

Efficient Oblivious Database Joins

by

Simeon Krastnikov

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

© Simeon Krastnikov 2020

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis contains content from the PVLDB 2020 paper “Efficient Oblivious Database Joins” [28]. The author of this thesis was responsible for all parts of the paper, while being supervised by his co-authors Professor Florian Kerschbaum and Professor Douglas Stebila.

Abstract

A major algorithmic challenge in designing applications intended for secure remote execution is ensuring that their execution is oblivious to their inputs, in the sense that their memory access patterns do not leak sensitive information to the server. This problem is particularly relevant to cloud databases that wish to allow queries over the client’s encrypted data. One of the major obstacles to such a goal is the join operator, which is non-trivial to implement obliviously without resorting to generic but inefficient solutions like Oblivious RAM (ORAM).

We present an oblivious algorithm for equi-joins which (up to a logarithmic factor) matches the optimal $O(n \log n)$ complexity of the standard non-secure sort-merge join (on inputs producing $O(n)$ outputs). We do not use expensive primitives like ORAM or rely on unrealistic hardware or security assumptions. Our approach, which is based on sorting networks and novel provably-oblivious constructions, is conceptually simple, easily verifiable, and very efficient in practice. Its data-independent algorithmic structure makes it secure in various different settings for remote computation, even in those that are known to be vulnerable to certain side-channel attacks (such as Intel SGX) or with strict requirements for low circuit complexity (like secure multiparty computation). We confirm that our approach is easily realizable by means of a compact implementation which matches our expectations for performance and is shown, both formally and empirically, to possess the desired security characteristics.

Acknowledgements

Above all, I would like to thank my supervisors Professor Florian Kerschbaum and Professor Douglas Stebila for their invaluable guidance, inspiration, and support. I am also grateful to my readers Professor Xi He and Professor Semih Salihoglu for their insightful questions and suggestions. Lastly, I would like to acknowledge the financial support provided by NSERC and the Royal Bank of Canada.

Table of Contents

List of Figures	viii
List of Tables	ix
1 Introduction	1
1.1 Contributions	3
1.2 Outline of Thesis	4
2 Oblivious Programs	5
2.1 Computing on Encrypted Data	5
2.2 Adversarial Model	7
2.3 Degrees of Obliviousness	7
2.4 Oblivious RAM (ORAM)	10
2.5 From Oblivious Programs to Circuits	10
2.6 Oblivious Sorting	12
3 Problem Overview	13
3.1 Problem Definition	13
3.2 Non-Oblivious Sort-Merge Join	13
3.3 Prior Work on Oblivious Joins	14
3.4 Security Characteristics	17
3.5 Overview of Approach	17
4 Algorithm Description	20
4.1 Obtaining Group Dimensions	20
4.2 Oblivious Distribution	23
4.3 Oblivious Expansion	27
4.4 Table Alignment	30

5	Evaluation	32
5.1	Security Analysis	32
5.2	Performance Analysis	35
6	Conclusions and Future Work	38
6.1	Compound Join Queries	38
6.2	Non Equi-Joins	39
6.3	Other Oblivious Algorithms	39
6.4	Obliviousness-Preserving Compilers	40
	References	42
	APPENDICES	48
A	Additional Pseudocode	49
A.1	Group Dimension Computation	49

List of Figures

3.1	Memory access patterns of the sort-merge algorithm on two different inputs.	15
3.2	Main idea of the algorithm.	19
4.1	Example group dimension calculation.	21
4.2	Example oblivious distribution.	25
4.3	Example oblivious expansion.	28
4.4	Example table alignment.	31
5.1	Summary of type system used to model level II obliviousness and verify implementation.	34
5.2	Visualization of our implementation's input-independent pattern of memory access.	35
5.3	Performance results for sequential prototype implementation.	37

List of Tables

1.1	Comparison of approaches for oblivious database joins.	3
2.1	Properties of three levels of obliviousness.	8
5.1	Comparison count and runtime share for each component of the algorithm.	37

Chapter 1

Introduction

With an increasing reliance on cloud-based services to store large amounts of user data securely, there is also a growing demand for such services to provide remote computation in a privacy-preserving manner. This is a vital requirement for cloud databases that store sensitive records and yet wish to support queries on such data.

Various different mechanisms exist that achieve this purpose, for instance, dedicated hardware in the form of secure cryptographic coprocessors or hardware enclaves like Intel SGX [11], which come in the form of a dedicated set of processor instructions that allow computation on encrypted data. Although such approaches provide good cryptographic guarantees in that they ensure the contents of the user’s data remain encrypted throughout the execution of the remote program, on their own they provide no guarantees to address a major source of information leakage: the memory access patterns of the execution. If the program reads and writes to specific addresses of the untrusted server’s memory in a way that depends on the input user data it receives, these access patterns can reveal, via various side-channels, information to the server about the user’s data.

Consider, for example, the standard $O(n \log n)$ sort-merge algorithm for database joins. The two input tables are first sorted by their join attribute values and then scanned for matching entries by keeping track of a pointer at each table. At each step either one of the pointers is advanced (depending on the result of the comparison between the two join attribute values), or an entry is appended to the output. If the tables are stored in regular memory, an adversary observing memory patterns will obtain input-dependent information at each step. Namely, at each step it will learn the locations of the two entries read from the input table, and depending on whether an entry is written to output, it will learn whether the entries match. This information can reveal critical information about the user’s input.

Protecting a program against such leaks amounts to making it *oblivious*: that is, ensuring that its decisions about which memory locations to access do not depend on the

actual contents of its input data. For many programs this is difficult to accomplish without introducing substantial computational overhead [47]. The generic approach is to use an Oblivious RAM (ORAM), which provides an interface through which a program can access memory non-obliviously, while at the same time providing guarantees that the actual physical accesses of such programs turn out to be oblivious. Though the design of ORAM schemes has been the central focus of oblivious algorithm design, such schemes have a very high computational overhead, not only due to their asymptotic overhead (due to a theoretical lower bound of $O(\log n)$ for every access to an array of n entries) but also due to the fact that they can be inefficient in practice [26, 27, 38, 47] and in some cases – insecure [1].

Another approach to guaranteeing obliviousness is to assume a limited but non-constant amount of memory that can be accessed non-obliviously. While such an assumption may make sense in certain settings (for example, cryptographic coprocessors may provide internal memory protected from the untrusted system), it is unsafe to make in the more common hardware enclave setting due to a wide range of attacks that can infer access patterns to enclave memory itself [9, 25, 29, 40, 42, 44].

These considerations motivate the need for the design of problem-specific oblivious algorithms that closely approach the efficiency of their non-secure counterparts without the use of generic primitives or reliance on hardware assumptions. Such algorithms are very similar to circuits in that their control flow is independent of their inputs. This not only provides security against many side-channel attacks (beyond those involving memory accesses), but also makes them very suitable for use in secure multiparty computation where programs with low circuit complexity achieve the best performance [43, 48]. One of the best-known oblivious algorithms are sorting networks such as those proposed by Batcher [6], which match (up to at most a logarithmic factor) the standard $O(n \log n)$ complexity of sorting and are often a critical component in other oblivious algorithms such as ours.

Many database operators such as selection and aggregation can be implemented obliviously as trivial linear passes through the table. On the other hand, database joins, being among the most algorithmically complex operators, have proven to be very difficult to make oblivious in the general case. This is due to the fact that one cannot allow any such oblivious algorithm to base its memory accesses on the structure of the input tables (with respect to how many entries from one table match a given entry in the other table). As such, joins have been the prime focus of work on oblivious database operators: prior work is summarized in Table 1.1 and discussed in detail in Section 3.3.

Algorithm/System	Time complexity	Local Memory	Limitations
Standard sort-merge join	$O(m' \log m')$	$O(1)$	not oblivious
Agrawal et al. [2] (Alg. 3)	$O(n_1 n_2)$	$O(1)$	insecure (see § 2.3.1 of [30])
Li and Chen [30] (Alg. A2)	$O(mn_1 n_2 / t)$	$O(t)$	–
Opaque [49] and ObliDB [14]	$O(n \log^2(n/t))$	$O(t)$	primary-foreign key joins only
Oblivious Query Processing [4]	$O(m' \log^2 m')$	$O(\log m')$	details missing; perf. concerns
Ours	$O(m' \log^2 m')$	$O(1)$	–

Table 1.1: Comparison of approaches for oblivious database joins. n_1 and n_2 are the input table sizes, $n = n_1 + n_2$, m is the output size, $m' = m + n_1 + n_2$, t is the amount of memory assumed to be oblivious. The time complexities are in terms of the number of database entries and assume use of an $O(n \log^2 n)$ sorting network for oblivious sorting (where applicable).

1.1 Contributions

We fully describe an oblivious algorithm for binary database equi-joins that achieves $O(n \log^2 n + m \log^2 m)$ running time, where n is the input length (the size of both tables) and m is the output length, thus matching the running time of the standard non-oblivious sort-merge join up to a logarithmic factor. It can also achieve a running time of $O(n \log n + m \log m)$ but only using a sorting network that is too slow in practice. Our algorithm does not use ORAM or any other computationally-expensive primitives and it does not make any hardware assumptions other than the requirement of a constant-size working set of memory (on the order of the size of one database entry), for example to compare two entries. In other words, any model of computation that can support a sorting network on encrypted data can also support our algorithm; this includes secure coprocessors or hardware enclaves like Intel SGX, in which case obliviousness provides resistance against various side-channel attacks. In addition, because our program is analogous to a circuit, it is very suitable for use in settings like secure multiparty computation and fully homomorphic encryption.

Our approach is conceptually very simple, being based on a few repeated runs of a sorting network and other basic primitives. As such, it is both efficient, amenable to a high degree of parallelization, and easy to verify for obliviousness. We have a working prototype implementation consisting of just around 600 lines of C++ code, as well as a version that makes use of SGX. We have used a dedicated type system to formally verify the obliviousness of the implementation and have conducted experiments that empirically examine its memory accesses and runtime.

1.2 Outline of Thesis

We begin by providing, in Chapter 2, a background on the different modes of computation that our algorithm is compatible with and discuss the goals, challenges, and methodologies related to oblivious algorithm design. Chapter 3 describes the target problem, prior related work, and the intuition behind our approach. The full algorithm is then described in detail in Chapter 4, followed by the specifics of our implementation in Chapter 5, where we also analyze its security and performance. Finally, Chapter 6 summarizes our results and discusses directions for future work.

Chapter 2

Oblivious Programs

Intuitively, a program is *data-oblivious* (or simply *oblivious*) if its control flow, in terms of the memory accesses it makes, is independent of its input given the input size. That is, for all inputs of the same length, the sequence of memory accesses made by an oblivious program is always identical (or identically distributed if the program is probabilistic). This is indeed well-defined for simple computational models like random-access machines and Turing machines; for instance, a Turing machine is said to be oblivious if the motions of its head are independent of its input. However, we need to carefully account for real-world hardware where there can be different types of memory as well as various side-channels unaccounted for in simpler models.

In this chapter, we talk about the various mechanisms used to perform computation on encrypted data, we define our adversarial model, and introduce different degrees of obliviousness that one can obtain under such model. We also introduce various tools and methodologies related to obliviousness.

2.1 Computing on Encrypted Data

Users who have securely stored their data on a remote server often need to perform computation on their encrypted data, for example, to execute database queries. We discuss and contrast several different approaches that strive to achieve this purpose.

Outsourced External Memory

In this setting (discussed in [24]), there is no support for server-side computation on the client's private data. The client treats the server as external memory: if they wish to com-

pute on their data, they must do so locally. This scenario is clearly impractical for intensive computations due to the significant differences between RAM and network latency.

Secure Cryptographic Coprocessors

A cryptographic coprocessor (e.g., [5]) is a tamper-proof device that can perform computation within its own trusted region isolated from its external host. Through the use of remote attestation, the client can send a trusted code base (TCB) to the coprocessor and have it execute within a secure environment shielded from the semi-trusted server (according to the Trusted Platform Module specification [15]). The drawback to this approach is that such hardware provides very limited memory and computational power, and imposes hardware requirements on the server.

Trusted Execution Environments (TEE)

Hardware enclaves such as Intel SGX [11], provide similar guarantees to coprocessors in that the client can make use of remote attestation to run a TCB within a trusted execution environment (TEE) that provides guarantees for authenticity and some protection against an untrusted OS. Such designs are becoming increasingly prevalent in new processors, taking the form of a specialized set of instructions for setup and access to the TEE. The enclave provides a limited amount of memory called the Enclave Page Cache (EPC), which resides on the system’s main memory but cannot be accessed by other processes (including the kernel). In addition, the contents of the enclave are encrypted, and the processor can seamlessly read, write and perform logical and arithmetic operations on this data. Although these properties make it seem that hardware enclaves provide a secure container completely isolated from the untrusted OS, numerous papers [9, 25, 29, 40, 42, 44] have shown that enclaves like Intel SGX are susceptible to numerous side-channel attacks, e.g., cache attacks that infer data-dependent information based on memory access patterns to enclave memory.

Secure Multiparty Computation (SMC)

In the general setting, secure multiparty computation [19] allows several parties to jointly compute *functionalities* on their secret inputs without revealing anything more about their inputs than what can be inferred from the output. The two standard approaches are Yao’s garbled circuit protocol [45], and the Goldreich-Micali-Wigderson protocol [20], based on secret sharing. Both approaches require the desired functionality to be expressed as a boolean circuit, and the output is computed gate by gate. Practical implementations of SMC include the SCALE-MAMBA system [3], as well as the OblivM framework [32], which

allows programs to be written in a (restricted) high-level syntax that can then be compiled to a circuit. Outsourcing computation using SMC is usually done using a distributed protocol involving a cluster of several servers [7].

Fully Homomorphic Encryption (FHE)

Cryptosystems such as that of Gentry et al. [16] allow arbitrary computation on encrypted data. As in SMC, such schemes require the target computation to be represented as a boolean circuit. Although FHE provides solid theoretical guarantees and several implementations already exist, it is currently computationally too expensive for practical use.

2.2 Adversarial Model

We will use an abstract random access machine model of computation where we distinguish between two types of memory. During an execution of a program, the adversary has complete view and control of the *public memory* (for example, RAM) used throughout its execution. However, the program may use a small amount of *local memory* (or *protected memory*) that is completely hidden from the adversary (for example, processor registers). The program may use this memory to perform computations on small chunks of data; the adversary learns nothing about such computations except the time spent performing them (we assume that all processor instructions involving local memory that are of the same type take an equal amount of time).

We assume that the adversary cannot infer anything about the individual contents of individual cells of public memory, as well as whether the contents of a cell match a previous value. This can be achieved through the use of a probabilistic encryption scheme.

2.3 Degrees of Obliviousness

We distinguish between three different levels of obliviousness that a given program may satisfy (summarized in Table 2.1), from weakest to strongest, each subsuming the lower levels. The distinctions will be based on how much local memory the program assumes and whether the program's use of local memory leaks information through side-channels. We will restrict our attention to deterministic programs; the concepts easily generalize to the probabilistic case.

Property/Setting	I	II	III
Constant local memory	×	✓	✓
Circuit-like	×	×	✓
Ext. Memory	t	t	✓
Secure Coprocessor	t	t	✓
TEE (enclave)	t, pd, pc, c, b	t, pc, c, b	✓
Secure Computation	n/a	n/a	✓
FHE	n/a	n/a	✓

Table 2.1: Properties of three levels of obliviousness. Top portion of the table shows whether the given property is satisfied at each of the levels. Bottom portion shows vulnerability of programs satisfying these levels to timing (t), page access attacks on data (pd), page access attacks on code (pc), cache-timing (c), or branching (b) attacks when used in different settings.

Level I

A program is oblivious in this sense if its accesses to public memory are oblivious but it requires a non-constant amount of local memory used for non-oblivious computation. This memory may be accessed whenever required and for any duration of time. Such programs are suitable for use in the outsourced external memory model since the client can use as much local memory as there is memory on his machine. They may also be suitable for use in a secure coprocessor model setting since coprocessors have an internal memory separated from the rest of the system. However, in both of these scenarios, timing attacks may be an issue: e.g., if the local memory is used for variable lengths of time between pairs of public accesses.

Examples of algorithms that are oblivious in this sense are those proposed by Goodrich [13, 22, 24], which are well-suited for the outsourced external memory model.

Level II

At this level we not only require, as before, accesses to public memory to be oblivious but also that the amount of local memory used by the program be bounded by a constant. In practice, the exact size of this constant depends on the amount of available CPU register and cache memory, which can be used for example, to compute the condition for a branch or to perform an arithmetic operation on two words read (obliviously) from RAM. Any such accesses must be on inputs that fit in one cache line so as to not cause non-oblivious

RAM accesses due to cache evictions.

Making the distinction between this level and the previous is motivated by the fact that hardware enclaves like Intel SGX are vulnerable to side-channel attacks based on page-level accesses patterns to enclave memory itself [40, 44], which have been shown to be extremely powerful, often succeeding in extracting sensitive data and even whole files. Therefore one cannot assume that the Enclave Page Cache provides oblivious memory. In works like Oblix [33] level II programs are called *doubly-oblivious* since, in the hardware enclave context, their accesses to both regular and enclave memory are oblivious.

Although it may seem that a level II program is safe against the above attacks, this is not quite the case, though it certainly fares better in this respect than a level I program. The *data* of the program will be accessed obliviously, but its actual machine code, which is stored in memory, will be accessed based on the control flow of the program. The program may branch in a data-dependent way and though the memory accesses to public data in both branches are required to be the same, each branch will access a different fragment of the program’s machine code, thus leaking information about the data that was branched on.

Level III

This is a strong notion of obliviousness where we require that the control flow of the program, down to the level of the exact processor instructions it executes, be completely independent of its input, except possibly its length. In other words, the program counter has to always go through the same sequence of values for all inputs of the same length. We can think of such a program as a family of circuits, one for each input size; as such, it is very well-suited for secure multiparty computation (and fully homomorphic encryption).

This definition is also motivated by the fact that additional measures are required to provide protection against attacks based on accesses to machine code as well as the fact that hardware enclaves have also been shown to be vulnerable to a variety of other side-channel attacks such as cache-timing [9, 25, 42], branching [29], or other types of timing attacks. Such attacks can infer data based on the control flow of a program at the instruction level: this includes the way it accesses the registers and cache of the processor, as well as the exact number of instructions it performs. Making sure that a given program compiles to a level III binary is a challenging problem that requires thorough consideration of the compiler’s enabled optimizations as well as the target architecture. Therefore this level pertains more to a specific implementation of an algorithm as opposed to its high-level description.

Revealing Output Length

By producing an output of length m , a program reveals data-dependent information about the input. We can always eliminate this problem by padding the output to its maximum possible size; however, this can result in suboptimal running time. For instance, the join operator can produce an output of up to $O(n^2)$ on an input of size n , which means that any join algorithm that pads its output must have at least quadratic runtime. For this reason, we will only consider programs that do not pad their output and thus leak the output size m , as well as their runtime.

2.4 Oblivious RAM (ORAM)

The most general approach to making arbitrary programs oblivious (in any of the above senses) is to use an Oblivious RAM (ORAM), a primitive first introduced by Goldreich and Ostrovsky [18, 21]. An ORAM simulates a regular RAM in such a way that its apparent physical memory accesses are independent of those being simulated, thus providing a general approach to compiling general programs to oblivious ones. In other words, by using an ORAM as an interface through which we store and access sensitive data, we can eliminate access pattern leaks, though in doing so we incur at least a logarithmic overhead per memory access according to the Goldreich-Ostrovsky lower bound [21]. Even if such overhead is acceptable in terms of the overall asymptotic complexity, ORAM constructions tend to have prohibitively large constant overhead, which make them impractical to use on reasonably-sized inputs.

One of the well-known ORAM schemes is Path ORAM [39], which produces programs that satisfy level I obliviousness (Obliv [33] gives a modification that is oblivious at level II). Various schemes have been introduced that make ORAM more suitable for SMC by optimizing its resulting circuit complexity (i.e., how close it is to producing a level III program) [12, 17, 43, 48]. Despite the abundance of ORAM schemes, their high performance cost [26, 27, 38, 47] (due to their polylogarithmic complexity overhead, their large hidden constants, and issues with parallelizability), calls for a need for problem-specific oblivious algorithm design.

2.5 From Oblivious Programs to Circuits

Given a program that satisfies level II obliviousness, approaches exist to transform it into a circuit-like level III program while introducing only a constant overhead [31, 32, 34, 37].

There are three additional constraints that the control flow of the program must satisfy for this to be the case:

1. Any loop condition must depend on either a constant or the input size. This corresponds to the fact that all loops must be unrolled if one wishes to obtain a literal boolean circuit. For instance if *secret* is a variable that depends on the contents of the input data, we cannot allow behaviour like:

```

i ← 0
while i < secret do
    i ← i + 1

```

Though this code will make no memory accesses if the *i* counter is stored in a register, it is very hard to automatically protect such code against timing attacks in general (though in this case the fix is obvious: replace the while loop with $i \leftarrow secret$).

2. The branching depth of the program — the maximum number of conditional branches encountered by any given run — is constant. This requirement allows us to eliminate conditional statements without affecting runtime complexity. A statement like

```

if secret then
    x1 ← y1
    x3 ← y3
else
    x1 ← z1
    x2 ← z2

```

can be replaced by the following, where *secret* is treated as a numerical value from the set $\{0, 1\}$:

```

x1 ← y1 · secret + z1 · (¬secret)
x2 ← z2 · 0 + z2 · (¬secret)
x3 ← y3 · secret + y3 · 0

```

This increases the total computation by a factor of 2. On the other hand, if we have a sequence of d nested conditional statements, the computational overhead will be on the order of 2^d , which is why we require d to be constant.

3. If the program reveals the output length m , it does so only after allocating $m_0 \in \Omega(m)$ memory. This is so the level II program can be split into two circuit-like level III programs that are to be run in sequence: one parameterized by n that computes the value of m_0 , and a second parameterized by both n and m_0 .

2.6 Oblivious Sorting

Sorting networks such as bitonic sorters [6] provide an in-place input-independent way to sort n elements in $O(n \log^2 n)$ time, taking the form of an $O(\log^2 n)$ -depth circuit. Although $O(n \log n)$ constructions also exist (such as Goodrich’s Zig-zag sort [23]), they are either very inefficient in practice (due to large constant overheads) or non-parallelizable. Each non-recursive step of a bitonic sorter reads two elements at fixed input-independent locations, runs a comparison procedure between the two elements and swaps the entries depending on the result. To ensure obliviousness, even if the elements are not to be swapped, the same (re-encrypted) entries are written to their original locations. When a probabilistic encryption scheme is used, this leaks no information about whether the two elements were swapped.

We parameterize our calls to a bitonic sorter with a lexicographic ordering on chosen element attributes. For example, if A is a list of elements where each element has attributes x, y, z, \dots , then

$$\text{BITONIC-SORT}\langle x \uparrow, y \uparrow, z \downarrow \rangle(A)$$

will sort the elements in A by increasing x attribute, followed by increasing y attribute, and then by decreasing z attribute.

We can use sorting networks as filters. For instance, we will use \emptyset to designate a void (null) entry that is marked to be discarded (or often a “dummy” entry) so that if we know that A of size n has k non-null elements, we can run

$$\text{BITONIC-SORT}\langle \neq \emptyset \uparrow \rangle(A)$$

and collect the first k non-null elements in the output. Alternatively, Goodrich [22] has proposed an efficient $O(n \log n)$ oblivious algorithm specifically for this problem (there referred to as *compaction*).

Chapter 3

Problem Overview

In this chapter, we describe the general problem, the security goals our solution will satisfy, and prior work in similar directions. We then briefly outline the general idea behind our approach.

3.1 Problem Definition

We are given as input two unsorted tables T_1 and T_2 and are interested in computing the binary equi-join of both tables. For simplicity, we can assume that T_1 and T_2 each consist of respectively n_1 and n_2 (possibly-repeated) pairs (j, d) (we call j a *join (attribute) value* and d a *data (attribute) value*). The output we would like to compute is the multiset

$$T_1 \bowtie T_2 = \{(d_1, d_2) \mid (j, d_1) \in T_1, (j, d_2) \in T_2\}.$$

The tables T_1 , T_2 , and $T_1 \bowtie T_2$ are not assumed to be ordered.

3.2 Non-Oblivious Sort-Merge Join

The sort-merge join algorithm (Algorithm 3.1) is, along with hash join algorithm, one of the frequently used implementations of the join operator. After the two input tables are sorted by their join attribute values, the matching output entries can be obtained in a linear merge phase. In this phase, the progress made in each of the two tables is tracked by two indices (i_1 and i_2 in the pseudocode), which in turn determine the next two entries to be fetched from memory and compared against each other. Thus a certain structure of the tables is revealed through the sequence of entries accessed through time (see Figure

Algorithm 3.1 Sort-merge join

```
1: function SORT-MERGE( $T_1, T_2$ )
2:   sort  $T_1$  and  $T_2$  on  $j$ 
3:   initialize output table  $T_O$ 
4:    $i_1 \leftarrow 1$ 
5:    $i_2 \leftarrow 1$ 
6:    $i'_2 \leftarrow 1$ 
7:   while  $i_1 \leq |T_1| \wedge i_2 \leq |T_2|$  do
8:     if  $T_1[i_1].j = T_2[i_2].j$  then
9:       add  $(T_1[i_1].d, T_2[i_2].d)$  to  $T_O$ 
10:       $i_2 \leftarrow i_2 + 1$ 
11:     else if  $T_1[i_1].j < T_2[i_2].j$  then
12:        $i_1 \leftarrow i_1 + 1$ 
13:        $i_2 \leftarrow i'_2$ 
14:     else
15:        $i_2 \leftarrow i_2 + 1$ 
16:        $i'_2 \leftarrow i_2$ 
17:   return  $T_O$ 
```

3.1). If, for example, there are many entries in the second table matching an entry from the first, an adversary observing memory will see a long sequence of consecutive accesses to the second table, as the index to the first table stays fixed. Even though in practice, the adversary would be able to access such information only via side channels, such leakage can completely undermine any encryption of the input data, especially if the adversary has prior knowledge about the data stored in the tables.

3.3 Prior Work on Oblivious Joins

The design of oblivious join operators has been studied both in isolation and also as part of larger privacy-oriented database systems, where it is emphasized as the most challenging component; we compare several different approaches in Table 1.1. Agrawal et al. [2] propose several join algorithms for use in a setting similar to ours; however, their roughly $O(n_1 n_2)$ complexity is close to that of a trivial $O(n_1 n_2 \log^2(n_1 n_2))$ oblivious algorithm based on a nested loop join. Additionally, their security definition allows leakage of a certain property of the inputs (as pointed out in [30], where the issue was fixed without significant improvements in runtime). SMCQL [7] is capable of processing SQL queries through se-

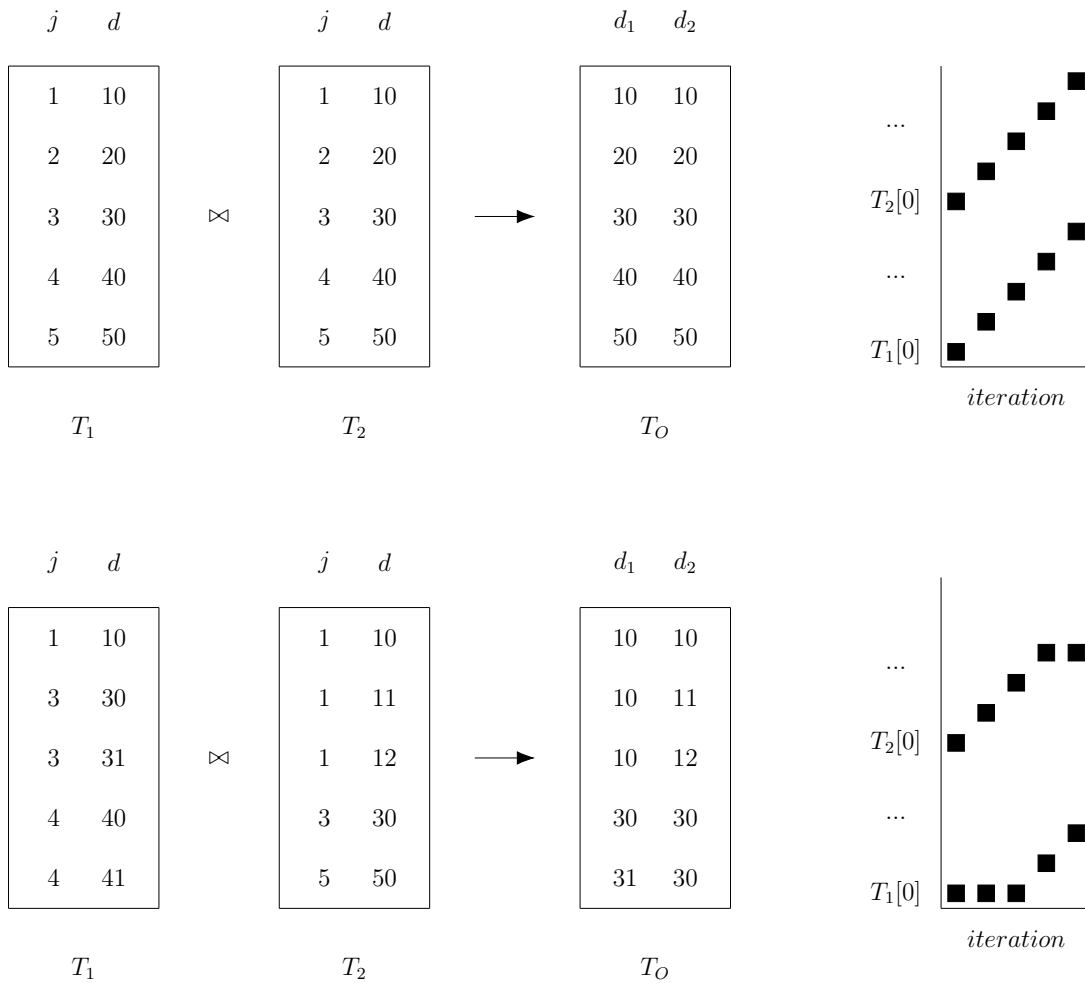


Figure 3.1: Memory access patterns of a sort-merge join's merge phase on two different inputs. Two examples are shown, each consisting of a pair of input tables with five entries each that produce an output with five entries. The graphs on the right show the sequence of accesses to the input tables at every iteration where an entry is added to the output. The two sequences completely reveal the join relationship between the corresponding input tables.

cure computation primitives but its secure join also runs in $O(n_1n_2)$ time. Conclave [41] implements join operators for SMC; however, its approach involves revealing entries to a “selectively-trusted party”.

Opaque, which is geared towards private database queries in a distributed setting, implements an oblivious sort-merge algorithm [49] (as well as its variant in OblIDB [14]) but handles only the specific case of primary-foreign key joins (in which case $m = O(n)$ and their $O(n \log^2(n/t))$ complexity matches ours for constant t). Though Opaque makes use of the $O(t)$ available enclave memory to optimize its running time, such optimizations rely on “enclave designs that protect against access patterns to the EPC” such as “Sanctum, GhostRider, and T-SGX” to obtain a pool of oblivious memory (meaning that the optimized versions are only level I oblivious). Such constructions could potentially introduce a computational overhead that outweigh any optimizations (GhostRider for instance relies on ORAM) or introduce additional hardware requirements and security assumptions.

The closest to our work is that of Arasu and Kaushik [4], which mirrors the overall structure of our algorithm but ultimately reduces to a different (arguably more challenging) problem than the one we deal with (“obliviously reordering [sequences] to make [them] barely prefix heavy”). The details for the proposed solution to this problem are incomplete and the authors have not provided a proof-of-concept implementation that shows empirical results. We believe that even if a solution to the problem in question exists, the overall algorithm will be less efficient than ours due to the high constant overheads from repeated sorts. Lastly, their approach also assumes, by default, a local memory of $O(\log(m + n))$ entries (thus being level I oblivious) since it is intended for use in a secure coprocessor setting. For use in more practical settings like Intel SGX, such memory would either have to be obtained in the same manner as was argued above for the case of Opaque or through an $O(\log(m + n))$ time complexity overhead for each local memory access (that is, if each access achieves obliviousness by reading all entries in local memory).

Encrypted databases like CryptDB [36] employ deterministic and partial homomorphic encryption to process databases in a hardware-independent way but such databases are non-oblivious. Private Set Intersection (PSI) and Private Record Linkage (PRL) are somewhat similar problems to the one considered in this work in that they involve finding matching entries among different databases. Although some protocols for these problems rely solely on SMC techniques (by constructing circuits as in our work), more efficient protocols make use of cryptographic primitives like oblivious transfer that are not applicable to database joins (see the survey on PSI in [35]).

3.4 Security Characteristics

Intuitively, our algorithm will be oblivious with regards to the way it accesses any memory with non-constant size. More precisely, we will provide security in the form of level II obliviousness, as described in Section 2.3. Hence, our condition will be that for all inputs of length n that produce outputs of equal length m , the sequence of memory accesses our algorithm makes on each the inputs is always the same (or identically distributed if we make use of randomness).

We will use a constant amount of local memory on the order of the size of a single database entry, which we will use to process entries and keep counters. That is, our accesses to public memory (where the the input, output and intermediate tables are stored) will be of the form

$$\begin{array}{l}
 e \stackrel{*}{\leftarrow} T[i] \\
 \dots \\
 \text{(sequence of operations on } e\text{)} \\
 \dots \\
 T[i] \stackrel{*}{\leftarrow} e
 \end{array}$$

The notation $e \stackrel{*}{\leftarrow} T[i]$ explicitly signifies that the i -th entry of the table T , which is stored in public memory, is read into the variable e stored in local memory. Our code will be such that the memory trace consisting of all $\stackrel{*}{\leftarrow}$ operations (distinguished by whether they read or write to public memory), are independent of the input contents (other than their impact on the output size).

As argued in Section 2.5, a level II program can in theory be transformed to a level III program with constant computational overhead as long as no loop conditions depend on the input and the branching depth is constant. Our approach will satisfy these properties and thus yield a program that is secure against many of the side-channel attacks listed in Section 2.3.

3.5 Overview of Approach

If j_1, \dots, j_t are the unique join attribute values appearing at least once in each table, then $T_1 \bowtie T_2$ can be written as the (multiset) union of t groups:

$$T_1 \bowtie T_2 = \bigcup_{i=1}^t \{(d_1, d_2) \mid (j_i, d_1) \in T_1, (j_i, d_2) \in T_2\}.$$

Each group corresponds to a Cartesian product on sets of size

$$\alpha_1(j_i) = |\{(j_i, d_1) \in T_1\}|,$$

and

$$\alpha_2(j_i) = |\{(j_i, d_2) \in T_2\}|,$$

respectively, which we call the *dimensions* of the group.

Each entry $(j_i, d_1) \in T_1$, needs to be matched with $\alpha_2(j_i)$ elements in T_2 ; similarly each element $(j_i, d_2) \in T_2$ needs to be matched with $\alpha_1(j_i)$ elements in T_1 . To this end, and in similar vein to the work of Arasu and Kaushik [4], we form two *expanded* tables S_1 and S_2 (this terminology is borrowed from their paper), each of size $m = |T_1 \bowtie T_2|$, such that there are $\alpha_2(j_i)$ copies in S_1 of each element $(j_i, d_1) \in T_1$ and $\alpha_1(j_i)$ copies in S_2 of each element $(j_i, d_2) \in T_2$. Once the expanded tables are obtained, it only remains to reorder S_2 to align with S_1 so that each copy of $(j_i, d_2) \in T_2$ appears at indices in S_2 that align with each of its $\alpha_1(j_i)$ matching elements from T_1 . At this point, obtaining the final output is simply a matter of iterating through both tables simultaneously and collecting the d values from each pair of rows. See Figure 3.2 for an illustration of the overall idea.

The approach we use to obtain the expanded tables is very simple, relying on an oblivious primitive that sends elements to specified distinct indices in a destination array. Namely, to expand a table T to S we will first *obliviously distribute* each entry of T to the index in S where it ought to first occur; this is achieved by sorting the entries in T by their destination index and then performing $O(m \log m)$ data-independent swaps so that the entries “trickle down” to their assigned indices. To complete the expansion, we then perform a single linear pass through the resulting array to duplicate each non-null entry to the empty slots (containing null entries) that succeed it.

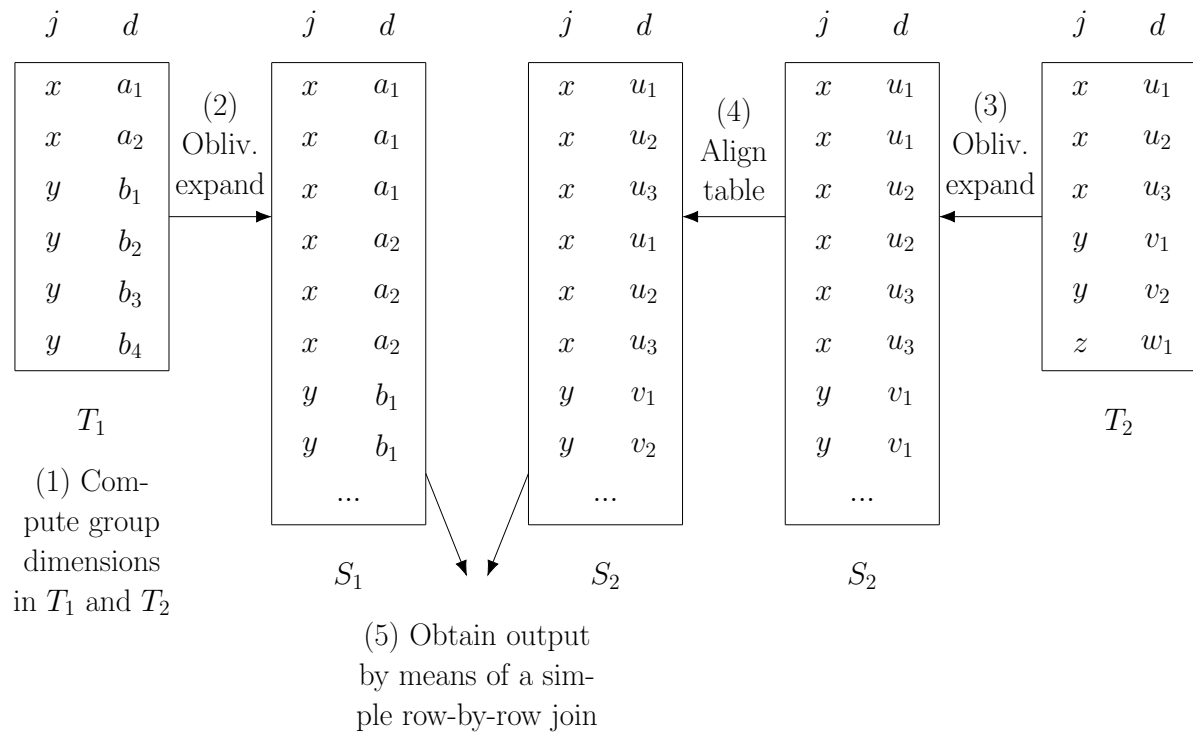


Figure 3.2: Main idea of the algorithm: The input tables T_1 and T_2 are expanded to produce S_1 and S_2 , and S_2 is aligned to S_1 . The output table is then readily obtained by “zipping” together the d values from S_1 and S_2 .

Chapter 4

Algorithm Description

The complete algorithm is outlined in Algorithm 4.1, and its subprocedures are described in the following sections. We use the notation $T(a_1, \dots, a_l)$ when we want to explicitly list the attributes a_1, \dots, a_l of table T , for clarity.

We first call AUGMENT-TABLES to augment each of the input tables with attributes α_1 and α_2 corresponding to group dimensions: this process is described in Section 4.1. Then, as detailed in Section 4.3, we obviously expand T_1 and T_2 into two tables S_1 and S_2 of size m each: namely, S_1 will consist of α_2 (contiguous) copies of each entry $(j, d_1, \alpha_1, \alpha_2) \in T_1$, and likewise S_2 will consist of α_1 copies of each entry $(j, d_1, \alpha_1, \alpha_2) \in T_2$. To achieve this, we rely on the oblivious primitive, OBLIVIOUS-DISTRIBUTE, which is the focus of Section 4.2. After expanding both tables, we call ALIGN-TABLE to align S_2 with S_1 (with the help of the α_1 and α_2 values stored in S_2): this amounts to properly ordering S_2 , as described in Section 4.4. Finally, we collect the data a values from matching rows in S_1 and S_2 to obtain the output table T_D .

4.1 Obtaining Group Dimensions

Before we expand the two input tables, we need to augment them with the $\alpha_1(j_i)$ and $\alpha_2(j_i)$ values corresponding to each join value j_i , storing these in each entry that matches j_i (Algorithm 4.2). To this end, we need to group all entries with common join values together into contiguous blocks, further grouping by them by their table ID. This is achieved by concatenating both tables (augmented with table IDs) together and sorting the result lexicographically by (j, tid) , thus obtaining a table T_C of size $n = n_1 + n_2$.

The α_1 and α_2 values for each group can then be obtained by counting the number of entries originating from table 1 and table 2, respectively. Since such entries appear in

Algorithm 4.1 The full oblivious join algorithm

```

1: function OBLIVIOUS-JOIN( $T_1(j, d), T_2(j, d)$ )
2:    $T_1, T_2(j, d, \alpha_1, \alpha_2) \leftarrow$  AUGMENT-TABLES( $T_1, T_2$ )
3:    $S_1(j, d, \alpha_1, \alpha_2) \leftarrow$  OBLIVIOUS-EXPAND( $T_1, \alpha_2$ )
4:    $S_2(j, d, \alpha_1, \alpha_2) \leftarrow$  OBLIVIOUS-EXPAND( $T_2, \alpha_1$ )
5:    $S_2 \leftarrow$  ALIGN-TABLE( $S_2$ )
6:   initialize  $T_D(d_1, d_2)$  of size  $|S_1| = |S_2| = m$ 
7:   for  $i \leftarrow 1 \dots m$  do
8:      $T_D[i].d_1 \leftarrow S_1[i].d$ 
9:      $T_D[i].d_2 \leftarrow S_2[i].d$ 
10:  return  $T_D$ 

```

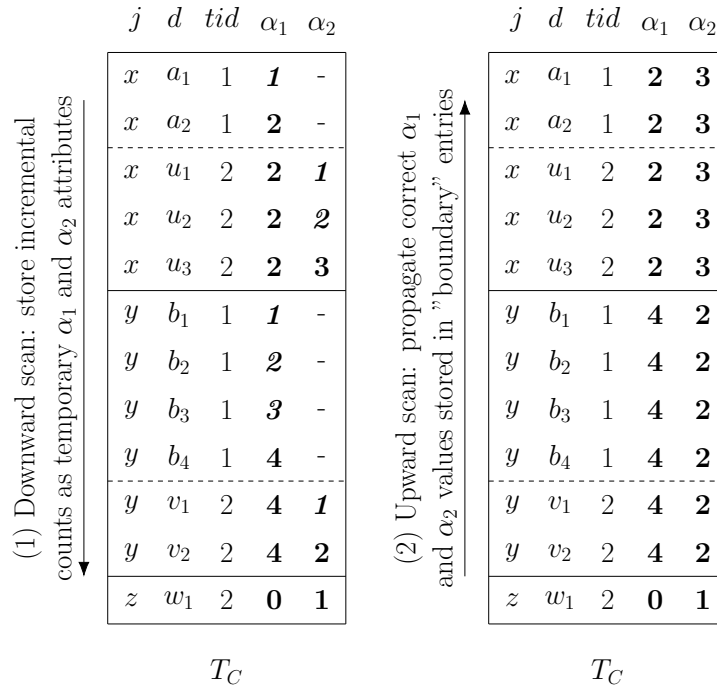


Figure 4.1: Example group dimension calculation. The dimensions of each group can be computed by storing temporary counts during a forward pass through T_C , and then propagating the totals backwards.

Algorithm 4.2 Augment the tables T_1 and T_2 with the dimensions α_1 and α_2 of each entry’s corresponding group. The resulting tables are sorted lexicographically by (j, d) . $n_1 = |T_1|$, $n_2 = |T_2|$, $n = n_1 + n_2$.

```

1: function AUGMENT-TABLES( $T_1, T_2$ )  $\triangleright O(n \log^2 n)$ 
2:    $T_C(j, d, tid) \leftarrow (T_1 \times \{tid = 1\}) \cup (T_2 \times \{tid = 2\})$ 
3:    $T_C \leftarrow \text{BITONIC-SORT}\langle j \uparrow, tid \uparrow \rangle(T_C)$ 
4:    $T_C(j, d, tid, \alpha_1, \alpha_2) \leftarrow \text{FILL-DIMENSIONS}(T_C)$ 
5:    $T_C \leftarrow \text{BITONIC-SORT}\langle tid \uparrow, j \uparrow, d \uparrow \rangle(T_C)$ 
6:    $T_1(j, d, \alpha_1, \alpha_2) \leftarrow T_C[1 \dots n_1]$ 
7:    $T_2(j, d, \alpha_1, \alpha_2) \leftarrow T_C[n_1 + 1 \dots n_1 + n_2]$ 
8:   return  $T_1, T_2$ 

```

contiguous blocks after the sort, this is a matter of keeping count of all entries with the same ID and storing these counts within all entries of the same group; in this manner, we can compute all α_1 and α_2 values in two linear passes through T (one forward and one backward), as shown in Figure 4.1 (full pseudocode for FILL-DIMENSIONS is given in Section A.1). Note that by keeping a sum of the products $\alpha_1 \alpha_2$, we also obtain the output size m , which is needed in subsequent stages.

Take for example the join value x , which corresponds to a group with dimensions $\alpha_1 = 2$ and $\alpha_2 = 3$ (since these are the number of entries with ID 1 and 2, respectively). While encountering entries with $j = x$ and $tid = 1$ during the forward pass, we temporarily store in the α_1 attribute of each entry an incremental count of all previously encountered entries with $j = x$. When we reach entries with $tid = 2$, we can propagate the final count $\alpha_1 = 2$ to all these entries, while starting a new incremental count, stored in α_2 . After iterating through the whole table T_C in this manner, T_C holds correct α_1 and α_2 values in each “boundary” entry (the last entry within a group, such as (x, u_3, \dots)), which can then be propagated to all remaining entries within the same group by iterating through T_C backwards.

It remains for us to extract the augmented T_1 and T_2 from T_C : to accomplish this, we re-sort T_C lexicographically by (tid, j, d) : the first n_1 values of T_C then correspond to T_1 (augmented and sorted lexicographically by (j, d)), the remaining n_2 values correspond to T_2 .

4.2 Oblivious Distribution

We will reduce expansion to a slightly generalized version of the following problem: given an input $X = (x_1, \dots, x_n)$ of n elements each indexed by an injective map $f : X \rightarrow \{1, \dots, m\}$ where $m \geq n$, the goal of OBLIVIOUS-DISTRIBUTE is to store element x_i at index $f(x_i)$ of an array A of size m . Note that for $m = n$, the problem is equivalent to that of sorting obliviously; however for $m > n$, we cannot directly use a sorting network since the output A needs to contain $m - n$ elements that are not part of the output (such as placeholder \emptyset values), and we do not know what indices to assign to such elements so that the x_i appear at their target locations after sorting.

One approach to this problem is probabilistic and requires us to first compute a pseudorandom permutation π of size m . We scan through the n elements, storing element x_i at index $\pi(f(x_i))$ of A . We then use a bitonic sorter to sort the m elements of A by increasing values of π^{-1} applied to each element's index in A . This has the effect of “undoing” the masking effect of the permutation π so that if x_i is stored at index $\pi(f(x_i))$ of A , then as soon as A is sorted, it will appear in its correct destination at index $f(x_i)$ of A . An adversary observing the accesses of this procedure observes writes at locations $\pi(f(x_1)), \dots, \pi(f(x_n))$ of A , followed by the input-independent accesses of the bitonic sorter. Since f is injective, $f(x_1), \dots, f(x_n)$ are distinct and so the values $\pi(f(x_1)), \dots, \pi(f(x_n))$ will correspond to a uniformly-random n -sized subset of $\{1, \dots, m\}$. This approach is therefore oblivious in the sense that the two distributions of memory accesses corresponding to two different inputs are computationally indistinguishable (in this case, they are equal distributions).

The second approach, which we use in our implementation and outlined in Algorithm 4.3 (visualized in Figure 4.2), is deterministic and does not require the use of a pseudorandom permutation, which can be expensive in practice and also introduces an extra cryptographic assumption. This method is similar to the routing network used by Goodrich et al. [22] for tight order-preserving compaction, except here it is used in the reverse direction (instead of compacting elements together it spreads them out). It makes the whole algorithm deterministic (making it easy to empirically test for obliviousness) and has running time $O(n \log^2 n + m \log m)$: the sort takes $O(n \log^2 n)$ time, the outer loop performs $O(\log m)$ iterations, and the inner loop performs $O(m)$. Intuitively, it is oblivious since the loops do not depend on the values of $A[i]$, and though the conditional statement depends on $f(A[i])$, both branches make the same accesses to A .

The idea is to first sort the x_i by increasing destination indices according to f (by using the notation $\text{BITONIC-SORT}\langle f \rangle(A)$, we assume that the f value of each element is stored as an attribute). Each element can then be sent to its destination index by a series of $O(\log m)$ hops, where each hop corresponds to an interval j that is a power of two. For decreasing values of j , we iterate through A backwards and perform reads and writes to

Algorithm 4.3 Obviously map each $x \in X$ to index $f(x)$ of an array of size $m \geq n$, where $f : X \rightarrow \{1 \dots m\}$ is injective.

```

1: function OBLIVIOUS-DISTRIBUTE( $X, f, m$ )
2:    $A[1 \dots n] \leftarrow X$ 
3:   BITONIC-SORT( $f \uparrow$ )( $A$ ) ▷  $O(n \log^2 n)$ 
4:    $A[n + 1 \dots m] \leftarrow \emptyset$  values
5:   extend  $f$  to  $\hat{f}$  such that  $\hat{f}(\emptyset) = 0$ 
6:    $j \leftarrow 2^{\lceil \log_2 m \rceil - 1}$ 
7:   while  $j \geq 1$  do ▷  $O(m \log m)$ 
8:     for  $i \leftarrow m - j \dots 1$  do
9:        $y \xleftarrow{*} A[i]$ 
10:       $y' \xleftarrow{*} A[i + j]$ 
11:      if  $\hat{f}(y) \geq i + j$  then
12:         $A[i] \xleftarrow{*} y'$ 
13:         $A[i + j] \xleftarrow{*} y$ 
14:      else
15:         $A[i] \xleftarrow{*} y$ 
16:         $A[i + j] \xleftarrow{*} y'$ 
17:       $j \leftarrow j/2$ 
18:   return  $A$ 

```

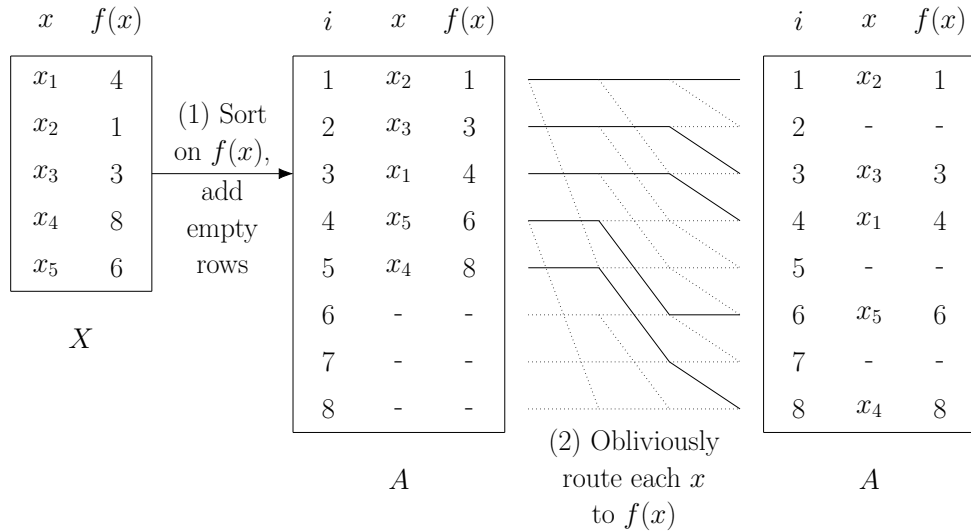


Figure 4.2: Example oblivious distribution with $n = 5$ and $m = 8$. The intermediate step involves sorting the elements by their target indices. The elements are then passed through a routing network, which for $m = 8$ has hop intervals of size 4, 2, and 1.

elements j apart. Most of these will be dummy accesses producing no effect; however, if we encounter an x_i such that x_i can hop down a distance of j and not exceed its target index, we perform an actual swap with the element stored at that location. This will always be a \emptyset element since the non-null elements ahead of x_i make faster progress, as we will formally show. Therefore each x_i will make progress at the values of j that correspond to its binary expansion, and it will never be the case that it regresses backwards by virtue of being swapped with a non-null element that precedes it in A .

In Figure 4.2, for example, each element must move a distance that for $m = 8$, has a binary expansion involving the numbers 4, 2 and 1. No x_i can make a hop of length 4 in this example; however, for the next hop length, 2, element x_4 will advance to index 7, after which element x_5 will advance to index 6 (which at this point corresponds to an empty cell containing a \emptyset value). Finally, for a hop length of 1, element x_4 will advance to index 8, element x_1 will advance to index 4 and element x_3 will advance to index 3, in that order; at this point all the elements will be stored at their desired locations.

We deal with correctness in the following theorem:

Theorem 4.1. *If $m > n$ and $f : \{x_1, \dots, x_n\} \rightarrow \{1, \dots, m\}$ is injective, then OBLIVIOUS-DISTRIBUTE(X, f, m) returns an A such that for $1 \leq i \leq n$, $A[f(x_i)] = x_i$ (and the*

remaining elements of A are \emptyset values).

Proof. Note that after A is initialized, any write to A is either part of a swap or leaves A unchanged; thus at the end of the procedure, A is a permutation of its initial elements and therefore still contains all the n elements of X and $m - n$ \emptyset values. After A is sorted, its first n elements, y_1, \dots, y_n , are the elements of X sorted by their values under f (their destination indices). Note that since f is injective, it follows that for all $i < j$,

$$j - i \leq f(y_j) - f(y_i) \quad (4.1)$$

Let $k = \lceil \log_2 m \rceil - 1$ and let $I_r(y_i)$ be the index of y_i at the end of the r -th outer iteration (for $0 \leq r \leq k + 1$ with $r = 0$ corresponding to the state at the start of the loop). We want to show that for all i , $I_{k+1}(y_i) = f(y_i)$; this will follow from the following invariant: at the end of the r -th outer iteration, we have that for all i ,

$$0 \leq f(y_i) - I_r(y_i) < 2^{k+1-r}.$$

For $r = 0$, the left inequality follows from the fact the y_i are sorted by their values under f and f is injective with minimum value equal 1; the right inequality is simply the bound $f(y_i) - i \leq f(y_i) \leq m < 2^{k+1}$. Assuming the invariant holds at iteration r , we show that it holds at iteration $r + 1$ as well.

Consider all (non-dummy) swaps that happen at iteration $r + 1$ between $y = y_i$ for some i ($y \neq \emptyset$ since $\hat{f}(\emptyset) = 0$) and some element y' at the index $I_r(y_i) + 2^{k-r}$. We show that it must be the case that $y' = \emptyset$. Suppose instead that $y' = y_j$ for some $j \neq i$. Since neither the index of y_i or y_j has at that point exceeded $I_r(y_i) + 2^{k-r} \leq f(y_i) < f(y_j)$, y_i and y_j must have both moved forwards the same total distance throughout the first r iterations and therefore $I_r(y_j) - I_r(y_i) = I_0(y_j) - I_0(y_i) = j - i$. Since we're assuming that this difference shrinks to zero at the $(r + 1)$ -th iteration, it must be the case that

$$I_{r+1}(y_i) = I_r(y_i) + 2^{k-r} = I'_{r+1}(y_j) = I_r(y_j),$$

where $I'_{r+1}(y_j)$ is the index of y_j just before it is swapped with y_i . Therefore

$$j - i = I_r(y_j) - I_r(y_i) = 2^{k-r}.$$

This contradicts the fact that

$$\begin{aligned} j - i &\leq f(y_j) - f(y_i) && \text{(by (4.1))} \\ &\leq f(y_j) - I_r(y_j) && (I_r(y_j) = I_{r+1}(y_i) \leq f(y_i)) \\ &< 2^{k-r} && \text{(since } I'_{r+1}(y_j) = I_r(y_j)) \end{aligned}$$

It follows that no two elements y_i and y_j are ever swapped, and therefore:

$$I_{r+1}(y_i) = \begin{cases} I_r(y_i) + 2^{k-r}, & I_r(y_i) + 2^{k-r} \leq f(y_i) \\ I_r(y_i), & \text{otherwise.} \end{cases}$$

We can now show that for $i < j$

$$0 \leq f(y_i) - I_{r+1}(y_i) < 2^{k-r}.$$

If $I_r(y_i) + 2^{k-r} \leq f(y_i)$, then

$$f(y_i) - I_{r+1}(y_i) = f(y_i) - I_r(y_i) - 2^{k-r} \geq 0,$$

and

$$\begin{aligned} f(y_i) - I_{r+1}(y_i) &= f(y_i) - I_r(y_i) - 2^{k-r} \\ &< 2^{k-r+1} - 2^{k-r} \\ &= 2^{k-r}. \end{aligned}$$

If $I_r(y_i) + 2^{k-r} > f(y_i)$, then

$$\begin{aligned} f(y_i) - I_{r+1}(y_i) &= f(y_i) - I_r(y_i) \geq 0 \\ f(y_i) - I_{r+1}(y_i) &= f(y_i) - I_r(y_i) < 2^{k-r}, \end{aligned}$$

which finishes the proof of the invariant.

It then follows from the invariant that $I_{k+1}(y_i) = f(y_i)$, and so when the $k+1$ iterations of the outer loop complete, each y_i will appear in its correct index according to f . \square

4.3 Oblivious Expansion

OBLIVIOUS-EXPAND takes an array $X = (x_1, \dots, x_n)$ and a function g on X which assigns non-negative integer counts to each x , and outputs

$$A = (\underbrace{x_1, \dots, x_1}_{g(x_1) \text{ times}}, \underbrace{x_2, \dots, x_2}_{g(x_2) \text{ times}}, \dots).$$

This can easily be achieved using OBLIVIOUS-DISTRIBUTE (see Figure 4.3) if we assume $m \geq n$ and $g(x_i) > 0$ for all x_i : we compute the cumulative sum $f(x_i) = 1 + \sum_{j=1}^{i-1} x_j$,

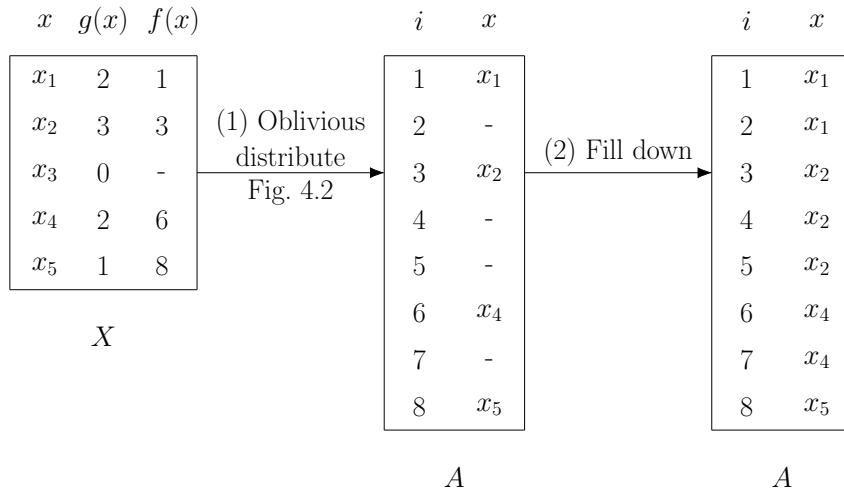


Figure 4.3: Example oblivious expansion. This is achieved by obviously distributing each element to where it ought to first appear and then scanning through the resulting array to duplicate each entry in the null slots that follow.

and obviously distribute the x_i according to f (in practice, the values of f are stored as attributes in augmented entries). The resulting array A is such that each x_i is stored in the first location that it needs to appear in the output of OBLIVIOUS-EXPAND; the next $g(x_i) - 1$ values following x_i are all \emptyset . Thus we only need to iterate through A , storing the last encountered entry and using it to overwrite the \emptyset entries that follow.

To account for the possibility that $g(x_i) = 0$ for certain x_i (which means that m may possibly be less than n), we simply need to modify OBLIVIOUS-DISTRIBUTE to take as input an n -sized array X such that the subset X' of X of entries not marked as \emptyset has size $n' \leq m$ and $f' : X' \rightarrow \{1 \dots m\}$ is injective. The output will be an array A with each $x_i \in X'$ stored at index $f(x_i)$ of A ; the remainder of A will consist of \emptyset values as before. This modified version of OBLIVIOUS-DISTRIBUTE (EXT-OBLIVIOUS-DISTRIBUTE) will allow OBLIVIOUS-EXPAND to mark entries x_i with $g(x_i) = 0$ as \emptyset (done in practice by first making sure the entries are augmented with an extra flag bit for this purpose) to the effect that they can be discarded by EXT-OBLIVIOUS-DISTRIBUTE, as shown in Algorithm 4.4.

Algorithm 4.4 Obviously duplicate each $x \in X$ $g(x)$ times.

```

1: function OBLIVIOUS-EXPAND( $X, g$ )
2:    $\triangleright$  obtain  $f$  values and distribute according to  $f$ 
3:    $s \leftarrow 1$ 
4:   for  $i \leftarrow 1 \dots n$  do  $\triangleright O(n)$ 
5:      $x \stackrel{*}{\leftarrow} X[i]$ 
6:     if  $g(x) = 0$  then
7:       mark  $x$  as  $\emptyset$ 
8:     else
9:       store  $f(x) = s$  at  $x$ 
10:     $s \leftarrow s + g(x)$ 
11:     $X[i] \stackrel{*}{\leftarrow} x$ 
12:   $A \leftarrow$  EXT-OBLIVIOUS-DISTRIBUTE( $X, f, s - 1$ )
13:   $\triangleright$  fill in missing entries
14:   $px \leftarrow \emptyset$ 
15:  for  $i \leftarrow 1 \dots s - 1$  do  $\triangleright O(m)$ 
16:     $x \stackrel{*}{\leftarrow} A[i]$ 
17:    if  $x = \emptyset$  then
18:       $x \leftarrow px$ 
19:    else
20:       $px \leftarrow x$ 
21:     $A[i] \stackrel{*}{\leftarrow} x$ 
22:  return  $A$ 
23:
24: function EXT-OBLIVIOUS-DISTRIBUTE( $X, f, m$ )
25:   $A[1 \dots n] \leftarrow X$ 
26:  BITONIC-SORT( $\neq \emptyset \uparrow, f \uparrow$ )( $A$ )  $\triangleright O(n \log^2 n)$ 
27:  if  $m \geq n$  then
28:     $A[n + 1 \dots m] \leftarrow \emptyset$  values
29:  extend  $f$  to  $\hat{f}$  such that  $\hat{f}(\emptyset) = 0$ 
30:  continue as in  $O(m \log m)$  loop of Algorithm 4.3...
31:  return  $A[1 \dots m]$ 

```

Algorithm 4.5 Reorder S_2 so that its m entries align with those of S_1 .

```

1: function ALIGN-TABLE( $S_2$ )
2:    $S_2(j, d, \alpha_1, \alpha_2, ii) \leftarrow S_2(j, d, \alpha_1, \alpha_2) \times \{ii = \text{NULL}\}$ 
3:   for  $i \leftarrow 1 \dots |S_2|$  do  $\triangleright O(m)$ 
4:      $e \stackrel{*}{\leftarrow} S_2[i]$ 
5:      $q \leftarrow$  (0-based) index of  $e$  within block for  $e.j$ 
6:      $e.ii \leftarrow \lfloor q/e.\alpha_2 \rfloor + (q \bmod e.\alpha_2) \cdot e.\alpha_1$ 
7:      $S_2[i] \stackrel{*}{\leftarrow} e$ 
8:    $S_2 \leftarrow$  BITONIC-SORT( $j, ii$ )( $S_2$ )  $\triangleright O(m \log^2 m)$ 
9:   return  $S_2$ 

```

4.4 Table Alignment

Recall that S_1 is obtained from T_1 based on the counts stored in α_2 since for each entry $(j_i, d_1) \in T_1$, $\alpha_2(j_i)$ is the number of entries in T_2 matching j_i , and these are all the entries that (j_i, d_1) must be matched with. Likewise S_2 is obtained from T_2 based on the counts stored in α_1 . It remains for us to properly align S_2 to S_1 so that each output entry corresponds to a row of S_1 and a row of S_2 with matching index. More precisely, we need to sort S_2 so that the sequence of pairs $\{(S_1[i].d_1, S_2[i].d_2)\}_{i=1}^m$ is a lexicographic ordering of all the pairs in $T_1 \bowtie T_2$. For example, in Figure 4.4, the $\alpha_2(x) = 2$ copies of (x, u_1) in S_2 , need to be matched with $\alpha_2(x) = 2$ entries from T_1 : (x, a_1) and (x, a_2) . Since the entries in S_1 occur in blocks of size $\alpha_1(x) = 3$, this means that the copies of (x, u_1) in S_2 need to occur a distance of $\alpha_1(x) = 3$ apart: at indices 1 and 4 in S_2 . In general, these indices can be computed from the α_1 and α_2 attributes, as outlined in Algorithm 4.5. Note that q is simply a counter that is reset when a new join value is encountered, similarly to the counter c in Algorithm 4.2.

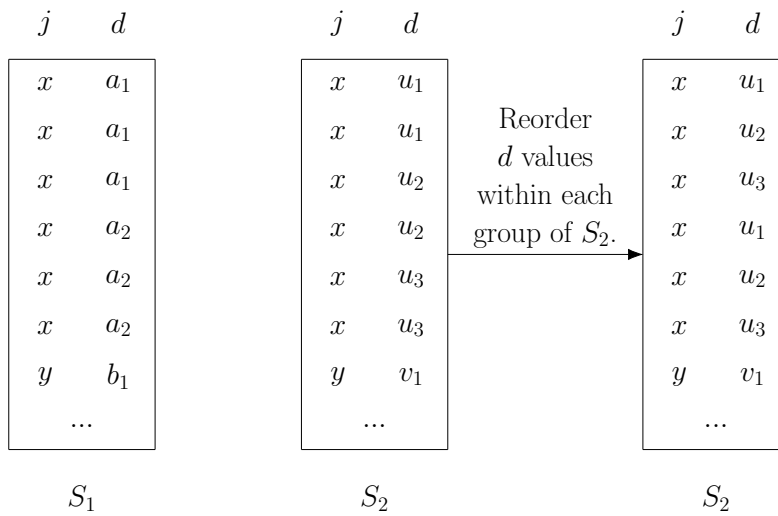


Figure 4.4: Example table alignment. S_2 is reordered to align with S_1 . In this example, each of the two copies of (x, u_1) in S_2 ends up appearing at two indices matching both (x, a_1) and (x, a_2) from S_1 ; the same applies to the copies of (x, u_2) and (x, u_3) .

Chapter 5

Evaluation

We implemented a (sequential) C++ prototype of the general algorithm, which we then readily adapted as an SGX application whose entire execution takes place within the enclave (code available at <https://git.uwaterloo.ca/skrastni/obliv-join-impl>). We empirically tested for correctness on varying input sizes n (10 to 1,000,000): for each n , we automatically generated 20 tests consisting of various different inputs of size n (for instance, one inducing n 1×1 groups, one inducing a single $1 \times n$ group, and several where the group sizes were drawn from a power law distribution). In the remainder of this chapter, we evaluate the security and performance of the implementation.

5.1 Security Analysis

We verified the obliviousness of our prototype both formally, through the use of a dedicated type system, and empirically, by comparing the logs of array accesses for different inputs. To ensure that the actual low-level memory accesses were also oblivious, we transformed it as per Section 2.5 and inspected its accesses using an instrumentation tool.

Verification of Obliviousness through Typing

Liu et al. [31] showed that programming language techniques can be used to verify the obliviousness of programs. The authors formally define the concept of memory trace obliviousness, and define a type system in which only programs satisfying this property are well-typed. We adapted a simplified version of their system that does not incorporate the use of ORAM (since we do not use any), and which corresponds to level II obliviousness in accordance with our high-level description in the previous chapter.

The type system is presented in Figure 5.1, in a condensed format. Each type is a pair of the form $\tau; T$, where τ is either $\text{Var } l$, $\text{Array } l$, or a statement, and T is a corresponding *trace*. In the case when τ is $\text{Var } l$ or $\text{Array } l$, the label l is either L (“low” security) if the variable or array stores input-independent data, or H (“high” security) otherwise. The ordering relation on labels, $l_1 \sqsubseteq l_2$, is satisfied when $l_1 = l_2 = L$, or $l_1 = L$ and $l_2 = H$. We define $l_1 \sqcup l_2$ to be H if at least one of l_1 or l_2 is H and L otherwise. In an actual program, we would set to L the label of variables corresponding to the values of n and m , and set to H the label of all allocated arrays that will contain input-dependent data (in our program all arrays are such). The trace T is a sequence of memory accesses $\langle R, y, i \rangle$ (reads) or $\langle W, y, i \rangle$ (writes), where y is the accessed array and i is the accessed index. We use ϵ to denote an empty trace and \parallel to denote the concatenation operator.

All judgements for expressions are of the form $\Gamma \vdash \text{exp} : \tau; T$, where Γ is an environment mapping variables and arrays to types, exp is an expression and τ is its type, and T is the trace produced when evaluating exp . Judgments for statements are of the form $\Gamma \vdash s; T$.

Note that all rules that involve reads and writes to only Var types emit no trace since they model our notion of local memory. The rule T-Asgn models the flow of high-security data: a variable x that is the target of an assignment involving an H variable y must always be labeled H . The rules T-Read and T-Write are similar to T-Asgn but also ensure two other properties: that arrays are always indexed by variables labeled L (for otherwise the memory access would leak data-dependent data), and that the reads and writes to arrays emit a trace consisting of the corresponding memory access. The two rules that play an important role in modeling obliviousness are T-Cond, which ensures that the two branches of any conditional statement emit the same memory traces, and T-For, which ensures the number of iterations of any loop is a low-security variable (such as a constant, n , or m).

We manually verified that our implementation is well-typed in this system by annotating the code with the correctly inferred types. For example, every if statement was annotated with the matching trace of its branches.

Experiments: Memory Access Logs

In our prototype all contents of (heap-allocated) memory that correspond to public memory — all except a constant number of variables such as counters and those used to store the results of a constant number of read entries — are accessed through a wrapper class which is used to keep a log of such accesses. For small n ($n \leq 10$), we manually created different test classes (around 5), where each test class corresponds to values of n_1 and n_2 (summing to n), and an output length m . We verified, by direct comparison, that the memory access logs for each of the inputs in the same class were identical. Figure 5.2 visualizes the full sequence of memory accesses for $n_1 = n_2 = 4$ and $m = 8$.

$$\begin{array}{c}
\text{T-Var} \frac{\Gamma(x) = \text{Var } l}{\Gamma \vdash x : \text{Var } l; \epsilon} \quad \text{T-Const} \frac{}{\Gamma \vdash \text{Var } L; \epsilon} \\
\\
\text{T-Op} \frac{\Gamma \vdash x : \text{Var } l_1; \epsilon \quad \Gamma \vdash y : \text{Var } l_2; \epsilon}{\Gamma \vdash x \text{ op } y : \text{Var } l_1 \sqcup l_2; \epsilon} \\
\\
\text{T-Asgn} \frac{\Gamma(x) = \text{Var } l_1; \epsilon \quad \Gamma \vdash y : \text{Var } l_2; \epsilon \quad l_2 \sqsubseteq l_1}{\Gamma \vdash x \leftarrow y; \epsilon} \\
\\
\text{T-Read} \frac{\Gamma(y) = \text{Arr } l' \quad l' \sqsubseteq l \quad \Gamma \vdash i : \text{Var } L; \epsilon \quad \Gamma \vdash x : \text{Var } l; \epsilon}{\Gamma \vdash x \stackrel{*}{\leftarrow} y[i]; \langle R, y, i \rangle} \\
\\
\text{T-Write} \frac{\Gamma(y) = \text{Arr } l' \quad l \sqsubseteq l' \quad \Gamma \vdash i : \text{Var } L; \epsilon \quad \Gamma \vdash x : \text{Var } l; \epsilon}{\Gamma \vdash y[i] \stackrel{*}{\leftarrow} x; \langle W, y, i \rangle} \\
\\
\text{T-Cond} \frac{\Gamma \vdash c : \text{Var } l; \epsilon \quad \Gamma \vdash s_1; T \quad \Gamma \vdash s_2; T}{\Gamma \vdash \mathbf{if } c \mathbf{ then } s_1 \mathbf{ else } s_2; T} \\
\\
\text{T-For} \frac{\Gamma \vdash t : \text{Var } L; \epsilon \quad \Gamma \vdash s; T}{\Gamma \vdash \mathbf{for } i \leftarrow 1 \dots t \mathbf{ do } s; \underbrace{T || \dots || T}_{t \text{ copies}}} \\
\\
\text{T-Seq} \frac{\Gamma \vdash s_1; T_1 \quad \Gamma \vdash s_2; T_2}{\Gamma \vdash s_1; s_2; T_1 || T_2}
\end{array}$$

Figure 5.1: Summary of type system used to model level II obliviousness and verify implementation.

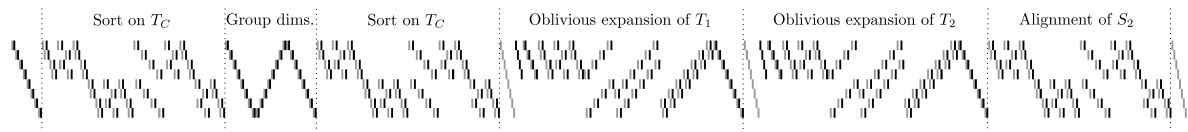


Figure 5.2: Visualization of our implementation’s input-independent pattern of memory access as it joins two tables of size 4 into a table of size 8. Horizontal axis is (discretized) time, vertical axis is the memory index; light shade denotes a read; dark denotes a write.

For larger values of n where the logs were too large to fit in memory, we kept a hash of the log instead. That is, we set $H = 0$, and for every access to an index i of an array A_r allocated by our program, we updated H as follows:

$$H \leftarrow h(H || r || t || i),$$

where h is a cryptographic hash function (SHA-256 in our case) and t is 0 or 1 depending on whether the access is a read or a write to A_r . With n ranging from 10 to 10,000, we generated a diverse range of tests, in the manner described at the beginning of this chapter, but also under the restriction that the tests for each n produce outputs of the same size. We verified that for each n the tests produced the same hash.

Experiments: Memory Trace Instrumentation

Through a mix of manual and automated code transformations similar to those outlined in Section 2.5, we obtained a program where all virtual memory accesses of the program are oblivious. To verify this we ran the same hash-based tests as previously described except that the target memory accesses were obtained by using Intel’s Pin instrumentation framework to inject the hash computation at every program instruction involving a memory operand. The verification was successful when the program was compiled with GCC 7.5.0 with an `-O2` optimization level (whereas `-O3` did not preserve the intended properties of our transformation).

5.2 Performance Analysis

Taking into account the fact that performing a bitonic sort on input n makes roughly $n(\log_2 n)^2/4$ comparisons, the cost breakdown of the full algorithm is summarized in Table 5.1, which supports the fact that our time complexity of $O(n \log^2 n + m \log m)$ does not hide large constants that would make the algorithm impractical.

In terms of space usage, the total (non-oblivious) memory we use is $\max(n_1, m) + \max(n_2, m)$ entries since the table T_C has size $n_1 + n_2$, the augmented tables T_1 and T_2 correspond to two regions of T_C , and the expanded tables S_1 and S_2 can be obtained from T_1 and T_2 by only allocating as many extra entries as needed to expand T_1 and T_2 to tables of size m (if one of the original tables has size less than m , then no extra entries will be allocated for that table’s expansion).

We ran the different variants of our implementation on a single core of an Intel Core i5-7300U 2.60 GHz laptop with 8 GB RAM; the runtime of the prototype, the SGX version, and the transformed SGX version is shown in Figure 5.3 and compared to a non-oblivious sort-merge join. The performance gap between the implementation and sort-merge can be explained by the additional logarithmic factor in the former’s complexity, as well as the overhead from having to store and process auxiliary attributes used by the oblivious primitives.

Since our SGX versions exclusively use the limited Enclave Page Cache (EPC) of size approximately 93 MiB for all allocated memory, we anticipate a drop in performance for input sizes where the EPC size is insufficient (due to swapping). However, this size is expected to be increased considerably in future versions of SGX.

The only related join algorithm with an implementation that has been evaluated on input sizes up to 10^6 is the one proposed by Opaque, which we remind is restricted to primary-foreign key joins. Its SGX implementation, despite being evaluated on better hardware and on multiple cores, runs approximately five times slower for an input size of $n = 10^6$.

Although our implementation is non-parallel, almost all parts of our algorithm are amenable to parallelization since they heavily rely on sorting networks, whose depth is $O(\log^2 n)$. The only exception is the sequence of $O(m \log m)$ operations following the sorts in each of the two calls to OBLIVIOUS-DISTRIBUTE. However, as is shown in Table 5.1, these operations account for a negligibly small fraction of the total runtime.

Subroutine	Comparisons	Runtime
initial sorts on T_C	$n(\log_2 n)^2/2$	60%
o.d. on T_1, T_2 (sort)	$n_1(\log_2 n_1)^2/2$	25%
o.d. on T_1, T_2 (route)	$2m \log_2 m$	3%
align sort on S_2	$m(\log_2 m)^2/4$	12%
total (when $m \approx n_1 = n_2$)	$n(\log_2 n)^2 + n \log_2 n$	100%

Table 5.1: Comparison count and runtime share for each component of the algorithm. For each (non-linear) component of the algorithm: approximate counts of total comparisons (or swaps) when $m \approx n_1 = n_2$, as well as empirical share of total implementation runtime for $n = 10^6$.

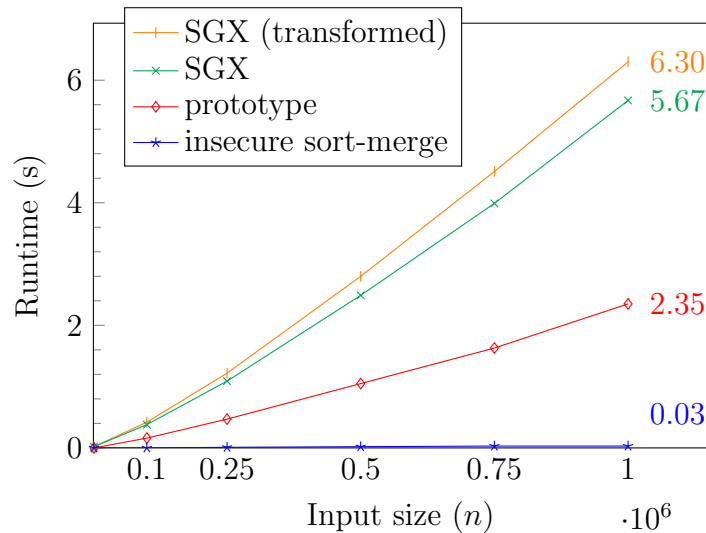


Figure 5.3: Performance results for sequential prototype implementation. The inputs are such that $m \approx n_1 = n_2 = n/2$.

Chapter 6

Conclusions and Future Work

Our algorithm for oblivious equi-joins has optimal runtime in the sense that its time complexity is determined by that of oblivious sorting, an algorithmic necessity for the problem. Its reliance on sorting networks, which are characterized by a small total operation count, means that its complexity does not hide prohibitively large constants, and we show by means of a prototype implementation that this translates into practical efficiency. Our approach has low circuit complexity and introduces novel data-independent techniques for query processing. There is an increasing demand for such approaches due to their resistance against side-channel attacks and suitability for secure computation.

In the remainder of this chapter, we give an outline of several directions for future work, including the consideration of other types of join-based queries, other types of oblivious algorithms, and the design of compilers for oblivious programs.

6.1 Compound Join Queries

In multi-way equi-joins, there are k tables T_1, \dots, T_k (again thought of as having join and data attributes), and we wish to compute

$$T_1 \bowtie \dots \bowtie T_k = \bigcup_{i=1}^t \{(d_1, \dots, d_k) \mid (j_i, d_1) \in T_1, \dots, (j_i, d_k) \in T_k\}.$$

One solution is to associatively apply our binary equi-join algorithm k times. This is problematic not only because its runtime depends on k but also because it leaks the intermediate table sizes $|T_1 \bowtie \dots \bowtie T_i|$ for $1 < i < k$, whereas a formulation of the problem in the data-oblivious setting allows leakage of only the final table size $|T_1 \bowtie \dots \bowtie T_k|$. However, Arasu and Kaushik [4] argue that if a multiway join satisfies a property called *acyclicity*, then it can be computed obliviously and efficiently (they also show, using

a hardness argument, that this is not possible for cyclic joins). Such approach can be combined with the oblivious primitives constructed in this thesis. The same applies to other queries considered in that paper such as grouping aggregations over joins, in which case the goal is to optimize the query so that a full join is not necessary.

6.2 Non Equi-Joins

This thesis focuses on equi-joins, where the join predicate is equality. A natural question is whether the ideas extend to more general predicates. For instance, in a band join $T_1 \overset{R}{\bowtie} T_2$ of two tables, the join attribute values j_i are integers, and an entry $(j_1, d_1) \in T_1$ is joined with an entry $(j_2, d_2) \in T_2$ whenever j_2 is within a “band” R of j_1 , that is, $|j_1 - j_2| \leq R$. Even more generally, one can consider spatial joins, where the j_i are vertices in two or more dimensions. It is an interesting question whether the techniques from this thesis can be extended to these types of joins.

6.3 Other Oblivious Algorithms

The primitives constructed in this thesis, for instance oblivious distribution and expansion, could potentially be useful in providing a general framework for oblivious algorithm design and possibly have direct applications in various different problem areas with similar security goals.

As an example of an application unrelated to the main problem considered in this thesis, consider a scenario where a client would like access to n memory cells stored on an untrusted server, and suppose that any sequence of accesses requested by the client is dominated by writes (it requests, on average, $O(n)$ writes for every read). For example, the client may need to run an experiment whereby it frequently updates statistical data in a database of users and collects the results only at the end of the experiment. Though one can use ORAM in such a scenario, the oblivious distribute primitive offers a much simpler and practically efficient solution. Namely, we can construct a primitive called write-heavy ORAM (not to be confused with write-only ORAM, which supports fast reads but does not hide them) that has an amortized $O(\log^2 n)$ overhead per write and $O(n)$ overhead per read.

The construction is as follows: when a request for a write is received, a record consisting of the index, data, and sequence number of the write is appended to a log, separate from the n memory cells. When the size of the log becomes n , the records in the log are sorted by increasing values of their indices, followed by increasing sequence numbers. The sorted

log is then scanned to invalidate all entries except those with the latest sequence number for a given index. Finally, oblivious distribute is used to route the data in the valid entries to their intended location within the n memory cells. Reads are implemented by scanning through the whole log of n entries to find the latest record with the given index, or if no such record exists, proceeding to scan all n memory cells.

6.4 Obliviousness-Preserving Compilers

Though a program may be designed to be oblivious in theory, achieving actual security guarantees in practice is a difficult language- and platform-dependent problem. Though the source code of the program may seem to be data-independent, a standard compiler provides no guarantees that such property is preserved, as it is free to apply optimizations that transform the code. Even if this is not the case, the target architecture may process data-independent machine code in a data-dependent way, as is the case in out-of-order processors. In addition, there is the question of how to enforce and verify the data-obliviousness property on the source code level in the first place.

There have been several works that elucidate and attempt to address these problems. Molnar et al. [34] employ a source-to-source transformation approach to guarantee a *straight-line* property of code, in the spirit of the ideas outlined in Section 2.5. However, they point out that gcc compiler does not preserve such property, having discovered behaviours like “[the compilation] of the logical negation operator into an assembly sequence involving a conditional branch”. Coppens et al. [10] describe several different properties of a program that potentially influence its timing behaviour: not only its control flow, but also its *data flow* (for example, in certain architectures the division instruction can exit early depending on the values of its operands), and the tendency of the out-of-order target processor to reorder instructions. They argue that a source-to-source approach is incapable of accounting for all these properties, and propose a way to guarantee input-dependent control flow on x86 processors by generating code for if-else statements in a way that makes use of the conditional move instruction available for that architecture.

Other related works in this direction include: ConfLLVM [8], an LLVM-based compiler that aims to prevent information flow violations in the presence of an active adversary; Raccoon [37], a tool that uses transactional memory to obfuscate code and defend it against side-channel attacks; OISA [46], an instruction set architecture extension, which supports low-level data-oblivious instructions.

Despite such abundance of research on the implementation-specific details of oblivious program design, unfortunately there does not yet exist a unified tool which allows a programmer to write a (possibly annotated or restricted) program in a standard language and

have it compiled to an executable with strong guarantees. A worthy starting goal would be the development of a tool that takes LLVM IR (intermediate representation), checks for data-dependent control flow, and upon detection of a branching instruction tainted by private data, attempts to transform the code (for instance by generating IR to execute both branches and then combining the results using a select IR instruction). This approach would allow the programmer to use the language of their choice (since LLVM compilers exist for a plethora of languages), and would give them some freedom over how they handle the private input data, such as allowing the use of certain if-else statements with tainted conditions.

References

- [1] Ittai Abraham, Christopher W Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing ORAM with PIR. In *IACR International Workshop on Public Key Cryptography*, volume 10174 of *Lecture Notes in Computer Science*, pages 91–120. Springer, 2017.
- [2] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, and Yaping Li. Sovereign joins. In *22nd International Conference on Data Engineering*, pages 26–26. IEEE, 2006.
- [3] Abdelrahman Aly, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P Smart, and Tim Wood. SCALE–MAMBA v1. 3 documentation. <https://homes.esat.kuleuven.be/~nsmart/SCALE>, 2019.
- [4] Arvind Arasu and Raghav Kaushik. Oblivious query processing. In *Proceedings of the 17th International Conference on Database Theory*, pages 26–37. Springer, 2014.
- [5] Todd W Arnold, Carl Buscaglia, Felix Chan, Vincenzo Condorelli, John Dayka, William Santiago-Fernandez, Nihad Hadzic, Michael D Hocker, Michael Jordan, Thomas E Morris, et al. IBM 4765 cryptographic coprocessor. *IBM Journal of Research and Development*, 56(1.2):10–1, 2012.
- [6] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, pages 307–314. ACM, 1968.
- [7] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. SMCQL: secure querying for federated databases. *Proceedings of the VLDB Endowment*, 10(6):673–684, 2017.
- [8] Ajay Brahmakshatriya, Piyus Kedia, Derrick P McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. ConFLVM: A

- compiler for enforcing data confidentiality in low-level code. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15. ACM, 2019.
- [9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies*, 2017.
 - [10] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2009.
 - [11] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. <https://eprint.iacr.org/2016/086>.
 - [12] Jack Doerner and Abhi Shelat. Scaling ORAM for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535, 2017.
 - [13] David Eppstein, Michael T Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *Proceedings of the 18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 13–22, 2010.
 - [14] Saba Eskandarian and Matei Zaharia. ObliDB: oblivious query processing for secure databases. In *Proceedings of the VLDB Endowment*, volume 13, pages 169–183. VLDB Endowment, 2019.
 - [15] International Organization for Standardization. Information technology – Trusted platform module library – Part 1: Architecture. Technical Report ISO/IEC TR 11889-1:2015, 2015.
 - [16] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.
 - [17] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *International Symposium on Privacy Enhancing Technologies*, pages 1–18. Springer, 2013.
 - [18] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 182–194, 1987.

- [19] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge University Press, 2009.
- [20] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 307–328. Morgan & Claypool, 2019.
- [21] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
- [22] Michael T Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 379–388, 2011.
- [23] Michael T Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 684–693, 2014.
- [24] Michael T Goodrich and Joseph A Simons. Data-oblivious graph algorithms in outsourced external memory. In *International Conference on Combinatorial Optimization and Applications*, pages 241–257. Springer, 2014.
- [25] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6. ACM, 2017.
- [26] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. S3ORAM: A computation-efficient and constant client bandwidth blowup ORAM with Shamir Secret Sharing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 491–505, 2017.
- [27] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium*, page 12, 2012.
- [28] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. In *Proceedings of the VLDB Endowment*, volume 13, pages 2132–2145, 2020.

- [29] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium*, pages 557–574, 2017.
- [30] Yaping Li and Minghua Chen. Privacy preserving joins. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1352–1354, 2008.
- [31] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 51–65, 2013.
- [32] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376, 2015.
- [33] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy*, pages 279–296, 2018.
- [34] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology*, pages 156–168. Springer, 2005.
- [35] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. *ACM Transactions on Privacy and Security*, 21(2):1–35, 2018.
- [36] Raluca Ada Popa, Catherine MS Redfield, Nikolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.
- [37] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium*, pages 431–446, 2015.
- [38] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium*, pages 72–75, 2014.

- [39] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xi-angyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and communications security*, pages 299–310, 2013.
- [40] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium*, pages 1041–1056, 2017.
- [41] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–18. ACM, 2019.
- [42] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.
- [43] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861, 2015.
- [44] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.
- [45] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE, 1986.
- [46] Jiyong Yu, Lucas Hsiung, Mohamad El’Hajj, and Christopher W Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *Network and Distributed System Security Symposium*, 2019.
- [47] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *2013 IEEE Symposium on Security and Privacy*, pages 493–507, 2013.
- [48] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: efficient random access in

multi-party computation. In *2016 IEEE Symposium on Security and Privacy*, pages 218–234, 2016.

- [49] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation*, pages 283–298, 2017.

APPENDICES

Appendix A

Additional Pseudocode

A.1 Group Dimension Computation

```
1: function FILL-DIMENSIONS( $T_C$ ) ▷  $O(n)$ 
2:    $c \leftarrow 0$ 
3:    $pe \leftarrow$  dummy entry with  $j = tid = \infty$ 
4:   for  $i \leftarrow 1 \dots n$  do
5:      $e \stackrel{*}{\leftarrow} T_C[i]$ 
6:     if  $e.tid = 1$  and  $e.j \neq pe.j$  then
7:        $c \leftarrow 1$ 
8:        $e.\alpha_1 \leftarrow c$ 
9:     else if  $e.tid = 1$  and  $e.j = pe.j$  then
10:       $c \leftarrow c + 1$ 
11:       $e.\alpha_1 \leftarrow c$ 
12:     else if  $e.tid = 2$  and  $e.j \neq pe.j$  then
13:       $c \leftarrow 1$ 
14:       $e.\alpha_1 \leftarrow 0$ 
15:       $e.\alpha_2 \leftarrow c$ 
16:     else if  $e.tid = 2$  and  $e.j = pe.j$  then
17:       $c \leftarrow c + 1$  if  $pe.tid = 2, 1$  otherwise
18:       $e.\alpha_1 \leftarrow pe.\alpha_1$ 
19:       $e.\alpha_2 \leftarrow c$ 
20:      $pe \leftarrow e$ 
21:      $T_C[i] \stackrel{*}{\leftarrow} e$ 
22:    $pe \leftarrow$  dummy entry with  $j = tid = \infty$ 
```

```

23: for  $i \leftarrow n \dots 1$  do
24:    $e \stackrel{\star}{\leftarrow} T_C[i]$ 
25:   if  $e.tid = 1$  and  $e.j \neq pe.j$  then
26:      $e.\alpha_2 \leftarrow 0$ 
27:   else if  $e.tid = 1$  and  $e.j = pe.j$  then
28:      $e.\alpha_1 \leftarrow pe.\alpha_1$ 
29:      $e.\alpha_2 \leftarrow pe.\alpha_2$ 
30:   else if  $e.tid = 2$  and  $e.j = pe.j$  then
31:      $e.\alpha_2 \leftarrow pe.\alpha_2$ 
32:    $pe \leftarrow e$ 
33:    $T_C[i] \stackrel{\star}{\leftarrow} e$ 
34: return  $T_C$ 

```