3.4 OBLIVP2P-0 Analysis

Our analysis follow from Ring ORAM construction. To access a block the tracker has to transmit $\sim 2.5 \log N \cdot B$ bits per access. During a block access or eviction, any peer at any time transmits $O(B)$ bits. The tracker's main

computational time consists of decrypting and encrypting the stash. Since the stash has a size of $O(\log N)$ blocks, the tracker does $O(\log N)$ blocks encryption/decryption. In terms of storage, every peer has $(z + s)$ blocks to store, where z is number of real blocks and s is a parameter for dummy blocks. From a security perspective, it is clear that if there are two sequences verifying the constraints of Definition 2.2, a malicious peer monitoring their access pattern cannot infer the retrieved blocks, since after every access the block is assigned to a random path in the simulated ORAM.

4 OBLIVP2P-1: Distributed Protocol

In this section, we describe our main contribution, OBLIVP2P-1 protocol that provides both security and scalability properties. In designing such a protocol, our main goal is to avoid any bottleneck on the tracker i.e., none of the real blocks should route through the tracker for performing an access or evict operations of ORAM. We outline the challenges in achieving this property while still retaining the obliviousness in the network.

4.1 Challenges

First Attempt. A first attempt to reduce tracker's overhead is to modify OBLIVP2P-0 such that the heavy computation of fetching the path of a tree and decrypting the correct block is offloaded to the initiator peer. On getting a resource request from a peer, the tracker simply sends information to the peer that includes the path of the tree to fetch, the exact position of the requested block and the key to decrypt it. However, unlike standard ORAM, the peer in our model is not trusted. Giving away the exact position of the block to the initiator peer leaks additional information about the requested resource in our model, as we explain next.

Recall that in a tree-based ORAM, blocks are distributed in the tree such that the recently accessed blocks remain in the top of the tree. In fact, after every eviction the blocks in the path are pushed down as far as possible from the root of the tree. As an instance, after N deterministic evictions, all blocks that were never accessed are (very likely) in the leaves. Conversely, consider that an adversarial peer makes two back-to-back accesses. In the first access, it retrieves a block from the top of the tree while in the second access it retrieves a block from a leaf. The adversarial peer (initiator) learns that the first block is a popular resource and is requested before by other peers while the second resource is a less frequently requested resource. This is a well known issue in tree-based ORAM, and is recently formulated as the *block history* problem [57]. Disclosing the block position, while hiding the scheme obliviousness requires to

address the block history challenge in ORAM. Unfortunately, an ORAM hides the block history only if the communication spent to access a block dominates the number of blocks stored in the entire ORAM. This would be asymptotically equivalent to downloading the entire ORAM tree from all the peers. We refer readers to [57] for more details.

Second Attempt. Our second attempt is a protocol that selects a block while *hiding* the block position from the adversary i.e., to hide which node on the path holds the requested block. Note that in a tree-based ORAM, disclosing the path does not break obliviousness, but leaking which node on the path holds the requested block is a source of leakage. One trick is to introduce a circuit, a set of peers from the P2P network, that will simulate the operations of a mixnet. That is, the peers holding the path of the tree send their content to the first peer in the circuit, who then applies a random permutation, adds a new encryption layer, and sends the permuted path to the second peer and so on. The tracker, who knows all the permutations, can send the final block position (unlinked from original position) to the initiator, along with the keys to decrypt the block. The mixing guarantees that the initiator does not learn the actual position of the block. We note that mixing used here is for only one accessed "path", which is already randomized by ORAM. Hence, it is not susceptible to intersection attack discussed in Section 2.3. Finally, the initiator then peels off all layers of the desired block to output the plaintext block.

However, there is an important caveat remaining in using this method. Note that the initiator has the keys to peel off all the layers of encryption and hence it has access to the same encrypted block fetched from the path in the tree. Thus, it can determine which peer's encrypted block was finally selected as the output of the mixnet. Hence, delegating the keys to the initiator boils down to giving her the block position. One might think of eliminating this issue by routing the block through the tracker to peel off all layers, but this will just make the tracker again a bottleneck.

So far, our attempts have shown limitations, but pointed out that there is a need to formally define the desired property. Considering a tracker, the initiator, and the peers holding the path, we seek a primitive that given a set of encrypted blocks, the initiator can get the desired plaintext block, while no entity can infer the block position but the tracker. We refer to this primitive as *Oblivious Selection* (OblivSel) and describe it next.

4.2 Oblivious Selection

4.2.1 Definitions

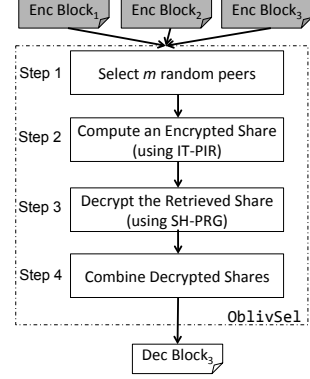We define OblivSel and its properties as follows:

Figure 2: Oblivious Selection protocol using IT-PIR and Seed Homomorphic PRG as base primitives

Definition 4.1. *(Oblivious Selection).* OblivSel *is a tuple of two probabilistic algorithms* $(\mathsf{Gen}, \mathsf{Select})$ *such that:*

- $(\vec{\sigma}, \vec{r}) \leftarrow \mathsf{Gen}(\mathsf{k}, \mathsf{pos})$*: a probabilistic algorithm run by the tracker, takes as input a key* k *and the block position* pos*, picks uniformly at random* m *peers* $(P_1, \cdots, P_m)$*, and outputs* $(\vec{\sigma}, \vec{r})$ *where* $\vec{\sigma} = \{\sigma_1, \cdots, \sigma_m\}$ *and* $\vec{r} = \{r_1, \cdots, r_m\}$ *such that* (σ_i, r_i) *is given to the ith peer* P_i*.*

- $\Delta \leftarrow \mathsf{Select}(\vec{\sigma}, \vec{r}, \mathsf{Enc}(\mathsf{k}_1, \mathsf{block}_1), \cdots, \mathsf{Enc}(\mathsf{k}_\mathsf{L}, \mathsf{block}_\mathsf{L}))$*: a probabilistic algorithm run by* m *peers, takes as input* $\vec{\sigma}$*,* $\vec{r}$*, and a set of encrypted blocks* $\mathsf{Enc}(\mathsf{k}_i, \mathsf{block}_i)$*, for* $i \in [\mathsf{L}]$*, and outputs the value* Δ*.*

Definition 4.2. OblivSel*, is **correct**, if*

$$\Pr[\forall \; \mathsf{pos} \in [\mathsf{L}], \mathsf{k} \in \{0,1\}^\lambda, (\vec{\sigma}, \vec{r}) \leftarrow \mathsf{Gen}(\mathsf{k}, \mathsf{pos});$$
$$\Delta \leftarrow \mathsf{Select}(\vec{\sigma}, \vec{r}, \mathsf{Enc}(\mathsf{k}_1, \mathsf{block}_1), \cdots, \mathsf{Enc}(\mathsf{k}_\mathsf{L}, \mathsf{block}_\mathsf{L}));$$
$$\Delta = \mathsf{Dec}(\mathsf{k}, \mathsf{Enc}(\mathsf{k}_{\mathsf{pos}}, \mathsf{block}_{\mathsf{pos}}))] = 1$$

For instance, if $(\mathsf{Enc}, \mathsf{Dec})$ is a private key encryption, OblivSel returns a decrypted block when the key given as input to the Gen function is the same as the private key of the block i.e., $\Delta = \mathsf{block}_{\mathsf{pos}}$ if $\mathsf{k} = \mathsf{k}_{\mathsf{pos}}$.

Definition 4.3. *(Position Hiding.) We say that* OblivSel *is a* position hiding *protocol if for all probabilistic polynomial time global adversaries, including the initiator and the* m *peers, guess the position of the block* pos *with a negligible advantage in the implicit security parameter.*

4.2.2 OblivSel **Overview**

The intuition for constructing OblivSel stems from the fact that the tracker cannot give the position or private key of the desired block to the peers in the network.

To privately select a block from the path without leaking its position, we propose to use an existing cryptographic primitive, called information-theoretical private

Structure	Mapping	Purpose
FileMap	file id fid <u>to</u> block addresses $\{\mathsf{adr}_i\}_{i \in [\frac{f}{B}]}$	Blocks identification
TagMap	block address adr <u>to</u> tag $\xleftarrow{\$} [N_B]$	Path identification
NetMap	peer id pid <u>to</u> network info $(\mathsf{IP}, \mathsf{port}) \in \{0,1\}^{128+16}$	Network representation
PosMap	block address adr <u>to</u> path and bucket position $\mathsf{pos} \in [N_p] \times [L \cdot z + |\mathsf{stash}|]$	Block exact localization
KeyMap	block address adr <u>to</u> key value $\mathsf{k} \xleftarrow{\$} \mathbb{Z}_q$	Input of key block generation
StashList	peers' identifiers $\{\mathsf{pid}_i\}_{i \in [|\mathsf{stash}|]}$	Stash localization

Table 1: Various meta-information contained in the state s, for OBLIVP2P-0 and OBLIVP2P-1. B is the block size in bits, N_P the number of peers, N_B number of blocks, L the path length, and z the bucket size.

information retrieval (IT-PIR) [58]. IT-PIR requires a *linear* computation proportional to the data size that makes it expensive to use for real time settings. However, note that in our setting, we want to obliviously select a block from a logarithmic number of blocks (i.e., a path of the tree). Thus, applying IT-PIR over tree-based ORAM comes with significant computational improvement, hence making it practical to use in our protocol. The high level idea is to apply IT-PIR primitive only on one path since the obliviousness is already guaranteed by the underlined tree-based ORAM construction.

Figure 2 shows the steps involved in our OblivSel primitive. As a first step, the tracker randomly samples m peers from the network. For a bounded number of colluding adversarial peers in the system, this sample will contain at least one honest peer with high probability. The blocks of the path are fetched by all of the m peers. Each of the m peers then locally computes an encrypted share of the desired block using IT-PIR from the set of input blocks. Note that the tracker must not download the shares or it will violate our scalability requirement. On the other hand, we require to decrypt the block without giving away the private key to the network's peers. For this purpose, we make use of a second cryptographic primitive — a seed homomorphic pseudo-random generator (SH-PRG) [59]. The tracker generates a valid key share for each of the m peers to be used as seeds to the PRG function. Each peer decrypts (or unblinds) its encrypted share using its own key share such that the combination of decrypted shares results in a valid decryption of the original encrypted block in the tree. This property is ensured by SH-PRG and explained in detail in Section 4.2.3. Finally, each peer submits its decrypted share to the initiator peer who combines them to get the desired plaintext block. The colluding peers cannot recover the private key or the encrypted block since there is at least one honest peer who does not disclose its private information. This solves the issues raised in our second attempt.

Remark. OblivSel primitive can be used as a black box

Algorithm 1: IT-PIR protocol by Chor et al. [58]

1 $(r_1, \cdots, r_m) \leftarrow \mathsf{Query}(q, L, \mathsf{pos})$
- randomly generate $m-1$ random vectors such that $r_i \xleftarrow{\$} Z_q^L$
- compute r_m such that for all $j \in [L] \setminus \{\mathsf{pos}\}$, set $r_{m,j} = -\sum_{k=1}^{m-1} r_{k,j}$, otherwise, $r_{m,\mathsf{pos}} = 1 - \sum_{k=1}^{m-1} r_{k,j}$

$R_i \leftarrow \mathsf{Compute}(r_i, \mathsf{DB})$
- parse database such as $DB = (\mathsf{block}_1, \cdots, \mathsf{block}_L)$
- compute $R_i = \sum_{j=1}^{L} r_{i,j} \mathsf{Block}_j$

$\mathsf{block}_{\mathsf{pos}} \leftarrow \mathsf{Recover}(R_1, \cdots, R_m)$: compute $\mathsf{block}_{\mathsf{pos}} = \sum_{j=1}^{m} R_i$

---

in different settings such as distributed ORAMs to decrease the communication overhead. We further show in Section 4.2.4 that OblivSel is highly parallelizable and can leverage peers in the network such that the computation takes constant time.

4.2.3 Base Primitives

Information-theoretic PIR. Information-theoretic private information retrieval (IT-PIR) [58] is a cryptographic primitive that performs oblivious read operations while requiring multiple servers $m \geq 2$. In the following, we present the details of one of the first constructions of IT-PIR by Chor et al. [58] which is secure even when $m - 1$ among m servers collude passively, i.e., the servers collude in order to recover the retrieved block while not altering the protocol. An IT-PIR is a tuple of possibly randomized algorithms $\mathsf{IT-PIR} = (\mathsf{Query}, \mathsf{Compute}, \mathsf{Recover})$. Query takes as an input the block position pos to be retrieved, and outputs an IT-PIR query for m servers. Compute runs independently by every server, takes as input the corresponding IT-PIR query and outputs a share. Recover takes as inputs all shares output by all m servers, and outputs the plaintext block. We give the construction in Algorithm 1.

Seed homomorphic PRG (SH-PRG). A seed homomorphic PRG, G, is a pseudo-random generator over algebraic group with the additional property that if given

---

Algorithm 2: OblivSel with seed-homomorphic PRG

---

1 $(\vec{\sigma}, \vec{r}) \leftarrow \mathsf{Gen}(\mathsf{k}, \mathsf{pos})$

- set $\vec{r} = (r_1, \cdots, r_m) \leftarrow \mathsf{IT-PIR.Query}(\mathsf{q}, \mathsf{L}, \mathsf{pos})$;

- set $\vec{\sigma} = (\sigma_1, \cdots, \sigma_m)$, s.t., $(\sigma_1, \cdots, \sigma_{m-1}) \xleftarrow{\$} \mathbb{S}^{m-1}$, and $\sigma_m = \mathsf{k} - \sum_{i=1}^{m-1} \sigma_i$;

$\mathsf{block} \leftarrow \mathsf{Select}(\vec{\sigma}, \vec{r}, \mathsf{DB})$ // Every peer P_i

- compute $Eshare_i \leftarrow \mathsf{IT-PIR.Compute}(\mathsf{r}_i, \mathsf{DB})$;

- set $Dshare_i = Eshare_i - G(\sigma_i)$;
  // Initiator

- compute $\Delta = \sum_{i=1}^{m} Dshare_i$;

$G(s_1)$ and $G(s_2)$, then $G(s_1 \oplus s_2)$ can be computed efficiently. That is, if the seeds are in a group $(\mathbb{S}, \oplus)$, and outputs in $(\mathbb{G}, \otimes)$, then for any $s_1, s_2 \in \mathbb{S}$, $G(s_1 \oplus s_2) = G(s_1) \otimes G(s_2)$. We refer to [60] for more details.

Decryption / Re-encryption using SH-PRG. Leveraging the property of SH-PRG, we explain the encryption, decryption and re-encryption of a block in our protocol. Every block in the tree is encrypted as $\mathsf{Enc}(\mathsf{k}_1, \mathsf{block}) = \mathsf{block} + G(\mathsf{k}_1)$. The decryption of the block can be then represented as $\mathsf{block} = \mathsf{Dec}(\mathsf{k}_1, \mathsf{Enc}(\mathsf{k}_1, \mathsf{block})) = \mathsf{block} + G(\mathsf{k}_1) - G(\mathsf{k}_1)$ For re-encrypting the encrypted block with a different key k_2, the tracker decrypts the encrypted block with a new secret key of the form $\mathsf{k}_1 - \mathsf{k}_2$ such that, $\mathsf{Dec}(\mathsf{k}_1 - \mathsf{k}_2, \mathsf{Enc}(\mathsf{k}_1, \mathsf{block})) = \mathsf{block} + G(\mathsf{k}_1) - G(\mathsf{k}_1 - \mathsf{k}_2) = \mathsf{block} + G(\mathsf{k}_2) = \mathsf{Enc}(\mathsf{k}_2, \mathsf{block})$.

4.2.4 OblivSel **Instantiation**

In the following, we present an instantiation of OblivSel. We consider a set of L encrypted blocks. Each block block_i is a vector of elements in a finite group $\mathbb{G}$ of order q. For every block, the key is generated at random from $\mathbb{Z}_q$. The tracker has to keep an association between the block key and its position. An algorithmic description is given in Algorithm 2.

The tracker runs the Gen algorithm, which takes as inputs the secret key k with which the block is encrypted and the block's position pos, and outputs a secret shared value of the key, $\vec{\sigma}$, as well as the IT-PIR queries, $\vec{r}$. Every peer P_i holds a copy of the L encrypted blocks and receives a share of the key, σ_i, as well as its corresponding query, r_i. Next, every peer runs locally an $\mathsf{IT-PIR.Compute}$ on the encrypted blocks and outputs a share, $Eshare_i$. After getting the encrypted share $Eshare_i$, each peer subtracts the evaluation of the SH-PRG G on σ_i from $Eshare_i$ ($Eshare_i - G(\sigma_i)$) to get the decrypted share $Dshare$. Finally, initiator outputs the sum of all the $Dshare_i$'s received from the m peers to

get the desired decrypted block. As long as there is one non-colluding peer among the m peers and G is a secure PRG, the scheme is *position hiding*.

Highly Parallelizable. Notice that, in Algorithm 2, each of the m peers performs scalar multiplications proportional to the number of encrypted input blocks. The encrypted blocks can be further distributed to different peers such that each peer performs constant number of scalar multiplications. Given the availability of enough peers in the network, OblivSel is extremely parallelizable and therefore provides a constant time computation.

OblivSel **as a building block.** OblivSel protocol can be used as a building block in our second and main OBLIVP2P-1. For fetching a block, an invocation of OblivSel is sufficient as it obliviously selects the requested block and returns it in plaintext to the initiator. Additional steps such as re-encrypting the block and adding it to stash are required to complete the fetch operation. The details of these steps are in Section 4.3.

However, the eviction operation in ORAM poses an additional challenge. Conceptually, an eviction consists of block sorting, where the tracker re-orders the blocks in the path (and the stash). Fortunately, our protocol can perform eviction by several invocation of OblivSel primitive. Given the new position for each block, the P2P network can be instructed to invoke OblivSel recursively to output the new *sorted* path. The encryption of blocks has to be refreshed, but this is handled within OblivSel protocol itself when refreshing the key, using seed homomorphic PRG. We defer the concrete details of performing oblivious eviction to Section 4.3.

4.3 OBLIVP2P-1: Complete Design

In a P2P protocol for a content sharing system the tracker is responsible for managing the sharing of resources among the peers in the network. To keep a consistent global view on the network, the tracker keeps some *state* information that we formally define below:

Definition 4.4. *P2P network's state consists of: (1) number of possible network connections per peer, and (2) a lookup associating a resource to a (set of) peer identifier.*

The tracker can store more information in the state depending on the P2P protocol instantiating the network. We start first by formalizing a P2P protocol.

Definition 4.5. *A P2P protocol is a tuple of four (possibly interactive) algorithms* $\mathsf{P2P} = (\mathsf{Setup}, \mathsf{Upload}, \mathsf{Fetch}, \mathsf{Sync})$ *involving a tracker, $\mathcal{T}$, and a set of peers, $(P_1, \cdots, P_n)$, such that:*

- $\mathsf{s}' \leftarrow \mathsf{Setup}(\mathsf{s}, \{\mathsf{pid}\})$: *run by the tracker $\mathcal{T}$, takes as inputs a state s and a (possibly empty) set of peers identifiers $\{\mathsf{pid}\}$, and outputs an updated state s'.*

9

Scheme	Tracker bandwidth (bits)	Network bandwidth (# blocks)	Tracker # encryption	Network computational overhead	Network Storage overhead	Tracker storage # blocks
OBLIVP2P-0	$O(\log N \cdot B)$	$O(1)$	$O(\log N \cdot E)$	$-$	$O(1)$	$O(\log N)$
OBLIVP2P-1	$O(\log^3 N)$	$O(\frac{\log N}{N})$	$-$	$O(\frac{\log^4 N}{N} \cdot \mathscr{E})$	$O(\text{burst})$	$-$

Table 2: Comparison of OBLIVP2P instantiation per access. B the block size, N the number of blocks in the network, E the overhead of a block encryption, $\mathscr{E}$ a multiplication in elliptic curve group, burst the number of versions

- $\bigl(\text{out}, (A_1', \cdots, A_m'), s'\bigr) \leftarrow$ Upload$\bigl((\text{fid}, \text{file}), (A_1, \cdots, A_m), s\bigr)$: *is an interactive protocol between an initiator peer, a (possibly randomly selected) set of $m \geq 0$ peers, and a tracker $\mathscr{T}$. The initiator peer has as input a file identifier* fid, *and the file* file, *the peers' input is memory array A_i each, while for the tracker its state* s. *The initiator's output is* out $\in \{\perp, \text{file}\}$, *the peers output each a modified local memory A_i', while the tracker outputs an updated state s'.*

- $(\text{file}, \perp, s') \leftarrow$ Fetch$\bigl(\text{fid}, (A_1, \cdots, A_m), s\bigr)$: *is an interactive protocol between an initiator peer, a (possibly randomly selected) set of $m \geq 0$ peers, and a tracker $\mathscr{T}$. The initiator peer has as input a file identifier* fid, *the peers' input is a memory array A_i each, while for the tracker its state* s. *The initiator outputs the retrieved file* file, *each peer gets $\perp$, while the tracker outputs an updated state* s.

- $\bigl((A_1', \cdots, A_m'), s'\bigr) \leftarrow$ Sync$\bigl((A_1, \cdots, A_m), s\bigr)$: *is an interactive protocol between the tracker and a (possibly randomly selected) set of $m \geq 0$ peers. The peers' input is a memory array A_i each, while for the tracker its state* s. *The peers output each a (possibly) modified memory array A_i', while the tracker outputs an updated state s'.*

Note that a modification of a file already stored in the network is always considered as uploading a new file.

Setup Algorithm. In a P2P network, different peers have different storage capacities and hence we differentiate between the number of blocks, N_B, and the number of physical peers N_P. For this, we fragment the conceptual ORAM tree into smaller chunks where every peer physically handles a number of buckets depending on its local available storage. In addition, to keep a consistent global view on the network, the tracker keeps some *state* information. In OBLIVP2P-1, the state is composed of different meta-information that are independent of the block size: FileMap, PosMap, TagMap, NetMap, KeyMap, and StashMap. Table 1 gives more details about the metadata. The state also contains a counter recording the last eviction step, and $\sim \frac{B}{\log q}$ points sampled randomly from a q-order elliptic curve group $\mathbb{G}$ to be used for DDH seed homomorphic PRG, where B is the block size. The number of points in the generator needs to be equal to those in the data block. These points are publicly known

---

Algorithm 3: Fetch(fid, s): OBLIVP2P-1 fetch operation

Input: file id fid, and state s
Output: file {block}, and updated state s
// Initiator requests tracker for a file
1 $\{adr\} \leftarrow$ FileMap(fid);
2 **for** adr **in** $\{adr\}$ **do**
3     $(\text{tag}, \text{pos}) \leftarrow \bigl(\text{TagMap}(adr), \text{PosMap}(adr)\bigr)$;
4     $k \leftarrow$ KeyMap(adr);
5   compute $(\vec{\sigma}, \vec{r}) :=$ OblivSel.Gen(k, pos);
6   set $A = \bigl(\text{stash}, \mathscr{P}(\text{tag}, 1), \cdots, \mathscr{P}(\text{tag}, L)\bigr)$;
    // Initiator retrieves the block
7   compute block := OblivSel.Select($\vec{\sigma}, \vec{r}, A$);
    // Re-encryption with a new secret
8   compute $k \overset{\$}{\leftarrow} Z_q$;
9   compute $(\vec{\sigma}, \vec{r}) :=$ OblivSel.Gen(k, pos);
10   append $\Delta :=$ OblivSel.Select($\vec{\sigma}, \vec{r}, A$) to the stash, and update state s;
11 **end**

---

to all peers in the network. The tracker randomly distributes the stash among the peers and records this information in the StashList.

Fetch Algorithm. The Fetch process is triggered when a peer requests a particular file. The tracker determines the block tag and position from its state for all the blocks composing the file. The m peers, the tracker, and the initiator runs OblivSel protocol such that the initiator retrieves the desired block. The OblivSel is invoked a second time to add a new layer to the retrieved block and send it to the peer who will hold the stash. The tracker updates its state, in particular, update KeyMap with the new key, update the PosMap with the exact position of the block in the network (in the stash), and TagMap with the new uniformly sampled tag. We provide an algorithmic description of the Fetch process in Algorithm 3.

Sync Algorithm. The Sync in OBLIVP2P-1 consists of: (1) updating the state of the network, but also, (2) evicting the stash. The tracker determines the path to be evicted, tag $= \nu \bmod 2^L$ and then fetches the position of all blocks in the stash and the path, $\mathscr{P}(\text{tag})$. The tracker then generates, based on the least common ancestor algorithm (LCA), a permutation π that maps every block in $A = \bigl(\text{stash}, \mathscr{P}(\nu \bmod 2^L, 1), \cdots, \mathscr{P}(\nu \bmod 2^L, L)\bigr)$ to its new position in A', a new array that will replace the evicted path and the stash. The block $A[\pi(i)]$ will be mapped *obliviously* to $A'[i]$, for all $i \in [|\text{stash}| + z \cdot L]$. The oblivious mapping between A and A' is performed by invoking OblivSel between the tracker, the peers in

Algorithm 4: Sync(s): OBLIVP2P-1 sync operation

Input: tracker state s
```
// Fetch necessary parameters
```
1 $v \leftarrow s$;
2 $\{adr\} \leftarrow \mathsf{PosMap}^{-1}(v \bmod 2^L)$;
3 **for** adr **in** $\{adr\}$ **do**
4 | set $T = T \cup tag \leftarrow \mathsf{TagMap}(adr)$;
5 **end**
6 set $A = (\mathsf{stash}, \mathscr{P}(v \bmod 2^L, 1), \cdots, \mathscr{P}(v \bmod 2^L, L))$;
7 Initialize an array A', $\pi \leftarrow \mathsf{LCA}(T, v)$;
```
// tracker generates key shares
```
8 **for** l **from** 1 **to** $z \cdot L + |\mathsf{stash}|$ **do**
9 | **if** $\exists adr, l = \mathsf{PosMap}(adr)$ **then**
10 | | set $k \leftarrow \mathsf{KeyMap}(adr)$;
11 | | set $k'' = k' - k$, $k' \xleftarrow{\$} Z_q$;
12 | | compute $(\vec{\sigma}_l, \vec{r}_l) = \mathsf{OblivSel.Gen}(k'', \pi(l))$;
13 | **else**
14 | | set $k'' \xleftarrow{\$} Z_q$, compute $(\vec{\sigma}_l, \vec{r}_l) = \mathsf{OblivSel.Gen}(k'', \pi(l))$;
15 | **end**
16 **end**
```
// Peers generate the new array A'
```
17 **for** j **from** 1 **to** $z \cdot L + |\mathsf{stash}|$ **do**
18 | set $A'[j] = \mathsf{OblivSel.Select}(\vec{\sigma}_j, \vec{r}_j, A)$;
19 **end**
20 **for** $j \in [m]$, send $A'_j[1, \cdots, |\mathsf{stash}|]$ and $A'_j[|\mathsf{stash}| + 1, \cdots, L]$ to peers in $\mathscr{P}(v)$ and the stash, and update state s;

---

the path and m peers, $|\mathsf{stash}| + z \cdot L$ times. Note that (1) the blocks in A' are encrypted with a freshly-generated key, and (2) the mapping is not disclosed to any peers in the path as long as there is one non-colluding peer. Refer to Algorithm 4 for more detail about the Sync algorithm.

Upload Algorithm. A peer can request the tracker to add a file. For this, the tracker selects uniformly at random a set of m peers. The peer sends the file in a form of blocks. Every block is secret shared such that every peer in the m peers receives a share. The tracker generates a secret unique to the block, k. The tracker secret shares k to the m peers. The peers evaluate a seed-homomorphic PRG on the received shares and add it to the block share. Finally, the block is appended to a randomly selected peer in the network to hold a part of the stash.

4.4 Optimization: Handling Bursts

OBLIVP2P-1 has a functional limitation inherited by ORAMs. Any access cannot be started unless the previous one has concluded [2]. In our case, the tracker can handle fetching several blocks before starting the Sync operation. In our setting, we target increasing the P2P network throughput while leveraging the network storage and communication. In order to build a scalable system, we propose several optimizations.

O1: Replication. In Ring ORAM, $A = 3$ accesses can be performed before an eviction is required. To support A parallel accesses, we replicate every block A times in the tree. This absorbs the fetching access time and allows A

---
[2]We do not consider a multi-processor architectures as those considered in OPRAM literature [56].

---

simultaneous accesses, even for requests to the same resource. Additionally, we may replicate every block over A times on different peers, in case that the peer holding the block is offline due to churn, and cannot serve the block to the other peers. Lastly, the network operator can deploy multiple trackers to serve peers simultaneously, which leads to the throughput of OBLIVP2P-1 proportional to the number of trackers.

O2: Pipelining. While the eviction is highly parallelizable in OBLIVP2P-1, an eviction can take a considerable amount of time to terminate. If we denote by f the average number of fetch requests in the P2P network, and by t the time to perform an eviction, then the system can handle all the accesses if $t < \frac{1}{f}$. However, in practice $t > \frac{1}{f}$ and therefore the accesses will be queued and creates a bottleneck. To address this issue, we create multiple copies of the buckets that are run with different instances of OBLIVP2P-1 protocol which overlays on the same network. In the setup phase, every node creates l copies of its bucket space. Every bucket will be associated to different versions of OBLIVP2P-1 instantiations. For example, with replication we can handle A accesses in parallel on the (same) ith version of the buckets, but the upcoming accesses will be made on the $(i+1)th$ version. This will absorb the eviction time. To sum up, having different versions will increase the throughput of the system to $\frac{l}{f}$. In order to prevent pipeline stalls, we need to choose $l \geq t \cdot f$ in our implementation.

Another aspect (not considered for our implementation) for further optimizations in our versioning solution is to distribute the communication overhead of the peers in the network. In fact, the peers holding blocks at the higher level of the tree will be accessed more often compared to lower levels. In order to distribute the communication load on the network peers, peers' location can be changed for different versions such that: the peer at the ith level of the tree in the jth version will be placed at the $(L-i+1)th$ level of the tree in the $(j+1)th$ version.

O3: Parallelizing Computation across m Peers. The scalar multiplication in the elliptic curve is expensive and can easily delay the fetch and sync time. For this, we consider every peer in the OblivSel as a set of peers. Whenever there is a need to perform scalar multiplication over a path, several peers participate in the computation and only the representative of the set will perform the aggregation. This optimization speeds up the OblivSel to be proportional to the number of peers' used to parallelize a single peer.

5 Implementation and Evaluation

Implementation. We implement a prototype of OBLIVP2P-0 and OBLIVP2P-1 in Python. The im-

plementation contains 1712 lines of code (LOC) for OBLIVP2P-0 and 3226 for OBLIVP2P-1 accounting to a total of 4938 lines measured using CLOC tool [61]. Our prototype implementation is open source and available online [34]. As our building block primitives, we implement the Ring ORAM algorithm, IT-PIR construction and seed-homomorphic PRG. For Ring ORAM, we have followed the parameters reported by authors [52]. Each bucket contains $z = 4$ blocks and $s = 5$ dummy blocks. The eviction occurs after every 3 accesses. The blocks in OBLIVP2P-0 are encrypted using AES-CBC with 256 bit key from the pycrypto library [62]. For implementing IT-PIR and seed homomorphic PRG in OBLIVP2P-1, we use the ECC library available in Python [63]. We use the NIST P-256 elliptic curve as the underlying group.

Experimental Setup. We use the DeterLab network testbed for our experiments [64]. It consists of 15 servers running Ubuntu 14.04 with dual Intel(R) Xeon(R) hexa-core processors running at 2.2 Ghz with 15 MB cache (24 cores each), Intel VT-x support and 24 GB of RAM. The tracker runs on a single server while each of the remaining servers runs approximately 2400 peers. Every peer process takes up to $4 - 60$ MB memory which limits the maximum network size to 2^{14} peers in our experimental set up. The tracker is connected to a 128 MBps link and the peers in each server share a bandwidth link of 128 MBps as well. We simulate the bandwidth link following the observed BitTorrent traffic rate distribution reported in [65]. In our experimental setting, multiple peers are simulated on a single machine hence our reported results here are conservative. In the real BitTorrent setting, every peer has its own separate CPU.

Evaluation Methodology. To evaluate the scalability and efficiency of our system, we perform measurements for a) the overall throughput of the system b) the latency for Fetch and Sync operations and c) the data transferred through the tracker for both OBLIVP2P-0 and OBLIVP2P-1. All our results are the average of 50 runs with 95% confidence intervals for each of them. Along with the experimental results, we plot the theoretical bounds computed based on Table 2. This helps us to check if our experiments match our theoretical expectations. In addition, we perform separate experiments to demonstrate the effect of our optimizations on the throughput of our OBLIVP2P-1 protocol. For our experiments in this section, we leverage the technical optimization introduced in Section 4.4.

We vary the number of peers in the system from 2^4 to 2^{14} peers (capacity of our testbed) and extrapolate them to 2^{21} peers. Note that, when increasing the number of peers, we implicitly increase the total data size in the entire network which is computed as the number of peers $\times$ the block size. That is, our P2P network handles a total

data size that spans from 16 KB to 32 GB. For our evaluation, we consider each peer holds one ORAM bucket because of the limited available memory. In reality, every peer can hold more buckets. Note that, we linearly extrapolate our curves to show the expected results for larger number of peers starting from $2^{15} - 2^{21}$ (shown dotted in the Figures) , and therefore larger data size in the network. Aligned to the chunks in BitTorrent, we select our blocksize as 128 KB, 512 KB and 1 MB.

## 5.1 Linear Scalability with Peers

The throughput is an important parameter in designing a scalable P2P protocol. We define the throughput, as the number of bits that the system can serve per second.

From Figure 3a, we observe that the throughput of OBLIVP2P-0 decreases with the increase in the total number of peers in the network. For a network size of 2^{14} peers, the experimental maximum throughput is 0.91 MBps. As we extrapolate to larger network size, the maximum throughput decreases, e.g., for 2^{21} peers, the throughput is 0.64 MBps. This shows that as the network size increases, the tracker starts queuing the requests that will eventually lead to a saturation. However, for OBLIVP2P-1, the maximum throughput for network size of 2^{14} is 3.19 MBps and is 3.29 MBps when extrapolated to 2^{21} peers. The throughput increases as there are more peers available in the network to distribute the computation costs. The throughput shows a similar behaviour for blocksize of 128 KB and 1 MB (as shown in Figure 5). Hence, we expect OBLIVP2P-1 to provide better throughput in a real setting where more computational and communication capacity for each peer can be provisioned. The throughput values for OBLIVP2P-1 are calculated after applying all the 3 optimizations discussed in Section 4.4. The behaviour of the theoretical throughput matches our experimental results. The theoretical throughput has higher values as it does not capture the network latency in our test environment.

Result 1. Our results show that the centralized protocol is limited in scalability and cannot serve a large network. Whereas, the throughput for OBLIVP2P-1 linearly scales $(0.15 - 3.39$ MBps) with increasing number of peers $(2^5 - 2^{21})$ in the network.

Result 2. For a block of size 512 KB and 2^{14} peers, OBLIVP2P-1 serves around 7 requests / second which can be enhanced with multiple copies of ORAM trees in the network.

Remark. The throughput may be acceptable to privacy-conscious users (e.g., whistleblowers), where privacy concerns outweigh download / upload latencies. As long as the number of request initiators is small, the perceived throughput remains competitive with a non-
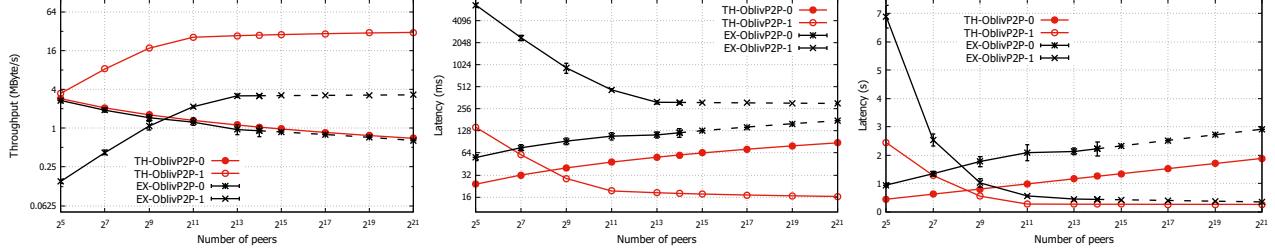
(a) The throughput for OBLIVP2P-1 linearly scales with the increase in network size.

(b) The latency for fetching a block for OBLIVP2P-1 reduces up to 2^{13} and then becomes constant.

(c) The latency for sync operation for OBLIVP2P-1 reduces up to 2^{13} and then becomes constant.

Figure 3: Theoretical (Th) and experimental (Ex) comparison of OBLIVP2P-0 and OBLIVP2P-1 parameters for block size of 512 KB
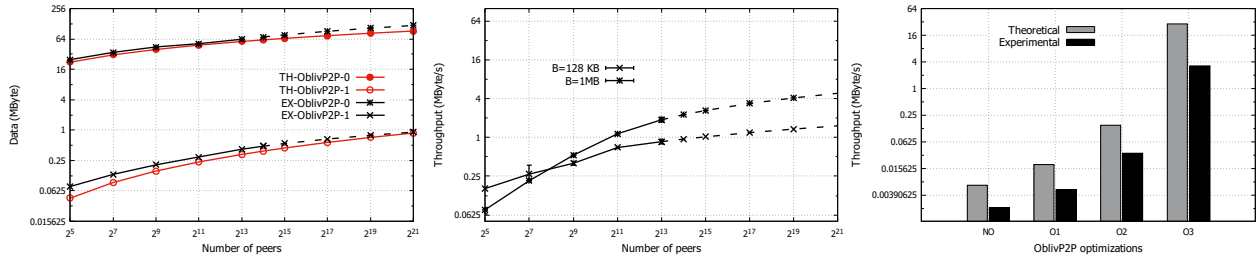

Figure 4: The data transferred through the tracker for OBLIVP2P-0 increases linearly with the number of peers

Figure 5: The throughput for blocksize 128 KB and 1 MB increases with increase in the network size.

Figure 6: Impact of optimizations (O1-O3) on the throughput of OBLIVP2P-1 for 2^{14} peers and blocksize of 512 KB.

oblivious P2P system. Further, the network operator can deploy multiple trackers to serve peers simultaneously, which leads to the throughput of OBLIVP2P-1 proportional to the number of trackers.

5.2 Latency Overhead and Breakdown

We define the latency as the time required to perform one ORAM operation in our P2P protocol. We measure the latency for the following operations:

Fetch. Figure 3b shows that the average time for fetching a block of 512 KB increases for OBLIVP2P-0 with increase in the size of the network. This is due to the increased computation and bandwidth overhead at the tracker. However, for OBLIVP2P-1, the latency initially reduces with the increasing number of peers (from 2^5 to 2^{11}) and then becomes constant after the network is large enough (around 2^{13}) to distribute the computation cost in the network[3]. OBLIVP2P-1 has a higher latency for fetch as compared to OBLIVP2P-0 due to the expensive computation required for performing scalar multiplication. The average time for fetching a block of size 512 KB is around 0.31 s for a network size of 2^{14} peers and

---

[3]Since a large number of nodes (e.g., over 1000 nodes) share one physical machine, its limited computation power drastically affects our result. Therefore, to be more realistic, we use the ideal computing and decoding/encoding time for each node solely in one physical machine as the computing time per node, and simulate our experiments for OBLIVP2P-1.

remains steady with increase in the number of peers.

Sync. We measure the time for performing a sync operation for different network sizes. Figure 3c shows that the time for performing a sync operation increases in OBLIVP2P-0 with increase in the number of peers. Whereas for OBLIVP2P-1, the sync time reduces gradually at first and then becomes steady after the network size reaches 2^{13} peers which is as expected through our theoretical calculation. OBLIVP2P-1 uses the peers in the network to distribute the computation load and hence the sync time tends to be steady for large network sizes.

Data transferred through tracker. Figure 4 shows the amount of data that is transferred through the tracker per request. We perform this measurement to show that the centralized tracker becomes a bottleneck in OBLIVP2P-0. The amount of data that the tracker has to process increases with increase in the number of peers. At 2^{21} peers, the amount of data is 118 MB (almost) reaching the bandwidth limit (128 MBps) of the tracker. Whereas, for OBLIVP2P-1 the amount of data transferred is around 1 MB for 2^{21} peers. This implies that the tracker could manage up to 128 copies of ORAM tree in parallel, which will increase the overall throughput by 128 times.

Result 3. OBLIVP2P has no centralized infrastructure as a bottleneck, ensuring that communication and computational overhead can be completely offloaded to the network.

13

5.3 Optimization Measurements

We perform incremental experiments to quantify the impact of each of the introduced optimizations on the overall throughput in Section 4.4, as shown in Figure 6. We fix the number of peers in the network to be equal to 2^{14} and the block size to 512 KB. We chose our optimization parameters based on our results in section 5.3. We fix the number of replicas to be equal to $A = 3$, i.e., the same data block is replicated three times. The burst parameter needs to be in $O(\frac{B}{\log q} \log N_p)$, where N_p is the number of peers, B the block size, and q the elliptic curve group order. Finally, we fix the number of parallel peers in the OblivSel.Select algorithm to be in $O(\frac{B}{\log q} \log N_p)$.

O1: Replication. Replication enables to perform $A = 3$ fetch operations in parallel. This implies that the throughput *theoretically* increases 3 times when compared to our baseline without any optimizations. Our experimental results show that we have 2.55 times improvement over the baseline, as expected theoretically.

O2: Pipelining. We evaluate the effect of our optimization (O2) that absorbs the eviction time by pipelining the fetch requests to different versions of the ORAM tree in the P2P network. We show that this optimization, when coupled to (O1), has theoretically increased the overall throughput by 23.05 times if compared to the baseline. Our experiments are aligned to our theoretical results and show 17.2 times improvement over the baseline with a burst parameter of 17. Clearly, if the number of versions increases beyond 17, then OBLIVP2P-1 can handle parallel accesses, hence increasing the system throughput.

O3: Parallelizing m peers. We measure the effect of parallelizing the computation load of m peers by leveraging more peers in the network on the overall throughput of the system. We increase the number of peers to 116 peers that are used to compute the fetch and sync operations. Our theoretical result shows an improvement of 4398 times over the baseline, when coupled with (O1) and (O2). Our experiments support this result and demonstrates 1589 times improvement, the difference is due to the real network latency are not considered in our theoretical calculation.

Result 4. OBLIVP2P-1 is subject to several optimizations due to its highly parallelizable design.

6 OBLIVP2P-1 Analysis

In this section, we present the theoretical analysis on computation / communication overhead of tracker / peers and security analysis for OBLIVP2P-1.

6.1 Performance

We report OBLIVP2P-1 computation and communication overhead for the tracker and the network in Table 2. In particular, OBLIVP2P-1's tracker transmits a number of bits *independent* of the block size, the tracker does not perform any computation on the blocksize or store any block locally.

Tracker overhead. To fetch a block, the tracker invokes OblivSel twice. While for the eviction, the tracker performs OblivSel $(L \cdot z + |\mathsf{stash}|)$ times. That is, it is sufficient to first analyze OblivSel overhead and than just conclude for the overall tracker overhead.

Within one instance of OblivSel, the tracker computes an $\mathsf{IT} - \mathsf{PIR.Query}$ that outputs m vectors for m peers, each of size $L \cdot z + |\mathsf{stash}|$. Each $\mathsf{IT} - \mathsf{PIR.Query}$ vector costs $\log q (L \cdot z + |\mathsf{stash}|)$ bits, where q is the group order. The tracker also needs to generate shares for the key, where the shares are in $\mathbb{Z}_q$. That is, one OblivSel costs the tracker $O\big(m \cdot \log q \cdot (L \cdot z + |\mathsf{stash}|)\big)$.

That is, the tracker has to transmit $O\big(m \cdot \log q \cdot (L \cdot z + |\mathsf{stash}|)^2\big)$ bits. Considering $L, |\mathsf{stash}| \in O(\log N)$, q the group order in $\mathsf{poly}(N)$, m the number of peers and z the bucket size as constants, then the tracker needs to send $O(\log^3 N)$ bits independently of the block size. That is, if block $\in \Omega(\log^3 N)$, the tracker has a constant communication work per block. Moreover, the tracker is very lightweight as it does not perform any heavy computation such as encryption, decryption of blocks, which permits the tracker to handle frequent accesses.

Peers overhead. Considering the communication between the peers, the main communication overhead comes from block transfer from the peers holding the path to the selected m peers. The m peers are selected uniformly at random. Each peer receives $(z \cdot L + |\mathsf{stash}|)$ blocks from the peers in the selected path and the stash. That is, in terms of communication overhead, the peers sends on average $\sum_{i=0}^{L} \frac{z}{2^i} + \frac{(z \cdot L + |\mathsf{stash}|)}{N} + \frac{z}{N}$ blocks per peers in the network. Considering z constant and $L, |\mathsf{stash}| \in O(\log N)$, implies that every peer is expected to transmit $O(\frac{\log N}{N})$ blocks per access.

In terms of computation, the main computational bottleneck consists of the scalar multiplication from the seed homomorphic PRG. For every OblivSel, every peer needs to perform $(z \cdot L + |\mathsf{stash}|) \cdot \frac{B}{\log q}$ scalar multiplications per block. The second term, $\frac{B}{\log q}$, represents the number of points that a block contains. We also have $(z \cdot L + |\mathsf{stash}|)$ instances of OblivSel during the eviction. That is, the total number of scalar multiplication equals $O\big((z \cdot L + |\mathsf{stash}|)^2 \cdot \frac{B}{\log q}\big)$. Finally, the amortized computation over the total number of peers in the network equals $O(\frac{\log^4 N}{N})$ multiplications per eviction, considering $B \in \Omega(\log^3 N)$ and $q \in \mathsf{poly}(N)$.

14

6.2 Security Analysis

We show that OBLIVP2P-1 is an *oblivious* P2P as stated by Definition 2.2. For this, it is sufficient to show that an adversary cannot distinguish between a randomly generated string and the access pattern leaked by any peer's real access. This underlines the fact that the access pattern is independent of the address of the requested block. In our threat model, the adversary can have access to the content of buckets, monitors the communication between the peers, and has a total view of the internal state of dishonest peers. Buckets' content is assumed to be transmitted without any additional layer of encryption.

We present our *address-tag* experiment AT that captures our security definition. Let OBLIVP2P $=$ (Setup, Upload, Fetch, Sync) represents an oblivious P2P protocol. Let $\mathcal{E} =$ (Gen, Enc, Dec) be an IND\$ $-$ CPA encryption scheme. Let $\mathcal{G}$ be a secure pseudo-random generator. $\mathsf{AT}^{OblivP2P}_{\mathcal{A},\mathcal{E},\mathcal{G}}$ refers to the instantiation of the *address-tag* experiment by algorithms OBLIVP2P, $\mathcal{E}$, $\mathcal{G}$, and adversary $\mathcal{A}$. We denote by Col the event that m peers in the network collude and set $\Pr[\mathsf{Col}] = \delta_m$, by $\mathscr{B}_{\delta_m}$ the Bernoulli distribution, and λ the security parameter.

In the following, we fix the number of colluding peers $c \in O(N^{\varepsilon})$, for $0 < \varepsilon < 1$. We consider every peer in the network as a random variable distributed based on a Bernoulli distribution with probability equal to $\frac{c}{N} \in O(N^{\varepsilon-1})$. Let us denote by $(X_1, \cdots, X_m)$ the random variables of the selected peers for every instantiation of OblivSel. Note that $\Pr[\mathsf{Col}] = \Pr[X_1 = 1 \text{ AND} \cdots X_m = 1]$. Since all X_i's are independent, then, $\Pr[\mathsf{Col}] = \Pi_{i=1}^{m} \Pr[X_i = 1] = (\frac{c}{N})^m$. That is, $\delta_m = (\frac{c}{N})^m$ which implies under our assumptions that $\delta_m \in O(2^{\log N \cdot m \cdot (\varepsilon-1)})$.

In the following experiment, we only consider the Fetch algorithm for obliviousness analysis. In our model, Upload sequences are indistinguishable by construction assuming that peers uploads blocks that are randomly distributed, and using random key for every block encryption. The experiment $\mathsf{AT}^{OblivP2P}_{\mathcal{A},\mathcal{E},\mathcal{G}}(\lambda, b)$ consists of:

- The adversary $\mathcal{A}$ picks one access operation (Fetch, adr, $\perp$) and sends it to the challenger $\mathscr{C}$
- If $b = 1$, pick $X \xleftarrow{\mathscr{B}_{\delta_m}} \{0,1\}$, if $X = 1$, then set var = adr , otherwise var = $\perp$ and set

$$\pi_1 = \{ (\mathscr{P}(\mathsf{tag},1), \cdots, \mathscr{P}(\mathsf{tag},L)), \mathsf{tag} \leftarrow \mathsf{TagMap}[\mathsf{adr}],$$
$$(\mathsf{Enc}(q_1), \cdots, \mathsf{Enc}(q_m)),$$
$$(q_1, \cdots, q_m) \leftarrow \mathsf{IT}-\mathsf{PIR.Query}(\mathsf{pos}),$$
$$\mathsf{pos} \leftarrow \mathsf{PosMap}[\mathsf{adr}], \mathsf{var}\}$$

If $b = 0$, set $\pi_0 = \{(P_1, \cdots, P_L) \xleftarrow{\$} \mathbb{G}^{z \times L},$
$(q_1, \cdots, q_m) \xleftarrow{\$} \{0,1\}^{\lambda \times m}, \perp\}$

- Adversary $\mathcal{A}$ has access to an oracle $\mathscr{O}^{OblivP2P}$ that issues the access patterns for polynomial number of accesses (while paths are re-encrypted for every request)
- $\mathcal{A}$ outputs b'
- The output of the experiment is 1 if $b = b'$, otherwise 0. If $\mathsf{AT}^{OblivP2P}_{\mathcal{A},\mathcal{E},\mathcal{G}}(\lambda, b') = 1$, we say that $\mathcal{A}$ won the experiment.

The experiment differentiates between a realistic setting where the adversary can see the access pattern, and in which a possible colluding setting can happen with a pre-fixed probability, δ_m, and an ideal setting where the adversary receives a random string. We slightly reformulate Definition 2.2 below.

Definition 6.1. *We say that a P2P is oblivious iff for all PPT adversaries $\mathcal{A}$, there exists a negligible function* negl *such that:*

$$\Pr[\mathsf{AT}^{OblivP2P}_{\mathcal{A},\mathcal{E},\mathcal{G}}(\lambda, 1) = 1] - \Pr[\mathsf{AT}^{OblivP2P}_{\mathcal{A},\mathcal{E},\mathcal{G}}(\lambda, 0) = 1] \leq \mathsf{negl}(\lambda)$$

Theorem 6.1. *If $\forall N > 1$, and $\forall \varepsilon < 1$, $\exists m > 1$ s.t. $2^{\log N \cdot m \cdot (1-\varepsilon)} \in \mathsf{negl}(\lambda)$, $\mathcal{G}$ is a secure pseudo-random generator, $\mathcal{E}$ is IND\$ $-$ CPA secure, then OBLIVP2P-1 is an oblivious P2P as in Definition 6.1.*

Proof. To prove our theorem, we proceed with a succession of games as follows:

- Game$_0$ is exactly the same as $\mathsf{AT}^{OblivP2P}_{\mathcal{A},\mathcal{E},\mathcal{G}}(\lambda, 1)$
- Game$_1$ is the same as Game$_0$ except that the blocks in the buckets $\mathscr{P}(\mathsf{tag}, i)$ are replaced with random points from $\mathbb{G}$
- Game$_2$ is the same as Game$_1$ except that the the encrypted IT $-$ PIR queries are replaced with random strings

From games' description, we have

$$\Pr[\mathsf{Game}_0 = 1] = \Pr[\mathsf{AT}^{OblivP2P}_{\mathcal{A},\mathcal{E},\mathcal{G}}(\lambda, 1) = 1], \quad (1)$$

For Game$_1$, we can build a distinguisher B_1 that reduces security of $\mathcal{G}$ to PRG security such that:

$$\Pr[\mathsf{Game}_0 = 1] - \Pr[\mathsf{Game}_1 = 1] \leq \mathsf{Adv}^{PRG}_{B_1, \mathcal{G}}(\lambda), \quad (2)$$

Similarly for Game$_1$, we can build a distinguisher B_2 that reduces $\mathcal{E}$ to IND\$ $-$ CPA security such that:

$$\Pr[\mathsf{Game}_1 = 1] - \Pr[\mathsf{Game}_2 = 1] \leq \mathsf{Adv}^{\mathsf{IND\$-CPA}}_{B_2, \mathcal{E}}(\lambda), \quad (3)$$

We need now to compute $\Pr[\mathsf{Game}_2]$.

$$\Pr[\mathsf{Game}_2] = \Pr[\mathsf{Col}] \cdot \Pr[\mathsf{Game}_2 = 1 | \mathsf{Col}] +$$
$$\Pr[\overline{\mathsf{Col}}] \cdot \Pr[\mathsf{Game}_2 = 1 | \overline{\mathsf{Col}}]$$
$$= \delta_m + (1 - \delta_m) \frac{1}{N}$$

On the other side $\Pr[\mathsf{AT}^{OblivP2P}_{\mathscr{A},\mathscr{E},\mathscr{G}}(\lambda,0)=1]=\frac{1}{N}$, since the tag is generated uniformly at random for every access.

$$\Pr[\mathsf{Game}_2] - \Pr[\mathsf{AT}^{OblivP2P}_{\mathscr{A},\mathscr{E},\mathscr{G}}(\lambda,0)=1] = \delta_m(1-\frac{1}{N}) \quad (4)$$

From equations 1, 2, 3, and 4 we obtain:

$$\Pr[\mathsf{AT}^{OblivP2P}_{\mathscr{A},\mathscr{E},\mathscr{G}}(\lambda,1)] - \Pr[\mathsf{AT}^{OblivP2P}_{\mathscr{A},\mathscr{E},\mathscr{G}}(\lambda,0)=1] \leq$$
$$\delta_m(1-\frac{1}{N}) + \mathsf{Adv}^{\mathsf{IND\$-CPA}}_{B_2,\mathscr{E}} + \mathsf{Adv}^{PRG}_{B_1,\mathscr{G}}.$$

Since $\delta_m \in O(2^{\log N \cdot m \cdot (\varepsilon-1)})$, this ends our proof.
□

Quantitatively, if the number of peers in the network equals 2^{20}, number of colluding peers in the network is $c = N^{\frac{1}{2}}$ and $m = 12$, then $\delta_{12} = 2^{-120}$. Given the number of colluding peers and total number of peer, the value of m can always be adjusted to handle the desired colluding probability δ_m. In case of churn, the fraction c can vary and therefore the length of the circuit m has to be adapted to the new value. Furthermore, we implicitly assumed so far that no peer among the m selected leaves in the middle of the OblivSel process. If that occurs, the entire process has to abort, re-calculates the number of required peers m, and perform the OblivSel from scratch.

7 Discussion

Existing approaches. A valid question to investigate is whether existing solutions such as unlinkability or path non-correlation techniques can be extended to handle global adversaries and therefore prevent traffic analysis at the cost of providing more resources. It is easy to see that unlinkability techniques (e.g., mixnet) can provide better security in a P2P network under some assumptions. As an instance, assuming the case where a large number of peers behave as senders and issue requests that will be mixed by *sufficient* network peers before being answered by corresponding receivers' peers. Also, assuming that there is at least one honest peer in the mixing network, this solution would provide slightly the same level of security as OBLIVP2P where a global adversary cannot distinguish the senders' peers access pattern. However, this solution suffers from two downsides. First, there is a need to have *sufficient* number of senders' peers *on-line* in order to prevent intersection attacks. That is, in order to prevent traffic analysis, the number of senders represents a security parameter of the system that has to be maintained throughout the entire run of the system. Second, as the receivers' contents are theirs and are not encrypted, plus, all peers are considered honest-but-curious, a global adversary can easily find out what content is being accessed independently of the sender identity. This therefore does not achieve obliviousness as defined in our work but only a weaker version of it. On

the other hand, path non-correlation techniques conceptually cannot prevent against global adversary as we have detailed in Section 2. To sum up, it is not clear if existing techniques, even if given enough resources, can provide similar security insurances as those in OBLIVP2P.

Does better network & computation help? As empirically demonstrated in our evaluation section, the throughput of OBLIVP2P is around 3.19 MBps while considering only *one* tracker in the network. In a plaintext version of P2P system such as BitTorrent, the network leverages multiple trackers in order to handle more queries, and therefore increase the overall throughput. In OBLIVP2P, if we consider multiple copies of the entire network, we can also handle multiple trackers, and the throughput is expected to increase linearly with the number of trackers. However, as we delegate computation to the peers in OBLIVP2P, increasing the number of trackers beyond a particular threshold might turn out to be useless as the computation would represent a bottleneck of the system. As future work, we plan to investigate the asymptotic and empirical implications of including multiple trackers in the system. Moreover, it would be interesting to find out the relation between the number of trackers, number of peers for an ideal throughput of OBLIVP2P.

8 Related Work

Long-term traffic analysis. Anonymous systems like mix networks and onion routing are susceptible to long-term traffic analysis as shown in Section 2. Statistical disclosure attacks proposed by Danezis and enhanced by other researchers improve the likelihood of de-anonymizing users on these systems [66–73]. Moreover, existing traffic analysis attacks on onion routing based approaches [27–30, 45] can reveal users' identities with observing multiple communication rounds. Other P2P systems like Crowds [19], Tarzan [18], MorphMix [20], AP3 [21], Salsa [22], ShadowWalker [23], Freenet [3] offer anonymity for users. However, these systems show limits against global adversary with long-term traffic analysis capabilities.

Side-Channels. Previous work has shown possible attacks by leveraging side channels such as packet sizes, number of packets and timing. These side channels leak users' private information, e.g., illnesses/medications/-surgeries, income and investment secrets [74]. An attacker can employ machine learning techniques (e.g., Support Vector Machines) on network traffic to identify the user's browsing websites [28–30, 45]. However, our focus in this paper is to only prevent long-term pattern traffic analysis. The aforementioned side-channels of traffic analysis are out of scope.

Multi-servers and parallel ORAM. There have been works on how to optimize ORAM constructions while leveraging multiple servers [75–77], multiple CPUs [56, 78], computational servers [53, 54], or distributed under a weaker threat model [55] . However, none of these recent constructions fit to a P2P setting as is. This is *mainly* due to the inherent client / server setting that results on a single entity bottleneck. The client has to either perform non-trivial computation or/and transmit several amount of bits. We briefly discuss these works below.

OblivStore [76], Lu and Ostrovsky [75], and Stefanov and Shi [77] demonstrate how to decrease the communication overhead while leveraging multiple ORAM nodes and servers. However, all these constructions are centralized and route the block through the tracker. This leads to a single entity bottleneck.

Recently, researchers have proposed oblivious parallel RAM (OPRAM) [56, 78]. This was motivated by current multi-cpu architectures that can access the same or multiple resources in parallel. However, OPRAM does not decrease the communication overhead making it as well a single-entity bottleneck. Dachman-Soled et al. introduced oblivious network RAM (ONRAM) [55]. ONRAM can reduce the communication overhead between the client and multiple banks of memory to be constant in the number of blocks. However, it assumes a weak threat model, and cannot achieve obliviousness in the case of a global adversary.

9 Conclusion

We advocate hiding data access patterns as a necessary step in defenses against long-term traffic pattern analysis in P2P content sharing systems. To this end, we propose OBLIVP2P— an oblivious peer-to-peer protocol. Our evaluation demonstrates that OBLIVP2P is parallelizable and linearly scalable with increase in number of peers, without bottleneck on a single entity.

References

[1] "Bittorrent," http://www.bittorrent.com/.

[2] "Storj.io," http://storj.io/.

[3] "Freenet: The free network," https://freenetproject.org.

[4] "Akamai," http://www.akamai.com/.

[5] S. Iyer, A. Rowstron, and P. Druschel, "Squirrel: A decentralized peer-to-peer web cache," in *PODC*, 2002.

[6] "Utorrent & bittorrent surge to 150 million monthly users," https://torrentfreak.com/bittorrent-surges-to-150-million-monthly-users-120109/.

[7] "Palo alto networks application usage & threat report," http://researchcenter.paloaltonetworks.com/app-usage-risk-report-visualization/.

[8] M. Piatek, T. Kohno, and A. Krishnamurthy, "Challenges and directions for monitoring p2p file sharing networks, or, why my printer received a dmca takedown notice," in *HotSec*, 2008.

[9] G. Siganos, J. M. Pujol, and P. Rodriguez, "Monitoring the bittorrent monitors: A bird's eye view," in *PAM*, 2009.

[10] S. Le Blond, C. Zhang, A. Legout, K. Ross, and W. Dabbous, "I know where you are and what you are sharing: exploiting p2p communications to invade users' privacy," in *IMC*, 2011.

[11] D. L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, 1981.

[12] G. Danezis, R. Dingledine, and N. Mathewson, "Mixminion: Design of a type iii anonymous remailer protocol," in *IEEE S&P*, 2003.

[13] U. Möller, L. Cottrell, P. Palfrader, and L. Sassaman, "Mixmaster protocol-version 2," 2003.

[14] M. G. Reed, P. F. Syverson, and D. M. Goldschlag, "Anonymous connections and onion routing," *J-SAC*, 1998.

[15] R. Dingledine, N. Mathewson, and P. F. Syverson, "Tor: The second-generation onion router," in *USENIX Security*, 2004.

[16] T. Wang, K. Bauer, C. Forero, and I. Goldberg, "Congestion-aware path selection for tor," in *FC*, 2012.

[17] M. Akhoondi, C. Yu, and H. V. Madhyastha, "Lastor: A low-latency as-aware tor client," in *IEEE S&P*, 2012.

[18] M. J. Freedman and R. Morris, "Tarzan: A peer-to-peer anonymizing network layer," in *CCS*, 2002.

[19] M. K. Reiter and A. D. Rubin, "Crowds: Anonymity for web transactions," *TISSEC*, 1998.

[20] M. Rennhard and B. Plattner, "Introducing morphmix: peer-to-peer based anonymous internet usage with collusion detection," in *WPES*, 2002.

[21] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach, "Ap3: Cooperative, decentralized anonymous communication," in *SIGOPS European Workshop*, 2004.

[22] A. Nambiar and M. Wright, "Salsa: a structured approach to large-scale anonymity," in *CCS*, 2006.

[23] P. Mittal and N. Borisov, "Shadowwalker: peer-to-peer anonymous communication using redundant structured topologies," in *CCS*, 2009.

[24] A. Pfitzmann and M. Hansen, "Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management-a consolidated proposal for terminology," *Version v0*, 2008.

[25] D. Agrawal and D. Kesdogan, "Measuring anonymity: The disclosure attack," *IEEE S&P*, 2003.

[26] D. Kesdogan and L. Pimenidis, "The hitting set attack on anonymity protocols." in *IH*, 2004.

[27] M. Edman and P. Syverson, "As-awareness in tor path selection," in *CCS*, 2009.

[28] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, "Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail," in *IEEE S&P*, 2012.

[29] T. Wang and I. Goldberg, "Improved website fingerprinting on tor," in *WPES*, 2013.

[30] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective attacks and provable defenses for website fingerprinting," in *USENIX Security*, 2014.

[31] "Bittorrent over tor isn't a good idea," https://blog.torproject.org/blog/bittorrent-over-tor-isnt-good-idea.

[32] S. L. Blond, P. Manils, C. Abdelberi, M. A. D. Kaafar, C. Castelluccia, A. Legout, and W. Dabbous, "One bad apple spoils the bunch: exploiting p2p applications to trace and profile tor users," *arXiv*, 2011.

[33] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, 1996.

[34] "Oblivious peer-to-peer protocol," https://github.com/jiayaoqijia/OblivP2P-Code.

[35] "Gnutella," https://en.wikipedia.org/wiki/Gnutella.

[36] Y. Jia, G. Bai, P. Saxena, and Z. Liang, "Anonymity in peer-assisted cdns: Inference attacks and mitigation," in *PETS*, 2016.

[37] "Scaneye's global bittorrent monitor," http://www.cogipas.com/anonymous-torrenting/torrent-monitoring/.

[38] K. Bauer, D. McCoy, D. Grunwald, and D. Sicker, "Bitstalker: Accurately and efficiently monitoring bittorrent traffic," in *WIFS*, 2009.

[39] T. Chothia, M. Cova, C. Novakovic, and C. G. Toro, "The unbearable lightness of monitoring: Direct monitoring in bittorrent," in *SECURECOMM*, 2012.

[40] G. Danezis and C. Diaz, "A survey of anonymous communication channels," Tech. Rep., 2008.

[41] H. Corrigan-Gibbs, D. Boneh, and D. Mazières, "Riposte: An anonymous messaging system handling millions of users," in *IEEE S&P*, 2015.

[42] M. Backes, A. Kate, S. Meiser, and E. Mohammadi, "(nothing else) mator(s): Monitoring the anonymity of tor's path selection," in *CCS*, 2014.

[43] "Tor suffers traffic confirmation attack," http://www.techtimes.com/articles/11711/20140802/tor-suffers-traffic-confirmation-attacks-say-goodbye-to-anonymity-on-the-web.htm .

[44] "Traffic confirmation attack," https://blog.torproject.org/blog/tor-security-advisory-relay-early-traffic-confirmation-attack.

[45] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website fingerprinting in onion routing based anonymization networks," in *WPES*, 2011.

[46] J. Kong, W. Cai, and L. Wang, "The evaluation of index poisoning in bittorrent," in *ICCSN*, 2010.

[47] K. El Defrawy, M. Gjoka, and A. Markopoulou, "Bottorrent: Misusing bittorrent to launch ddos attacks." *SRUTI*, 2007.

[48] "Software Guard Extensions Programming Reference." software.intel.com/sites/default/files/329298-001.pdf, 2013.

[49] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE S&P*, 2013.

[50] E. Shi, T.-H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost," in *ASIACRYPT*, 2011.

[51] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *CCS*, 2013.

[52] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants Count: Practical Improvements to Oblivious RAM ," in *USENIX Security*, 2014.

[53] S. Devadas, M. van Dijk, C. Fletcher, L. Ren, E. Shi, and D. Wichs, "Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM," *IACR*, 2015.

[54] T. Moataz, T. Mayberry, and E.-O. Blass, "Constant Communication ORAM with Small Blocksize," in *CCS*, 2015.

[55] D. Dachman-Soled, C. Liu, C. Papamanthou, E. Shi, and U. Vishkin, "Oblivious network RAM and leveraging parallelism to achieve obliviousness," in *ASIACRYPT*, 2015.

[56] E. Boyle, K. Chung, and R. Pass, "Oblivious parallel RAM and applications," in *TCC*, 2016.

[57] D. S. Roche, A. J. Aviv, and S. G. Choi, "A practical oblivious map data structure with secure deletion and history independence," *IACR*, 2015.

[58] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *FOCS*, 1995.

[59] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan, "Key homomorphic prfs and their applications," in *CRYPTO*, 2013.

[60] M. Naor and O. Reingold, "Number-theoretic constructions of efficient pseudo-random functions," in *FOCS*, 1997.

[61] "Cloc," http://cloc.sourceforge.net/.

[62] "Python cryptography toolkit," https://pypi.python.org/pypi/pycrypto.

[63] "Python ecc," https://github.com/johndoe31415/joeecc.

[64] "Deterlab," https://www.isi.deterlab.net/index.php3.

[65] A. R. Bharambe, C. Herley, and V. N. Padmanabhan, "Analyzing and improving bittorrent performance," *Microsoft Research*, 2005.

[66] G. Danezis, "Statistical disclosure attacks," in *Security and Privacy in the Age of Uncertainty*, 2003.

[67] G. Danezis, C. Diaz, and C. Troncoso, "Two-sided statistical disclosure attack," in *PETS*, 2007.

[68] G. Danezis and A. Serjantov, "Statistical disclosure or intersection attacks on anonymity systems," in *IH*, 2005.

[69] N. Mathewson and R. Dingledine, "Practical traffic analysis: Extending and resisting statistical disclosure," in *PETS*, 2005.

[70] N. Mallesh and M. Wright, "The reverse statistical disclosure attack," in *IH*, 2010.

[71] C. Troncoso, B. Gierlichs, B. Preneel, and I. Verbauwhede, "Perfect matching disclosure attacks," *LNCS*, 2008.

[72] G. Danezis and C. Troncoso, "Vida: How to use bayesian inference to de-anonymize persistent communications," in *PETS*, 2009.

[73] F. Pérez-González and C. Troncoso, "Understanding statistical disclosure: A least squares approach," in *PETS*, 2012.

[74] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *IEEE S&P*, 2010.

[75] S. Lu and R. Ostrovsky, "Distributed oblivious RAM for secure two-party computation," in *TCC*, 2013.

[76] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious distributed cloud data store," in *NDSS*, 2013.

[77] ——, "Multi-cloud oblivious storage," in *CCS*, 2013.

[78] B. Chen, H. Lin, and S. Tessaro, "Oblivious parallel RAM: improved efficiency and generic constructions," in *TCC*, 2016.