Performance and Security Trade-offs for Data Obliviousness

Towards efficient and secure set intersection

Alex Jacey Ligthart-Smith
Supervised by Dr Olga Ohrimenko
and Professor Anthony Wirth

Presented for a 25-point Research Project - COMP90055
Master of Information Technology (Computing)

School of Computing and Information Systems
The University of Melbourne
December 2020

Abstract

Data oblivious algorithms offer a strong guarantee of security against information leakage through external memory access-patterns. This security comes with a cost to efficiency, with many oblivious algorithms incurring $\Omega(\log n)$ access overheads, or large constant factors that make them impractical. We propose a relaxed definition of *output-constrained obliviousness*, suitable when the outputs of computation are to be revealed to an adversary. We apply this definition in the context of set intersection, with promising results under the condition of $O(\sqrt{n})$ private memory size. We also present novel set intersection algorithms that have the potential to provide further efficiency improvements with some small probability of privacy failure.

Acknowledgements

I acknowledge that my work has been undertaken on the stolen land of the Wurundjeri Woi Wurrung people of the Kulin nation, and that sovereignty has never been ceded.

I wish to thank my supervisors, Olya and Tony, for their patience and generosity with me throughout this difficult year. It has been a pleasure to work with them and to draw on their knowledge and wisdom.

Thank you to darcy and Joshua for your support and encouragement, and to Caspian, Hester, and Ronly Honly Bing for reminding me to take time to nap in sunbeams and to seek the comfort and warmth of loved ones.

I owe a debt of gratitude and solidarity to the retail, food service, warehouse, and freight/delivery workers whose work has allowed me to complete this thesis in relative comfort and safety at home during a pandemic.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Data is increasingly moving away from on-site servers onto external servers in the cloud. While this offers reduced financial costs and increased flexibility, it comes with increased costs in terms of data security and network latency. Encryption offers strong security against unauthorised parties viewing the contents of data, however there is some information that can still be discerned from the way the data is accessed. One point of information leakage is through the sequence of read and write operations requested on the data. For example, Islam *et al.* [1] present an attack on searchable encryption schemes that is able to identify 80% of search queries on an encrypted email database, from access-patterns alone.

Data obliviousness is a security guarantee developed to protect against such attacks. Data oblivious algorithms ensure that the sequence of accesses made on an input data set are computationally indistinguishable for all inputs of the same size. Oblivious algorithms are available for a variety of purposes, including sorting [2, 3, 4, 5] and shuffling [5, 6], as well as various oblivious RAM constructions (ORAM) [7, 8, 9, 10] that provide generic access-pattern privacy.

Data oblivious algorithms provide strong security guarantees, however these come with reduced efficiency in terms of the number of I/O operations performed on the external data, which is exacerbated by network latency. Classic sorting networks, that provide obliviousness through deterministic compare and swap sequences, have impractical overheads; Batcher's famous sorting network [2] has a cost of $O(n^2 \log n)$, while the $O(n \log n)$ AKS sorting network [3] has a constant factor that makes it unrealistic in practice. In fact, oblivious sorting, ORAMs, and some other oblivious algorithms including merging, have a lower bound access overhead of $\Omega(\log n)$ [7, 11, 12, 13]. Relaxed security guarantees, such as differentially private obliviousness [13, 14], provide a compromise between access overhead and access-pattern privacy. With Chan *et al.* [13] proving $O(n \log \log n)$ runtimes for sorting and merging under their definition.

We consider the case of an *output-constrained* obliviousness, whereby our adversary has access to the output of a computation. For example, when outputs are to be revealed in a publicly available report or article. When the output is revealed, there is some amount of information already available to the adversary that we no longer need to conceal. We apply this relaxed security definition particularly in the context of set intersection, whereby the size of the intersection and the values of the elements within it are revealed to the adversary. As such, any information within the access-patterns that reveals only intersection size and values need not be concealed. The goals of this project are to provide a precise definition of output-constrained (OC) obliviousness, and to develop efficient set intersection algorithms that fulfill this new security definition.

We draw on existing data oblivious algorithms, ORAMs, and relaxed notions of obliviousness to provide a definition of OC obliviousness that is consistent with the literature. We then use the definition, as well as common set intersection algorithms, and techniques from the literature, to develop novel OC-oblivious set intersection algorithms. These algorithms are able to compete with the efficiency of their non-oblivious counterparts when equipped with an $O(\sqrt{n})$ private memory. We also briefly explore the efficiency benefits of a common relaxation, δ-statistical obliviousness, that allows a small probability of privacy failure, and begin developing efficient set intersection algorithms under this definition.

2 Preliminaries

Data oblivious algorithms are defined under an external memory model: a private CPU with a small private memory (the "client") and a large external memory (the "server"). It is assumed that the server is an "honest-but-curious" adversary, and that all data is securely encrypted and decrypted by the client, so the only information visible to the server is the access-pattern.

2.1 External Memory Model

Data oblivious algorithms are carried out in an external memory environment, where a *client* reads and writes *data blocks* from/to a *server*:

- *Data Blocks* are fixed size units of data that can be stored, read, and written. Assume that the size of each block is sufficient to contain its location in a memory containing up to N blocks, i.e. the block size must be $\Omega(\log N)$. Going forward, size complexity is measured in terms of the number of data blocks.
- *The Client* can perform private computation and has a small private memory that persists between operations. The client stores encryption keys in private memory, and encrypts and decrypts data blocks privately using a *semantically secure* encryption scheme, so that same data block re-encrypted looks different from the point of view of the server [15]. Unless otherwise specified, assume that the client's private memory is of size $O(1)$.
- *The Server* is a large external memory. It does not perform computation, decryption, or encryption. The server can also be considered as the *honest-but-curious* adversary under this model. This server carries out all data access requests faithfully, but has an imperative to learn as much as possible about the client's data [16]. Since encryption prevents the server from learning anything about the contents of any data block, data blocks can be treated as opaque balls, and the data access-patterns as the only information available to the server. The server is of size $\Omega(N)$.

In this model, running algorithm $\mathcal{M}$ on input I produces some (deterministic) *output* (or result) $o \in O$ and a (possibly randomised) access-pattern $s \in \mathcal{S}$. While the access-pattern is a type of output, it must be distinguished from the intended output of the algorithm, which is more easily concealed from the server.

Definition 2.1 (Access-Pattern). *Let* op_j *be a read/write operation between private memory and external memory. When* $\mathrm{op}_i := \mathrm{read}(\mathrm{a}_i)$*, the data block at location* a_i *of the external memory is read to private memory. When* $\mathrm{op}_i := \mathrm{write}(\mathrm{a}_i, \mathrm{data}_i)$ *data block* data_i *is copied from private memory to external memory location* a_i*. An access-pattern* $\mathcal{A} := (\mathrm{op}_0, \mathrm{op}_1, ..., \mathrm{op}_{n-1})$ *is a sequence of* n *read/write operations.*

The primary efficiency concern for external memory algorithms is the I/O complexity of algorithms: that is, the number of requests sent to the server, rather than the number of computation steps performed in private memory [6]. This is because communication between the client and server is affected by network latency.

2.2 Data Oblivious Algorithms

Data oblivious algorithms are designed to prevent an adversary from learning information from access-patterns, by ensuring that the sequence of memory accesses is independent of the input data [17, 13].

Definition 2.2 (Data Oblivious Algorithm [17])**.** *An algorithm $\mathcal{M}$, that produces some access-pattern $A^{\mathcal{M}}(I) \in \mathcal{S}$ is data oblivious if*

$$\Pr[A^{\mathcal{M}}(I_1) \in S] = \Pr[A^{\mathcal{M}}(I_2) \in S] \tag{1}$$

for all inputs $|I_1| = |I_2|$, where $S \in \mathcal{S}$ is a subset of all possible memory access-patterns produced by the algorithm.

That is, for any two inputs of the same size, the probability distribution of access-patterns is the same, hence an adversary cannot distinguish the inputs based on access-patterns alone.

A slightly relaxed definition of obliviousness allows some small probability of privacy failure. A statistical difference of up to δ between the distributions of access-patterns is allowed for any two inputs of the same size.

Definition 2.3 (δ-Statistically Oblivious Algorithm [13, 18])**.** *An algorithm $\mathcal{M}$, that produces some access-pattern $A^{\mathcal{M}}(I) \in \mathcal{S}$ is δ-Statistically data oblivious if*

$$\Pr[A^{\mathcal{M}}(I_1) \in S] \leq \Pr[A^{\mathcal{M}}(I_2) \in S] + \delta \tag{2}$$

for all inputs $|I_1| = |I_2|$, where $S \in \mathcal{S}$ is a subset of all possible memory access-patterns produced by the algorithm.

Throughout this document, the phrase *full obliviousness* is used to describe algorithms meeting the requirements of Definition 2.2. *δ-obliviousness*, as in Definition 2.3, is not addressed until Section 5.4.

3 Background and Related Work

In order to achieve our goal of producing efficient and oblivious set intersection algorithms, there are several techniques that we either use directly or draw inspiration from. In this section we describe some algorithms that we employ as part of our set intersection algorithms, oblivious sorting and oblivious shuffle; provide some context for the inspiration of our techniques from Oblivious RAMs and Private Record Linkage; and offer for comparison another relaxation on obliviousness in Differentially Private Oblivious Algorithms.

3.1 Oblivious Sorting

Some of the first oblivious algorithms were sorting networks. Batcher introduced an odd-even merge sorting network in 1968 [2], this was designed as a hardware switching network to parallelise sorting however also functions as a deterministic oblivious sorting algorithm. The sorting network compares elements in a predefined sequence, with a total of $\Theta(n \log^2 n)$ comparisons required to sort n items. Since the sequence of comparisons is predefined, every two inputs of the same size follow the exact same sequence of comparisons, making this algorithm deterministically oblivious. Non-oblivious comparison-based sorting algorithms can be performed in $\Theta(n \log n)$ comparisons, so this network has a logarithmic access overhead.

The AKS sorting network [3] overcomes this overhead, as an $\Theta(n \log n)$ comparison network, however the original algorithm has a constant factor overhead of 2^{99}, with the most efficient modification obtaining a constant factor of 1.36×10^7 [19, 20]. These constant factors make this algorithm unfeasible to implement in practice. The number of comparisons in a deterministic sorting network is further improved by Goodrich's Zig-Zag sort [20], which requires $1.96 \times 10^4 \times n \log n$ comparisons. Unlike Batcher's and the AKS sorting network, Zig-Zag sort is not parallelisable.

Allowing for a larger private memory, and some probability of failure, Bucket Oblivious Sort [5] provides a more realistic $O(n \log n)$ oblivious sort. This algorithm runs with a constant factor of just 6, with a failure probability of $e^{-Z/6}$, where Z is the number of data blocks that can be stored in private memory. So with $Z = \Omega(\log N)$ the failure probability is negligible in N.

An alternative oblivious sorting method involves first performing an oblivious random permutation (*shuffle*) of the data, and then sorting using a non-oblivious comparison-based method [10]. This is promising, but most known oblivious shuffle algorithms rely on an intermediate oblivious sort, so in those cases this cannot offer any improvement.

3.2 Oblivious Shuffle

Given an input I containing N data blocks, a random permutation re-orders the blocks, selecting from the $N!$ possible orderings uniformly at random; a pseudo-random permutation is one that is indistinguishable from a random permutation by any *efficient*[1] adversary [21]. An *oblivious shuffle* is an algorithm that performs a random permutation of its input without revealing any correlation between the original and final locations of the data blocks [6].

[1] running in polynomial time

An oblivious shuffle of the input data can be a useful step in a number of other oblivious algorithms, such as the sorting example above. Assuming that each data block is identified by some unique, ordinal key x from the space X, then a random permutation of data blocks can be performed by choosing a a bijection $F : X \to X$ uniformly at random, *tagging* each element of I with a new random key $F(x)$, and sorting the data based on these tags. As such, traditional oblivious shuffle algorithms are performed by obliviously sorting the data based on the values of random tags [6]. The performance of these algorithms depends on that of the underlying oblivious sort.

The Melbourne Shuffle [6] overcomes this limitation, providing an oblivious shuffle that does not depend on an oblivious sorting algorithm. This algorithm can shuffle N data blocks with just $O(\sqrt{N})$ I/O operations and $O(N)$ client computation steps, but relies on a private memory of size $O(\sqrt{N})$ and a message size of $O(\sqrt{N})$, and has a negligible probability of failure.

3.3 Oblivious RAM (ORAM)

Oblivious RAM, introduced by Goldreich and Ostrovsky [7, 8], is a generalised definition of data obliviousness where the input is a requested access-pattern, and the actual sequence of accesses performed must fulfil the requirements of obliviousness given in Section 2.2. That is, for any sequence of operations requested, an oblivious RAM accurately carries out the request while performing an access sequence that is computationally indistinguishable from that for any requested sequence of the same length. Unfortunately, there is a $\Omega(\log N)$ lower bound on blowup in the number of data access operations for any oblivious RAM with N data blocks [7, 11].

Path ORAM [9] is a near-optimal ORAM protocol that employs a binary-tree structure combined with a private memory of size $O(\log N) \cdot \omega(1)$, and performs well in practice. Under this protocol, access overheads are limited by relying on a larger bandwidth to fetch a full *path* of the binary-tree at each access. The binary-tree is made up of buckets, with each data block mapped to a random leaf node. Write operations are *stashed*, and when a path is fetched for a read operation any stashed write blocks are inserted into the fetched path in the deepest bucket that is on the path to their leaf node, if there is sufficient space. The idea of randomly mapping data blocks onto leaf nodes of a binary-tree in order to achieve obliviousness is revisited in Section 5.4.

The $\Omega(\log(N))$ lower bound for ORAM can be extended more generally to a variety of oblivious algorithms [13]. These overheads can make *full* obliviousness impractical for large-scale applications.

3.4 Differentially Private (DP) Obliviousness

In order to reduce overhead while still providing access-pattern privacy, Chan *et al.* [13] and Wagh *et al.* [14] have developed definitions of *differentially private* (DP) obliviousness. DP-obliviousness draws on the concept of differential privacy (DP), introduced by Dwork *et al.* [22], generalising the definitions of DP for computed statistics to consider data access-patterns themselves as a statistic to be kept differentially private.

Definition 3.1 (Differentially Private Oblivious Algorithm). *For neighbouring inputs I and I' (differing by only one entry), Chan et al. [13] define (ϵ, δ)-Differential obliviousness as*

$$\Pr[A^{\mathcal{M}}(I) \in S] \leq e^{\epsilon} \cdot \Pr[A^{\mathcal{M}}(I') \in S] + \delta \tag{3}$$

where e^{ϵ} is an allowed difference factor in the distributions of access-patterns, and δ is some small probability of failure.

This relaxes the requirements of full obliviousness both in the allowed statistical difference between distributions, and the limitation to neighbouring databases. These relaxations are shown to reduce the overhead for sorting and merging algorithms to $O(\log \log N)$ [13].

3.5 Private Record Linkage

The goal of Private record linkage (PRL) is to identify matching pairs of records between private databases held by two (or more) parties, in such a way that the privacy of the source databases is maintained. This is similar to oblivious set intersection in that it is concerned with privately matching records, however it differs in the nature of the adversary. While we are interested in concealing the access-patterns made on a single server holding two input sets, from the server itself, PRL can be considered as two separate servers concealing their contents from one another, revealing only the size of their data sets and the contents of the intersection [23].

PRL is usually achieved using secure two-party computation (S2PC), however these solutions have either low recall or do not meet efficiency requirements. He *et al.* [23] propose a novel security definition, *Output Constrained Differential Privacy*, which reveals the output of the computation performed on two private databases, while releasing only differentially private statistics about any data not revealed in the output. In the context of PRL, this composes differential privacy and S2PC to provide strong guarantees for both efficiency and recall, while maintaining DP for all elements of each database not in the intersection.

Our definition of *output constrained* obliviousness is inspired, in part, by this work.

4 Output-Constrained Obliviousness

As in the case of PRL, where the two parties must reveal the matching records to one another, there are circumstances where the server has access to the output of some computation. For example, when the output is to be released publicly in a report or article. Since not all information about the data can be extrapolated from the released outputs, there is still value in maintaining access-pattern privacy. In this situation, though, full obliviousness also conceals information that can be deduced from the outputs. We propose a new definition, *output-constrained* (OC) obliviousness, with the goal of reducing the computational overhead of data obliviousness by taking into account the adversary's knowledge of the output.

4.1 Definitions

We first give a definition of output neighbors (*o-neighbors*), any two inputs of the same size whose output is the same. This is definition is inspired by a definition of *r-neighbors* given by Chan *et al.* [13] for their analysis of DP-oblivious algorithms.

Definition 4.1 (o-Neighbors)**.** *Let $\mathcal{M} : \mathcal{I} \to \mathcal{O}$ be an algorithm with input database $I \in \mathcal{I}$, that produces a deterministic output $o \in \mathcal{O}$. Input databases I and I' are o-Neighbors, denoted $(I, I') \in \mathcal{N}^o(\mathcal{M})$ if $\mathcal{M}(I) = \mathcal{M}(I')$. That is, if the algorithm produces the same output when the input is either of the databases.*

While an algorithm's *output* must be deterministic to fit this definition, its *access-patterns* can be randomised. An algorithm is OC-oblivious if the distributions of access-patterns are equal for o-neighboring inputs.

Definition 4.2 (OC-Obliviousness)**.** *Let $\mathcal{M} : \mathcal{I} \to \mathcal{S}$ be a (possibly randomised) algorithm with input database $I \in \mathcal{I}$, that produces an access-pattern $s \in \mathcal{S}$ and a deterministic output $o \in \mathcal{O}$. $\mathcal{M}$ is output-constrained data oblivious (OC-oblivious) if for every two o-neighboring inputs $(I, I') \in \mathcal{N}(\mathcal{A})$, and every subset of possible memory access-patterns $S \subseteq \mathcal{S}$, we have:*

$$\Pr[\mathcal{A}^{\mathcal{M}}(I) \in S] = \Pr[\mathcal{A}^{\mathcal{M}}(I') \in S]. \tag{4}$$

That is, access-patterns are independent of input within every set $\mathcal{N}^o(\mathcal{M})$ of inputs that produce the same output.

4.2 Example: OC-Oblivious Sorting

As an exercise, let us consider the case of sorting. OC-Oblivious sorting can be used if the sorted values are to be released publicly, but their original entry order into the data set must be concealed. Let $\mathcal{A} : \mathcal{I} \to (\mathcal{S}, \mathcal{O})$ be a sorting algorithm with input database $I \in \mathcal{I}$, that produces some access-pattern $s \in \mathcal{S}$ and output $o \in \mathcal{O}$. $\mathcal{A}$ is OC-oblivious if for every permutation, I', of I Equation 4 is satisfied. That is, the access-patterns appear independent of the order in which their entries appear in the database.

As mentioned in Section 3.1, an alternative to sorting networks for oblivious sorting is to first obliviously shuffle the data and then sort using a non-oblivious comparison-based method. Known oblivious permutations that do not rely on sorting networks [6, 5] require some additional client storage, as well as incurring some small probability of error. In the case where data blocks are to be sorted based on some key that can be represented with w bits, and allowing $O(\sqrt{N})$ client memory and a negligible error, we can perform OC-Oblivious sorting by first obliviously permuting the data using the Melbourne Shuffle [6] then sort using a non-comparison-based sort, such as radix sort. This gives us an $O(w \cdot N)$ sorting algorithm, an improvement on the known $\Omega(N \log N)$ lower bound for fully oblivious sorting when $w \in o(\log N)$.

4.3 Comparison of Obliviousness Definitions

One way to compare definitions of full and relaxed obliviousness is to compare the sizes of the sets that must share access-pattern distributions. Under full obliviousness, all inputs of the same size must map onto the same distribution of access-patterns, whereas under OC-obliviousness we only require that of inputs that are both the same size and have the same deterministic output. Figure 1 shows the difference between the set of inputs that must produce the same distributions of access-patterns under the two definitions. DP-obliviousness can be considered a further relaxation on oblivious not only because of the allowed ϵ and δ errors, but also because access-pattern distributions must only be similar for inputs that differ by at most one entry.

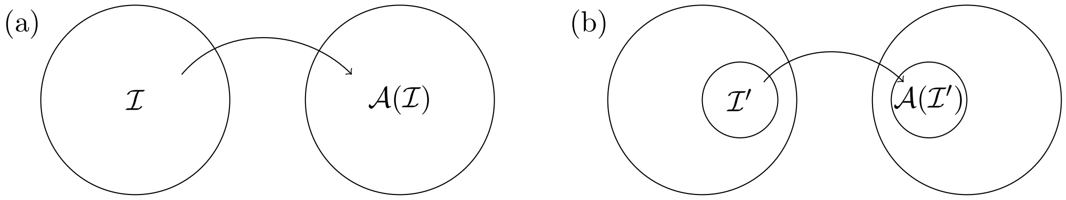

Figure 1: Difference in domains between full obliviousness and OC-obliviousness. In (a) we see the domain of all possible inputs to an algorithm, $\mathcal{I}$ mapping to all possible access-patterns, $\mathcal{A}(\mathcal{I})$. In (b) we see the sub-domain $\mathcal{I}'$ of some specific o-neighboring set $\mathcal{N}^o$, mapping to some subset of access-patterns $\mathcal{A}(\mathcal{I}')$.

Table 1 shows the size of neighbour sets under different definitions of obliviousness, where the a 'neighbour' under full obliviousness is every input of the same size (in terms of the number of data blocks). Here we have that n is the full size of the input and k is the size of the key-space $\mathcal{K}$ that comparisons are being made on; m_1 and m_2 are the sizes of two input arrays used in set intersection and merging and m is the size of the set intersection; h is the number of equal-sized ranges in a frequency histogram and s_i is the size of bar i of the output histogram. Assume $k \geq n$. All algorithms are disallowing duplicates for simplicity of calculating permutations.

We can see that in general the size of the neighbor sets for DP-obliviousness is smallest, and for full obliviousness is largest, with the differences between OC-obliviousness and full obliviousness depending strongly on the nature of the algorithm. This indicates that we can expect OC-obliviousness to provide stronger performance improvements for some classes of algorithms over others.

Table 1: The number of neighbours of a given input set I for various algorithmic classes and neighbour definitions. Here m_1 and m_2 are input set sizes, with $m_1 \leq m_2$, m is the size of the intersection, $n = m_1 + m_2$, and $k \geq n$ is the size of the key-space data block identifiers are drawn from.

	DP-Oblivious	OC-Oblivious	Fully Oblivious
Sorting	$n(k-n+1)$	$n!$	$n!\binom{k}{n}$
Set Intersection	$m_1(k-m_1+1)\cdot$ $m_2(k-m_2+1)$	$m_1!\binom{k}{m_1-m}\cdot$ $m_2!\binom{k}{m_2-m}$	$m_1!\binom{k}{m_1}\cdot m_2!\binom{k}{m_2}$
Merging (pre-sorted)	$m_1 m_2 (k-n+1)^2$	$\binom{n}{m_1}$	$\binom{k}{n}\binom{n}{m_1}$
Key search (pre-sorted)	$n(k-n+1)$	$\binom{k-1}{n-1}$ present, $\binom{k-1}{n}$ absent	$\binom{k}{n}$
Frequency histogram (unsorted)	$n(k-n+1)$	$n!\prod_{i=1}^{h}\binom{\frac{k}{h}}{s_i}$	$n!\binom{k}{n}$

5 Set Intersection Algorithms

We now move to the description and evaluation of set intersection algorithms, first describing some non-oblivious and fully oblivious set intersection algorithms as a baseline, and moving on to novel algorithms with relaxed security guarantees. Here we assume that a *set* is stored in memory as an *unsorted* list, with no duplicate elements, and that a set intersection algorithm, with inputs S_1 and S_2 returns the set $S_1 \cap S_2 = S$ as an unsorted list. Each element of each set is a single data block, and elements can be sorted and matched based on some ordinal key. We have $|S_1| = m_1$, $|S_2| = m_2$, and $|S| = m$; without loss of generality, we assume that $m_1 \leq m_2$. We must modify our definitions of oblivious (Definitions 2.2 and 2.3) slightly to accommodate the two input sets. Not only is our obliviousness only guaranteed for inputs (I, I') of the same size $|I| = |I'|$, we must also have that corresponding sets within the inputs are the same size, that is, $m_1 = m_1'$ and $m_2 = m_2'$.

Table 2 compares the asymptotic time complexity of the set intersection algorithms presented throughout this section. Note that the δ-oblivious algorithms perform asymptotically equivalent or close to their non-oblivious counterparts, while OC-oblivious algorithm performance depends on the size of private memory, due to the complexity of the underlying shuffle algorithms.

Table 2: Comparison of asymptotic set intersection algorithm performance. Here m_1 and m_2 are input set sizes, with $m_1 \leq m_2$, and $n = m_1 + m_2$.

Algorithm	I/Os $O(1)$ private memory	I/Os $O(\sqrt{n})$ private memory
1: Non-oblivious merging	$O(m_2 \log m_2)$	
2: Non-oblivious binary-search	$O(m_2 \log m_1)$	
3: Non-oblivious hashing	$O(m_2)$ exp.	
4: Fully oblivious merging	$O(m_2 \log m_2)$	
5: OC-oblivious binary-search	$O(m_2 \log m_2)$	$O(m_2 \log m_1)$
6: OC-oblivious hashing	$O(m_2 \log m_2)$ exp.	$O(m_2)$ exp.
7: δ-oblivious binary-search	$O(m_2 \log m_1)$	
8: δ-oblivious hashing	$O(m_1 \log m_1 + m_2)$ exp.	$O(m_2)$ exp.

5.1 Non-oblivious Set Intersection

A standard external memory set intersection algorithm, Algorithm 1 sorts each list, and then performs a merge-like iteration through them to find matching elements. This algorithm performs in $O(m_1 \log m_1) + O(m_2 \log m_2) + O(m_1 + m_2) = O(m_2 \log m_2)$ I/Os, provided that sorting comprises an $O(n \log n)$ comparison-based sorting algorithm.

In addition to the information revealed during the non-oblivious sorting of the sets, this algorithm reveals the size of the intersection. Depending on the amount of location information revealed during sorting, concrete or probabilistic information about which items from each set are in the intersection are also revealed. For example, the speed at which the algorithm iterates through each of the lists trivially reveals the relative order of items in the two sets.

If the two sets are of significantly different sizes the efficiency of set intersection can

Algorithm 1: Basic merge-based set intersection

```
Input: Sets $S_1$ and $S_2$ stored in external memory
Output: Set $S = S_1 \cap S_2$, written to external memory
initialise $S$;
sort($S_1$); sort($S_2$);
$i, j \leftarrow 0$;
while $i < m_1$ and $j < m_2$ do
    if $S_1[i] = S_2[j]$ then
        append($S, S_1[i]$);          ▷ add the element to the output set
        $i \leftarrow i + 1$;
        $j \leftarrow j + 1$;
    else if $S_1[i] < S_2[j]$ then
        $i \leftarrow i + 1$;
    else
        $j \leftarrow j + 1$;
return $S$;
```

be improved by only sorting the smaller of the two sets. Algorithm 2 finds the set intersection $S_1 \cap S_2$ by first sorting the smaller set S_1 and then searching it for each element of the larger set S_2 via binary-search. Using a comparison-based sort, this performs in $O(m_1 \log m_1) + O(m_2 \log m_1) = O(m_2 \log m_1)$ I/Os, an improvement on the performance of Algorithm 1 when $m_1 < n^c$ for all $c > 0$.

Algorithm 2: Set intersection using sorting and binary-search[2]

```
Input: Sets $S_1$ and $S_2$ stored in external memory, with $|S_1| \leq |S_2|$
Output: Set $S = S_1 \cap S_2$, written to external memory
initialise $S$;                       ▷ New set to contain intersection
sort($S_1$);
for $s \in S_2$ do
    if $s \in S_1$ then                          ▷ Binary-search
        append($S, s$);                        ▷ Add record to $S$
return $S$;
```

This algorithm again reveals information through sorting, and during the binary-search phase reveals which items from S_2 are in the intersection, as its original order is maintained in the outer loop.

Avoiding sorting altogether, expected performance for set intersection can be improved by instead searching in a hash table. Algorithm 3 finds the set intersection by first hashing S_2 to a table H and then searching H for the elements of the smaller set S_1. Since S_2 is guaranteed to contain no duplicates, each insertion can be performed in $O(1)$ (guaranteed or expected, depending on implementation) I/Os, giving a total expected $O(m_2)$ I/Os to build the hash table. Assuming simple uniform hashing in a table larger than $|S_2|$, search requires $O(1)$ expected I/Os. Thus the algorithm performs in $O(m_2) + O(m_1) = O(m_2)$ expected I/Os, and requires $\Omega(m_2)$ additional external memory.

[2] "Maria's Algorithm"

Algorithm 3: Set intersection using a hash table

```
Input: Sets $S_1$ and $S_2$ stored in external memory, with $|S_1| \leq |S_2|$
Output: Set $S = S_1 \cap S_2$, written to external memory.
initialise $S$;                          ▷ New set to contain intersection
$H \leftarrow$ hash($S_2$);
for $s \in S_1$ do
    if $s \in H(s)$ then
        append($S, s$);                  ▷ Add record to $S$
return $S$;
```

Through use of an appropriate hash function, this algorithm obscures the relative order of items in the sets. The adversary can, however, learn exactly which elements from both sets are in the intersection by closely observing the pattern of hash table insertions and searches, and writes to S.

These algorithms provide a benchmark for the efficiency of set intersection on unsorted input data.

5.2 Fully Oblivious Set Intersection

Algorithm 4: Merge-based oblivious set intersection

```
Input: Sets $S_1$ and $S_2$ stored in external memory
Output: Set $S$ written to external memory. $|S| = n + 1$. $S[0]$ is a size variable
        $m$, $S[1..m]$ contains $S_1 \cap S_2$, and $S[m+1..n]$ contains dummy variables.
$m \leftarrow 0$;
$S \leftarrow$ append($[0], S_1, S_2$);
obliviously sort $S$;
for $i \in [1, |S| - 1)$ do
    $a \leftarrow$ read($S[i]$);            ▷ read to private memory and decrypt
    $b \leftarrow$ read($S[i+1]$);
    if $a = b$ then
        write($S[i], a$);                 ▷ encrypt and write to external memory
        $m \leftarrow m + 1$;
    else
        write($S[i]$, dummy);
write($S[0], m$) ;           ▷ to an appropriate location in external memory
obliviously sort $S[1..n]$ ; ▷ s.t. dummy variables appear at the end of $S$
return $S$;
```

A fully oblivious set intersection can be built from a merge-based approach, like that in Algorithm 1. Arasu and Kaushik [24] offer this approach, specifically in the context of relational database joins. In Algorithm 4 we offer a simplified version of their algorithm, with generalised syntax. It involves first appending the two sets together into a list S, of size n, before obliviously sorting S and scanning it for duplicates. When a pair of adjacent elements are not duplicates, the first element of the pair is replaced with a dummy element to prevent the adversary from learning how many elements are in the intersection, and their relative positions in the union of the sets. Finally, S must be obliviously sorted again such that the dummy elements appear at the end of the list. The I/O complexity of this algorithm is dominated by the oblivious sort, $O(n \log n)$ [16]. In fact, there is an $\Omega(n \log n)$ lower bound on fully oblivious merging algorithms [13, 12]. Overall, Algorithm 4 requires constant private memory (sufficient to hold two records), and $O(n)$ additional external memory.

This algorithm is trivially oblivious as, when a deterministic sorting network is used for the sorting phase, the access-patterns are deterministic and depend only on the sizes of the two sets.

5.3 Output-Constrained Oblivious Set Intersection

5.3.1 Binary-search based algorithm

While Algorithm 4 provides full data obliviousness, it does so at a cost in I/O operations relative to the non-oblivious implementations. The security guaranteed is stronger than necessary in the case where the intersection itself is to be revealed. In particular, revealing the size of the intersection allows us to take advantage of any significant difference in the sizes of S_1 and S_2.

In Algorithm 5 we offer a binary-search-based approach to an OC-Oblivious set intersection. Inputs S_1 and S_2 are first each obliviously shuffled and Algorithm 2 is run on random tags applied during the shuffle (see Section 3.2). The performance of this algorithm depends on that of the underlying shuffle. If an oblivious-sort-based shuffle is used, then we have $O(m_1 \log m_1) + O(m_2 \log m_2) + O(m_2 \log m_1) = O(m_2 \log m_2)$ I/Os and $O(m)$ additional external memory. If we use the Melbourne Shuffle this is reduced to $O(m_1) + O(m_2) + O(m_2 \log m_1) = O(m_2 \log m_1)$ I/Os, matching the asymptotic performance of its non-oblivious counterpart, but we require up to $O(\sqrt{m_2})$ private memory.

```
Algorithm 5: OC-Oblivious Set intersection (binary-search)
  Input: Sets $S_1$ and $S_2$ stored in external memory, with $|S_1| \leq |S_2|$
  Output: Set $S = S_1 \cap S_2$, written to external memory.
  obliviously shuffle $S_1$;                          ▷ applying random tags
  obliviously shuffle $S_2$;
  Call Algorithm 2, binary-searching in $S_1$ for each element in $S_2$, matching
    random tags;
```

To prove that this algorithm is OC-oblivious we must first define an ideal oblivious shuffle function *o-shuffle*.

Definition 5.1 (O-Shuffle). *O-shuffle is a fully oblivious algorithm, meeting the requirements of Definition 2.2. It performs a random permutation on an input I, of size $|I| = n$, by calling on a random oracle to compute random tags for each element. O-shuffle writes to memory the set I^p, containing the elements of I with their tags attached, sorted according to the tag values. The random oracle ensures that the elements of I^p are ordered according to each possible permutation of I with equal probability $\frac{1}{n!}$.*

Theorem 5.1. *Algorithm 5 is OC-Oblivious, according to Definition 4.2, when o-neighboring inputs (I, I') have $m_1 = m'_1$ and $m_2 = m'_2$, and where we have access to an ideal oblivious shuffle algorithm o-shuffle.*

Proof. Let us first assume that we have access to some ideal oblivious shuffle algorithm, *o-shuffle*. As such, the first and second steps of Algorithm 5 are, by definition, oblivious.

Now, we have some input $I = (S_1, S_2)$ with $S_1 \cap S_2 = S$, where $|S_1| = m_1, |S_2| = m_2$, and $|S| = m$. After performing o-shuffle on each set, giving us S_1^p and S_2^p, the locations of the elements of S within the list representations of S_1 and S_2 must now be uniformly randomly distributed, according to Definition 5.1. Specifically, there are $\binom{m_1}{m}$ ways that the set S can be distributed throughout S_1^p, each occurring with equal probability, and similarly $\binom{m_2}{m}$ for S_2^p. The permutations of the elements of S within those distributions must also be uniformly distributed, so we have $\binom{m_1}{m} \cdot m!$ equally likely ways of mapping the elements of S into the list S_1^p, and $\binom{m_2}{m} \cdot m!$ for S_2^p.

Moving on to the binary-search step, assuming we bypass the sorting step of Algorithm 2, since we already have our sets sorted according to their random tags, we must consider the access-patterns produced by searching for each element of S_2^p within S_1^p. Given that S_1^p and S_2^p are sorted according to random tags, the probability distribution of access-patterns produced by performing these searches depends only on the values m, m_1, and m_2. So for any input I, and every subset, B, of all possible binary-search access-patterns on inputs with the same dimensions (m_1, m_2, m), we have $\Pr[\mathcal{A}(I) \in B] = f(m_1, m_2, m, B)$. Therefore, if we have two o-neighboring inputs I and I' with $m_1 = m_1'$, $m_2 = m_2'$ and, by the o-neighbor definition, $m = m'$, then for every such subset, B, we have

$$\Pr[\mathcal{A}(I) \in B] = f(m_1, m_2, m, B) = f(m_1', m_2', m', B) = \Pr[\mathcal{A}(I') \in B].$$

Since the shuffle phase is fully oblivious by definition, Algorithm 5 is therefore OC-oblivious by Definition 4.2. ■

Table 3: Comparison of approximate I/O operations for merge-based oblivious set intersection, Algorithm 4, and binary-search based OC-oblivious set intersection, Algorithm 5. Here m_1 and m_2 are input set sizes, with $m_1 \leq m_2$, m is the size of the intersection, and $n = m_1 + m_2$.

	Private memory	I/Os
Fully Oblivious Alg. 4 (AKS sorting network [3])	$O(1)$	$1.08 \times 10^8 \times n \log n + 4n$
Fully Oblivious Alg. 4 (Bucket Oblivious Sort [5])	$O(\sqrt{n})$	$12n \log n + 2\sqrt{n}$
OC-Oblivious Alg. 5 (AKS Sorting Network [3])	$O(1)$	$2n + 5.4 \times 10^7 \times m_1 \log m_1$ $+5.4 \times 10^7 \times m_2 \log m_2$ $+m_2 \log m_1 + m$
OC-Oblivious Alg. 5 (Melbourne Shuffle [6])	$O(\sqrt{n})$	$2m_2 \log m_1 + 14(\sqrt{m_1} + \sqrt{m_2})$ $+\frac{n}{m_1} + \frac{2m_1}{\sqrt{n}}$

It is important to note here that only in the case where $m_1 \in o(m_2^c)$ for all $c \in \mathbb{R}^+$ is $O(m_2 \log m_1) \in o(m_2 \log m_2)$, and in this case, we could perform even a fully oblivious set intersection with $O(\sqrt{m_2})$ I/Os given $O(\sqrt{m_2})$ private memory. Hence, our OC-Oblivious algorithm offers no asymptotic performance improvement over a fully oblivious set intersection. With this in mind, Table 3 compares the actual I/Os performed by these algorithms. Comparison of the fully oblivious set intersection using oblivious bucket sort [5] with private memory $O(\sqrt{n})$, and OC-Oblivious intersection using the Melbourne Shuffle [6], gives a performance increase of approximately a factor of 12 when $m_1 \in [\sqrt{n}, \frac{n}{2}]$.

5.3.2 Hash based algorithm

In Algorithm 5 we offer a hash based approach to an OC-Oblivious set intersection. Sets S_1 and S_2 are first obliviously permuted, and then Algorithm 3 is performed based on random tags applied during the permutation step. The performance of this algorithm depends on that of the underlying shuffle. If an oblivious sorting based shuffle is used, then we have $O(m_1 \log m_1) + O(m_2 \log m_2) + O(m_1) + O(m_2) = O(m_2 \log m_2)$ expected I/Os, which again does not provide and asymptotic improvement. However, via Melbourne Shuffle we have $O(m_2)$ expected performance. This approaches the performance of its non-oblivious counterpart, and is asymptotically better (in expectation) than Algorithm 4 even when $m_1 \in \Omega(m_2^c)$.

Algorithm 6: OC-Oblivious Set intersection (hashing)

```
Input: Sets $S_1$ and $S_2$ stored in external memory, with $|S_1| \leq |S_2|$
Output: Set $S = S_1 \cap S_2$, written to external memory.
obliviously shuffle $S_1$;                              ▷ applying random tags
obliviously shuffle $S_2$;
Call Algorithm 3, hashing each element of $S_2$ to a table $H$, and searching for
  each element of $S_1$ in $H$, matching random tags;
```

Theorem 5.2. *Algorithm 6 is OC-Oblivious, according to Definition 4.2, when o-neighboring inputs (I, I') have $m_1 = m_1'$ and $m_2 = m_2'$, and where we have access to an ideal oblivious shuffle algorithm o-shuffle, and a truly random hash function.*

Proof. As in the proof of Theorem 5.1, we can assume that the shuffle step of this algorithm is fully oblivious. We now show that, given the o-shuffled sets S_1^p and S_2^p, the call to Algorithm 3 is OC-oblivious.

We assume that in place of a hash function we have a random oracle that maps our input space onto our hash table. Given that we have a truly random hash function, each element of S_2^p is hashed to each bucket of H with equal probability, so we have $m_2 \cdot |H|$ equally likely hash tables. When searching for the elements of S_1^p in the hash table, elements not in S are also hashed to each bucket of H with equal probability, $\frac{1}{|H| \cdot (m_1 - m)}$. Finally, since S_1^p and S_2^p are sorted according to their random tags, there are $\binom{m_1}{m}$ and $\binom{m_2}{m}$ ways the elements of S can be distributed in the respective input sets.

For every input I, and every subset, B, of all possibly binary-search access-patterns on inputs with the same dimensions (m_1, m_2, m), we have

$$\Pr[\mathcal{A}(I) \in B] = \frac{1}{(|H|)^2 \cdot m_2 \cdot (m_1 - m) \cdot \binom{m_1}{m} \cdot \binom{m_2}{m}} \cdot |B|.$$

Since $|H|$ depends on m_2 this gives us $\Pr[\mathcal{A}(I) \in B] = f(m_1, m_2, m, |B|)$. Therefore, if we have two o-neighboring inputs I and I' with $m_1 = m_1'$, $m_2 = m_2'$ and, by the o-neighbor definition, $m = m'$, then for every subset, B, of all possibly binary-search access-patterns on inputs with the same dimensions, we have

$$\Pr[\mathcal{A}(I) \in B] = f(m_1, m_2, m, |B|) = f(m_1', m_2', m', |B|) = \Pr[\mathcal{A}(I') \in B].$$

Since the shuffle phase is fully oblivious by definition, Algorithm 6 is therefore OC-oblivious by Definition 4.2. ■

5.4 δ-statistically Oblivious Set Intersection

An alternative relaxation on obliviousness is δ-statistical obliviousness, as described in Definition 2.3. This allows some small difference in the distribution of access-patterns between two inputs of the same size, or, some small probability of privacy failure. We have developed the following algorithms with this in mind, however it is important to note that calculating their δ values, and thus proving their level of obliviousness, was outside the scope of this small project. They are presented as a demonstration of novel techniques to be explored further in future work.

The security guarantees of Algorithms 5 and 6 can be improved by concealing the size of the intersection in some way. For example, by *marking* the items in the original sets when they are in the intersection, and later sorting based on these marks, or by outputting some dummy block for unsuccessful searches. These methods do not provide full obliviousness because of the access-patterns during the searches, but we predict they do provide some acceptable probability of privacy failure. The following algorithms improve on their efficiency further by obliviously shuffling only S_1, and relying on the randomness of random tags, and the fact that $|S_2|$ is larger than $|S_1|$, with many elements not in S, to obscure most of the information that is revealed by the non-oblivious algorithms.

5.4.1 Binary-search based algorithm

Algorithm 7 relies on binary-search. Like Algorithm 5 this algorithm depends on an oblivious shuffle, however more like the non-oblivious binary-search algorithm, Algorithm 2, it is only the smaller of the two sets that is shuffled. The outcome of the oblivious shuffle is that S_1 is sorted according to random tags. A random tag is calculated for each item in S_2 based on the same function used during the shuffle, and we search for these in shuffled S_1. To conceal which items in S_2 are contained in the intersection, each binary-search is continued to the same depth, using random coin-flips if necessary. If we model the binary-search process as a binary-tree, we can imagine that when a match is found (at a non-leaf node) we continue down the tree randomly until we reach a leaf node, as shown in Figure 2. In order to maintain the intersection set, items in S_1 must be *marked* in some way, so there must be space for an additional 'marker' bit when random tags are applied. Finally, after searching for each element of S_2, S_1 is obliviously sorted (or shuffled and then scanned) in order that the marked elements appear at the start of the list. The complexity of Algorithm 7 depends on the shuffle used. If we shuffle using an oblivious sort, we have an overall time complexity of $O(m_1 \log m_1) + O(m_2 \log m_1) + O(m_1 \log m_1) = O(m_2 \log m_1)$, and we use additional space $O(m_1)$ for the shuffled copy of S_1.

This algorithm is not fully oblivious because the distribution of elements of S_2 into the modelled leaf-nodes of our binary-tree reveals some information about which elements

Algorithm 7: δ-Oblivious Set intersection (binary-search)

```
Input: Sets $S_1$ and $S_2$ stored in external memory, with $|S_1| \leq |S_2|$
Output: Set $S$ written to external memory. $|S| = m_1 + 1$. $S[0]$ is a size variable
        $m$, $S[1..m]$ contains $S_1 \cap S_2$, and $S[m+1..m_1]$ contains
        non-intersection elements of $S_1$.
$m \leftarrow 0$;
$S \leftarrow$ append($[m], S_1$);
obliviously shuffle $S[1..m_1]$;                          ▷ Applying random tags
for $i \leftarrow 0$ to $m_2 - 1$ do
    $s_2 \leftarrow$ read($S_2[i]$);                        ▷ Extended binary-search
    $l \leftarrow 1$; $r \leftarrow m_1$;
    while $l \leq r$ do
        $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$; $s_1 \leftarrow$ read($S[m]$);
        if $s_1 < s_2$ then
            write($S[m], s_1$); $l \leftarrow m + 1$;
        else if $s_1 > s_2$ then
            write($S[m], s_1$); $r \leftarrow m - 1$;
        else
            mark $s_1$; write($S[m], s_1$);                 ▷ $s_2$ found
            while $l \leq r$ do                         ▷ Simulate search to completion
                $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$; $s_1 \leftarrow$ read($S[m]$);
                $c \leftarrow$ random bit;
                if $c = 0$ then
                    write($S[m], s_1$); $l \leftarrow m + 1$;
                else
                    write($S[m], s_1$); $r \leftarrow m - 1$;
write(S[0],m);
return $S$;
obliviously sort $S[1..m_1]$ ;              ▷ s.t. marked entries appear at the
 beginning
```

are in the intersection. Only $\approx \log(m_1)$ items mapped to each leaf node can be in the intersection, as that is the number of elements from S_1 that appear on any path. For example, in Figure 2 a maximum of 3 intersection elements can map to the node tagged 1, or 4 to the node tagged 7. Specifically, for any subset $A \subseteq S_2$ all mapped to a given leaf node with depth d, at most $d+1$ elements of A can be in S. With that information, the adversary knows that, if $|A| > d+1$, there must be at least $|A| - (d+1)$ items in A not in S.

We can show that Algorithm 7 is not fully oblivious by constructing a subset S of access-patterns such that some input I has $Pr[\mathcal{A}(I) \in S] = 0$. Consider two o-neighboring inputs $I = \{S_1, S_2\}$ and $I' = \{S_1, S_2'\}$, where the size of the intersection $m \geq \log m_1$. Take the indexes of the elements of S in the list representation of S_2 to be $loc = \{i_0 ... i_{m-1}\}$ and those of S_2' to be $loc' = \{i_0' ... i_{m-1}'\}$, with $loc \cap loc' = \varnothing$. We construct a subset S of possible access-patterns where all items in loc are mapped to the same leaf node of the binary-tree. In that case, we have $\Pr[\mathcal{A}(I) \in S] = 0$ and $\Pr[\mathcal{A}(I') \in S] > 0$. Since $\Pr[\mathcal{A}(I) \in S] > \Pr[\mathcal{A}(I') \in S]$, this algorithm is not fully oblivious. However, we hope to bound the latter by some negligible δ such that

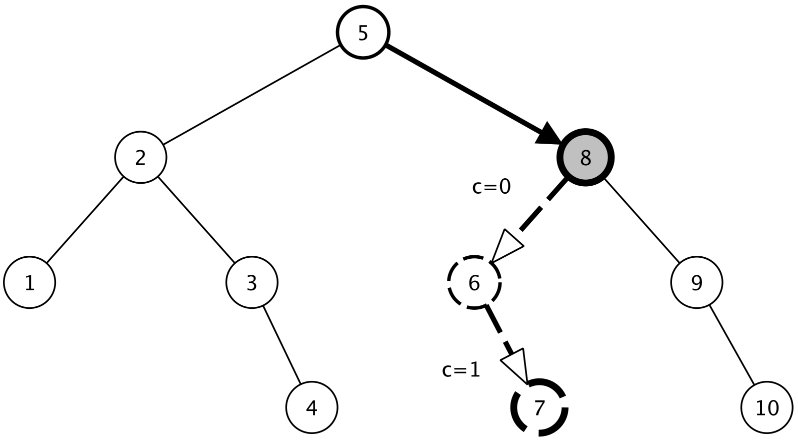

Figure 2: A binary-search-tree model of a modified binary-search from Algorithm 7. Here we are searching in a list tagged [1..10] for the element with tag 8. This element is marked and we continue down the tree according to the labelled 'random bits', ending at the element tagged 7.

$\Pr[\mathcal{A}(I) \in S] \leq \Pr[\mathcal{A}(I') \in S] + \delta$, fulfilling the requirements of δ-statistical obliviousness in Definition 2.3.

The method of mapping elements randomly to leaf nodes of a binary-tree has similarities to methods used in Path ORAM [9].

5.4.2 Hash based algorithm

Algorithm 8: δ-Oblivious Set intersection (hashing)

```
Input: Sets $S_1$ and $S_2$ stored in external memory, with $|S_1| \leq |S_2|$
Output: Set $S$ written to external memory. $|S| = m_1 + 1$. $S[0]$ is a size variable
        $m$, $S[1..m]$ contains $S_1 \cap S_2$, and $S[m+1..m_1]$ contains
        non-intersection elements of $S_1$.
$m \leftarrow 0$;
obliviously shuffle $S_1$;                          ▷ applying random tags
$H \leftarrow$ hash($S_1$);
for $s \in S_2$ do
    if $s \in H(S_1)$ then
        mark $S$;                                   ▷ Mark matching record in $H$
$S \leftarrow$ append($[m], [s : s \in H]$);
$S \leftarrow$ obliviously sort $S[1..m_1]$;       ▷ s.t.  marked entries appear first
return $S$;
```

Algorithm 8 is instead inspired algorithms 3 and 6. S_1 is again shuffled and is then hashed into a table based on a hash function computed on random tags. Random tags are calculated for the items from S_2 are these searched for in the table. When we have

$O(1)$ private memory, the algorithm completes in $O(m_1 \log m_1) + O(m_2)$ expected I/Os. With $O(\sqrt{m_1})$ private memory we can perform the shuffle step in $O(m_1)$ time, so we have overall performance of $O(m_2)$ expected I/Os.

Here we have built the hash table based on the smaller of the sets rather than the larger (as in Algorithms 3 and 6) for two reasons. First, efficiency gains from shuffling only one set are achieved only by shuffling the smaller set. Second, by hashing S_1 we have a smaller hash table, $|H| = \Theta(m_1)$; when searching for elements of S_2 in H we have an average of $a = \frac{m_2}{|H|}$ items hashed to each bucket, since $m_2 \gg m_1 \Rightarrow a \gg 1$, this adds additional noise to the access-pattern, with many items from S_2 hashed to each bucket.

The obliviousness in this case is again compromised by S_2 not being shuffled. Several information leaks, as well as possible mitigating alterations, are identified:

1. By observing the hashing of S_1, the adversary knows how many elements are in each bucket of the H. Consider some empty bucket of H: any elements in S_2 hashed to that bucket can be excluded as an element of S. An additional shuffle step, whereby the hash table itself is padded with dummy values and shuffled, to remove this concern.

2. Similarly, consider some bucket of H containing some x number of elements from S_1: of the *group* of y elements from S_2 hashed to that bucket, the adversary knows that at most x of them are in S. Capping the size of buckets, and padding each bucket to this size, would result in a more even distribution of the probabilities of each element of S_2 being in the intersection, but does not stop the grouping effect. This solution also adds to the I/O complexity of the algorithm, and adds a possibility of failure when a bucket overflows.

3. We also must consider how the buckets are searched, if there are several elements from S_1 within a bucket, and a search for some $s \in S_2$ hashed to that bucket completes before they have all been compared, the adversary knows that $s \in S_2$. This can be avoided by ensuring that every element of a bucket is compared during a search, a similar principle to extending the binary search in Algorithm 7.

Further analysis is required to determine the effects of these leaks and alterations on a δ privacy guarantee and, in the case of alterations, the efficiency of the algorithm.

6 Conclusion

We have provided a background on oblivious algorithms and defined a new security guarantee, output constrained (OC) obliviousness. We have applied this definition, drawing on common set intersection algorithms, to develop novel OC-oblivious set intersection algorithms based on binary-search and hashing. We have also considered an alternate relaxation, δ-statistical obliviousness and made steps toward developing more efficient set intersection under this definition.

Our results are encouraging, with OC-oblivious set intersection algorithms outperforming a fully oblivious counterpart when allowed a private memory of size $O(\sqrt{n})$. These results, along with analysis of the definitions of *neighboring* inputs under various obliviousness constructions, suggest that OC-obliviousness can provide improved efficiency while maintaining a strong security guarantee. The small scope of this project did not allow us to perform a more in-depth analysis of the security provided by our definition, or to prove any lower bounds on performance. We also took steps towards developing even more efficient δ-statistically oblivious set intersections, but were not able to prove they had negligible δ bounds on privacy failure.

These limitations open up a number of avenues for further enquiry:

- Applying our OC-oblivious security definition to develop algorithms for a wider range of problems
- Attempting to improve OC-oblivious set intersection with smaller private memory
- Exploring fast set intersection techniques from information retrieval [25, 26, 27] in OC-oblivious set intersection, as these rely on both sets being pre-sorted, which is the case for random tags when both sets are shuffled
- Proving lower bounds for set intersection under all the presented security definitions
- Further analysis of our (possibly) δ-statistically oblivious set intersection algorithms
- Composition of our OC-obliviousness definition with DP-obliviousness, with the outlook of further efficiency improvements.

We hope that this work will provide a wider range of more efficient options for access-pattern privacy, both in set intersection and further afield.

References

[1] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: ramification, attack and mitigation," in *NDSS*, vol. 20, p. 12, 2012.

[2] K. E. Batcher, "Sorting networks and their applications," in *SJCC*, pp. 307–314, AFIPS, 1968.

[3] M. Ajtai, J. Komlós, and E. Szemerédi, "An $O(n \log n)$ sorting network," in *STOC*, pp. 1–9, ACM, 1983.

[4] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious RAM simulation," in *SODA*, pp. 157–167, SIAM, 2012.

[5] G. Asharov, T. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi, "Bucket oblivious sort: An extremely simple oblivious sort," in *SOSA*, pp. 8–14, SIAM, 2020.

[6] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal, "The Melbourne shuffle: Improving oblivious storage in the cloud," in *ICALP*, pp. 556–567, EATCS, 2014.

[7] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *STOC*, pp. 182–194, ACM, 1987.

[8] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *STOC*, pp. 514–523, ACM, 1990.

[9] 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*, pp. 299–310, ACM SIGSAC, 2013.

[10] S. Tople, H. Dang, P. Saxena, and E.-C. Chang, "Permuteram: Optimizing oblivious computation for efficiency.," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 885, 2017.

[11] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[12] N. Pippenger and L. G. Valiant, "Shifting graphs and their applications," *Journal of the ACM (JACM)*, vol. 23, no. 3, pp. 423–432, 1976.

[13] T. H. Chan, K.-M. Chung, B. M. Maggs, and E. Shi, "Foundations of differentially oblivious algorithms," in *SODA*, pp. 2448–2467, SIAM, 2019.

[14] S. Wagh, P. Cuff, and P. Mittal, "Differentially private oblivious RAM," *PoPETs*, vol. 2018, no. 4, p. 64, 2018.

[15] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of Computer and System Sciences*, vol. 28, no. 2, pp. 270 – 299, 1984.

[16] Z. Chang, D. Xie, and F. Li, "Oblivious RAM: a dissection and experimental evaluation," *Proc. VLDB Endow.*, vol. 9, no. 12, p. 1113–1124, 2016.

[17] J. Allen, B. Ding, J. Kulkarni, H. Nori, O. Ohrimenko, and S. Yekhanin, "An algorithmic framework for differentially private data analysis on trusted processors," in *NeurIPS*, pp. 13635–13646, 2019.

[18] K. G. Larsen, T. Malkin, O. Weinstein, and K. Yeo, “Lower bounds for oblivious near-neighbor search,” in *SODA*, pp. 1116–1134, SIAM, 2020.

[19] J. Seiferas, “Sorting networks of logarithmic depth, further simplified,” 2006.

[20] M. T. Goodrich, “Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time,” in *STOC*, pp. 684–693, ACM, 2014.

[21] J. Katz and Y. Lindell, *Introduction To Modern Cryptography.* CRC Press, 2015.

[22] C. Dwork, F. McSherry, K. Nissim, and A. Smith, “Calibrating noise to sensitivity in private data analysis,” in *TCC*, p. 265–284, IACR, 2006.

[23] X. He, A. Machanavajjhala, C. Flynn, and D. Srivastava, “Composing differential privacy and secure computation: A case study on scaling private record linkage,” in *CCS*, pp. 1389–1406, ACM, 2017.

[24] A. Arasu and R. Kaushik, “Oblivious query processing,” in *ICDT*, 2014.

[25] R. Baeza-Yates, “A fast set intersection algorithm for sorted sequences,” in *CPM*, pp. 400–408, 2004.

[26] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger, “An experimental investigation of set intersection algorithms for text searching,” *Journal of Experimental Algorithmics (JEA)*, vol. 14, pp. 3–7, 2010.

[27] B. Ding and A. C. König, “Fast set intersection in memory,” *Proc. VLDB Endow.*, vol. 4, no. 4, 2011.