Fall 10-14-2018

libOblivious: A C++ Library for Oblivious Data Structures and Algorithms

Scott D. Constable
Syracuse University, sdconsta@syr.edu

Steve Chapin
Syracuse University, chapin@syr.edu

libOblivious: A C++ Library for Oblivious Data Structures and Algorithms

Scott Constable
Syracuse University

Steve Chapin
Syracuse University

Abstract

Infrastructure as a service (IaaS) is an enormously beneficial model for centralized data computation and storage. Yet, existing network-layer and hardware-layer security protections do not address a broad category of vulnerabilities known as side-channel attacks. Over the past several years, numerous techniques have been proposed at all layers of the software/hardware stack to prevent the inadvertent leakage of sensitive data. This report discusses a new technique which integrates seamlessly with C++ programs. We introduce a library, libOblivious, which provides thin wrappers around existing C++ standard template library classes, endowing them with the property of memory-trace obliviousness.

Contents

List of Figures

List of Tables

Listings

1 Introduction

The recent proliferation of infrastructure as a service (IaaS) has allowed big data computations to be offloaded into the cloud. While this confers computational and cost benefits to small companies, large companies, research organizations, and end users, it has also been plagued by concerns over data privacy. Technologies such as the secure sockets layer (SSL) ensure confidentiality and integrity as sensitive data is transferred to or from the cloud. New hardware technologies such as Intel SGX [2] and ARM TrustZone [4] use strategies ranging from encryption to process isolation to protect data while it being used or stored in the cloud. Yet these techniques are wholly inadequate to protect sensitive data from side-channel attacks.

A side channel is an inadvertent revelation of information, which can be monitored by external observers. Side channels fall into one of several interrelated categories by which they signal information: implicit flows, termination channels, timing channels, probabilistic channels, resource exhaustion channels, and power channels [42].

Many contemporary techniques for defeating or mitigating side-channel attacks are deployed at a low level, for instance by modifying a compiler [20, 31], instrumenting binaries [41], or dynamically shuffling memory access patterns [43]. These solutions use computationally expensive program trace obfuscation techniques. Hardware-based solutions [33] may introduce less overhead, but are not cross-platform, and typically target only one kind of attack.

At a higher level, recent software-layer solutions have deployed data-oblivious computation strategies. A program is said to be *data oblivious* if the adversary's view of its data memory accesses (i.e. reads and writes) does not vary with respect to secret program data [23, 37]. An even more secure property is *memory-trace obliviousness* [30, 31, 41], which additionally covers the trace of instruction fetches made by a program. In this report we often use the truncated term *obliviousness* in place of memory-trace obliviousness.

The definition of "secret" program data is often subjective. In this report, we generally treat data values, such as values in a container or a value which determines a search query, as secret. We treat metadata, such as the number of values in a container, as non-secret. So an oblivious search algorithm must not leak the value of the search query or any of the values in the dataset being searched, though it may leak the size of the dataset, or the number of queries on the dataset.

This report describes libOblivious, a C and C++ library which exports APIs to facilitate oblivious computation at the software level. We must emphasize the word "facilitate" because other solutions [30, 31, 41, 53] attempt to guarantee either data- or memory-trace obliviousness for an entire program. libOblivious, on the other hand, provides operations and algorithms which are locally memory-trace oblivious. For example, a call to a libOblivious search algorithm will not leak the value of the search query or any value in the container being searched through either the memory access pattern or the algorithm's control flow. It is up to the user to ensure that his/her own code does not leak secrets elsewhere.

The libOblivious comprises four memory-trace oblivious parts: primitives, such as copying and memory accesses; algorithms built on the primitives; containers (e.g., linked lists, arrays, etc.); and iterators. These last two parts were not built from scratch. Rather, we define

wrappers around C++ standard template library (STL) containers and iterators which endow their operations with the property of memory-trace obliviousness. This allows our implementation to be light-weight, easy to use, and flexible. libOblivious is supported and has been tested on Linux, macOS, and Windows 64-bit operating systems with Intel x86-64 CPUs.

Adversary Model. We assume that the adversary—by means of a side channel or otherwise—may observe the sequence of cache blocks touched by the victim program. This assumption covers both instruction fetch accesses and data accesses (reads and writes). We also assume that the user has deployed libOblivious in the manner recommended above, so as to obfuscate memory accesses and control flow branching conditions influenced by secret data. Under this assumption, libOblivious can guarantee memory-trace oblivious program execution.

Roadmap. The remainder of the report is structured as follows. Section 2 discusses the problem of side-channel attacks in more depth and reviews recent work on oblivious computation. Section 3 outlines a popular machine learning algorithm, and suggests how it can be made memory trace oblivious. Sections 4 and 5 describe the design and implementation of libOblivious. We provide an in-depth analysis of libOblivious' performance compared to the STL in Section 6. Section 7 describes some contemporary related work on oblivious computation. We discuss our next set of implementation goals for libOblivious in Section 8. Finally, we state our conclusions in Section 9.

2 Background

With the advent of distributed computations taking place in the cloud, side-channel attacks have increasingly become a threat to user privacy [9, 34, 36]. For example, many users may want to deploy a popular data aggregation algorithm in the cloud, such as MapReduce [17]. If the user wishes to protect her sensitive data, and she does not trust the cloud provider, then she could use a secure (e.g. encryption-protected) implementation of MapReduce, such as VC3 [44]. However, cloud applications protected only by encryption in the cloud have been demonstrably vulnerable to side-channel attacks [36].

The recently discovered Spectre-style attacks exploit a common CPU optimization feature called speculative execution [27]. This vulnerability was so severe that numerous software products, including operating systems and web browsers, needed to be patched immediately [39, 40]. Moreover, portions of commodity CPUs are still being redesigned with new protection schemes [50]. More recent research has even shown that side-channel attacks can be launched across networks [45].

2.1 Side-Channel Attacks on the Memory Hierarchy

The work in this report addresses a specific category of side-channel attacks: attacks on the memory hierarchy. Xu et. al. demonstrated on an Intel SGX platform that an adversary with control over the untrusted operating system running on that platform can observe

the sequence of pages in memory touched by an SGX enclave program [52]. By constantly evicting the enclave program's pages from memory, the adversary can force each attempted memory access to trigger a page fault, thus notifying the adversary which page has been touched.

On x86 platforms the page size is 4 KB. The page resolution of this side-channel attack may be sufficient for attacks on applications which operate on large datasets, such as images [52]. Side-channel attacks have also been demonstrated at much finer resolution. Specifically, Liu et. al. demonstrated that last-level cache attacks are possible on multi-tenant (i.e. multiple VMs) server platforms [34]. Their implementation uses the PRIME+PROBE [38] technique to "prime" a CPU cache with code or data, then wait for the victim tenant to execute for some fixed time interval, and finally "probe" by measuring the time required to access memory within the blocks that had been primed. If a probe on a block that had been primed is slower after the victim has run, then that block must have been evicted by the victim.

Our work in this report targets side-channel attacks at the cache level. However, since the cache blocks on x86 are 64 bytes wide (and aligned to 64 bytes), the attack window on caches is strictly a finer resolution than the attack window on page faults, thus our memory-trace oblivious technique also defeats the page fault attack.

2.2 Memory-Trace Oblivious Computation

We first formalize the notion of memory-trace obliviousness that was introduced briefly in Section 1. Let $\mathcal{M}$ be a model of a machine controlled by the adversary. The adversary can use $\mathcal{M}$ to run a program P with secret and non-secret inputs I_{Secret} and I_{Public}, respectively. Non-secret inputs are visible to the adversary; secret inputs are not visible. On a concrete machine, I_{Secret} and I_{Public} may correspond to data in memory or in CPU registers, or data read from a file, `stdin`, etc. Execution of a program on $\mathcal{M}$ emits a memory trace τ, denoted $\mathcal{M}(P, I_{Secret}, I_{Public}) \rightsquigarrow \tau$. The trace τ is a sequence of memory addresses of length $|\tau|$.

Definition 1. The adversary's observational power is characterized by a relation $\approx_{Adv}$ such that for some positive integer constant n, $\tau \approx_{Adv} \tau'$ if:

1. $|\tau| = |\tau'|$, and
2. all corresponding addresses in τ and τ' are equivalent modulo n.

We refer to n as the *granularity* of $\approx_{Adv}$[1].

Example 1. An adversary using the forced page fault strategy [52] to observe τ has an observational granularity of 4 KB.

Example 2. An adversary using the cache block PRIME+PROBE strategy [34, 38] to observe τ has an observational granularity of 64 bytes.

[1] Thus $\tau \approx_{Adv} \tau'$ if $\tau = \tau'$, but $\tau \approx_{Adv} \tau'$ does not necessarily imply $\tau = \tau'$.

Definition 2. A contiguous region of memory with size and alignment equal to the granularity of $\approx_{Adv}$ is called the *mask* of $\approx_{Adv}$.

For instance, the mask of $\approx_{Adv}$ in Example 1 is an x86 4 KB page, and in Example 2 it is a cache block. Intuitively, the mask is the smallest contiguous unit of memory within which the adversary cannot distinguish between accesses at different memory addresses.

Definition 3. A program P is *memory trace oblivious* for $\approx_{Adv}$ if, for all I_{Public}, I_{Secret}, and I'_{Secret},

$$\mathcal{M}(P, I_{Secret}, I_{Public}) \rightsquigarrow \tau \text{ and } \mathcal{M}(P, I'_{Secret}, I_{Public}) \rightsquigarrow \tau'$$

implies $\tau \approx_{Adv} \tau'$.

That is, for any fixed non-secret input, any variation in the secret input must not affect the program trace visible to the adversary. Hence the adversary cannot deduce anything about the values of the secret inputs by observing the program traces.

Theorem 1. *Let $\approx_{Adv_1}$ and $\approx_{Adv_2}$ characterize the observational power of two adversaries with granularity n and m, respectively. If m is divisible by n, then a program P which is memory trace oblivious for $\approx_{Adv_1}$, is also memory trace oblivious for $\approx_{Adv_2}$.*

Proof. Suppose that P is memory trace oblivious for $\approx_{Adv_1}$, and let I_{Secret} and I'_{Secret} be arbitrary secret input sequences accepted by P, and likewise have I_{Public} as an arbitrary public input sequence accepted by P. Hence if

$$\mathcal{M}(P, I_{Secret}, I_{Public}) \rightsquigarrow \tau \text{ and } \mathcal{M}(P, I'_{Secret}, I_{Public}) \rightsquigarrow \tau',$$

then $\tau \approx_{Adv_1} \tau'$. By Definition 1, $|\tau| = |\tau'|$ and all corresponding addresses in τ and τ' are equivalent modulo n. Since m is divisible by n, it follows that all corresponding addresses in τ and τ' are also equivalent modulo m. Thus $\tau \approx_{Adv_2} \tau'$. Because I_{Secret}, I'_{Secret}, and I_{Public} were arbitrary, it follows from Definition 3 that P is memory trace oblivious for $\approx_{Adv_2}$. □

Example 3. By Theorem 1, a program which is memory-trace oblivious against an adversary using the cache block PRIME+PROBE strategy [34, 38] is also memory-trace oblivious against an adversary using the forced page eviction strategy [52], because the x86 page size is divisible by the x86 block size.

Memory-trace oblivious computations must not allow secret data to influence memory accesses in two ways: instruction fetches (via control flow) and data accesses (reads and writes). This implies that secret data must not be used as a loop termination condition or any other branch, such as an `if`/`else if`/`else` or `switch`. This rule also extends to branching operators (ternary `?:`) and short-circuiting operators: `&&` and `||`.

This may at first seem too restrictive for the developer. However, many algorithms can be expressed in terms of ternary operators. A non-branching ternary operator can be constructed using the `cmov` family of instructions on x86 [15]. This technique has been utilized in related projects [37, 41, 43].

```
1  int32_t read(const int32_t *I, const int32_t *E,
2               const int32_t *addr) {
3    int32_t ret = 0;
4    while (I++ != E) {
5      ret = o_copy(I == addr, *I, ret);
6    }
7    return ret;
8  }
9
10 int32_t write(const int32_t *I, const int32_t *E,
11                int32_t *addr, int32_t val) {
12   while (I++ != E) {
13     *I = o_copy(I == addr, *I, val);
14   }
15 }
```

Listing 1: Naïve memory-trace oblivious read and write operations

With a non-branching ternary operator (call this operation `o_copy()`), it becomes possible to write oblivious read and write operations. One such solution is given in Listing 1. The `read()` function iterates through the entire region of memory spanning the range $[\texttt{I}, \texttt{E})$, reading from every address (at 4-byte granularity). Only when the current address matches the argument `addr` is the value of `ret` updated to the value of `*I`, which at that point is equal to the value of `*addr`. This heuristic does not leak the value of `addr` because every value in $[\texttt{I}, \texttt{E})$ is read exactly once. Later in this report, we describe how many kinds of common algorithms can be built using just a few oblivious primitives such as the non-branching ternary.

Within the context of our adversary model for x86-64, the machine $\mathcal{M}$ does not emit a trace at byte granularity. Because we assume that the adversary can only observe which cache block is touched on a given access, and not the specific address within the block, our model of $\mathcal{M}$ emits a memory trace with 64-byte granularity. Thus the solution in Listing 1 is naïve for our adversary model because we do not actually need to read from every element in the array. It suffices to read only one value from each cache block within a range of memory, rather than read every single value [13]. Listing 1 could then be modified to read one integer from each cache block covered by $[\texttt{I}, \texttt{E})$, e.g. it could read every 16^{th} integer before and after `addr`.

An even more efficient solution is to vectorize the read. New x86-64 CPUs provide AVX2 instructions which operate on 256-bit vector registers [15]. In particular, AVX2 introduced the `vpagther` instruction family. A single `vpgatherdd` instruction can read eight 32-bit integers from non-contiguous addresses in memory. As shown in Figure 1, we can use `vpgatherdd` to touch eight cache blocks per instruction. We have used primitives built on `vpgatherdd` to implement a family of oblivious read and write primitives. This vectorized oblivious read strategy has also been deployed in related work [37].

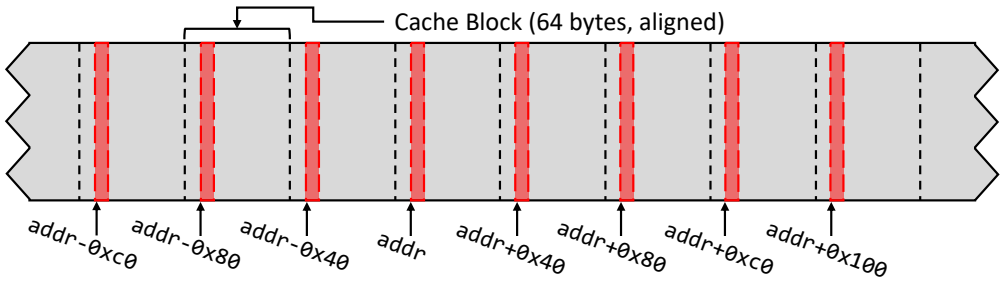

Figure 1: Oblivious read using `vpgatherdd`

```
1  INPUT: z, k, classes, d, T = {(x_i, y_i)}_{i=1,...,n}
2    Find T_Near = {(x_i, y_i)}_{i=1,...,k} and T_Far = {(x_i, y_i)}_{i=1,...,n-k} such that:
3      T = T_Near ∪ T_Far and T_Near ∩ T_Far = ∅ // T_Near and T_Far partition T
4      ∀(x, y) ∈ T_Near, (x', y') ∈ T_Far. d(y, z) ≤ d(y', z)
5    For c in classes:
6      Set count[c] := 0
7    For all (x, y) ∈ T_Near:
8      Set count[x] = count[x] + 1
9    Find c ∈ classes such that ∀c' ∈ classes. count[c'] ≤ count[c]
10 OUTPUT: c
```

Listing 2: The k-nearest neighbors (k-NN)

3 A Motivating Example: k-Nearest Neighbors

One popular application of cloud computing is machine learning. One simple, yet powerful classification algorithm is the k-nearest neighbors algorithm [16]. In brief, k-NN is a non-parametric classification algorithm where the program inputs consist of a training set and a test set. Each set has a series of data points, each of which is characterized by a sequence of attributes. A generic implementation of k-NN would also accept a measure of distance d between data points as a parameter. For each data point z in the test set, k-NN finds its k nearest "neighbors" in the training set (using d). The class c assigned to z by k-NN is the mode of the classes of its k nearest neighbors. If the neighbors have no mode (e.g. two classes are equally represented among the k neighbors), then k-NN must use some heuristic (e.g. randomness) to choose the class for z.

Listing 2 describes the algorithm for a k-NN classifier in more detail. The k-NN classifier uses a training set T to classify a data point z. The training set consists of pairs (x_i, y_i), where each y_i is a data point, and each x_i is the classification (some element of *classes*) assigned to y_i. The algorithm proceeds in three steps. The first step is to partition T into T_{Near} and T_{Far}: the k data points nearest to z (according to d), and the remaining $n-k$ data points, respectively. Next, the classes of the data points in T_{Near} are tallied. The classifier assigns to z the class which has won the highest tally among z's nearest neighbors. The k-NN algorithm leaves unspecified how a tie between class tallies should be settled.

The procedures in each of these three steps have the potential to leak data through a side channel. Typically, the first step would be implemented by sorting the data structure containing T and selecting the first k elements. Common sorting algorithms such as quick sort, merge sort, and insertion sort all have control flow patterns which do depend on the ordering relationships between container elements. If the class tallying is performed with any kind of dictionary-like data structure, as Listing 2 would suggest, then each key lookup would almost certainly leak the value of the key. Finally, determining the class with the highest tally could leak any or all of the tallies. Consider this typical implementation of a function to find the maximum value in a container:

```
1  template<typename It>
2  It find_largest(It first, It last)
3  {
4      if (first == last) return last;
5
6      It largest = first;
7      ++first;
8      for (; first != last; ++first) {
9          if (*largest < *first) { // leaks the ordering relationship
10                                  // between ‘largest‘ and ‘first‘
11             largest = first;
12         }
13     }
14     return largest;
15 }
```

The relationship between the current element and the largest element found so far is leaked in line 9, where the `<` relation influences a branch condition.

In general, it is difficult to write memory-trace oblivious algorithms from scratch, which is why it is essential to provide programmers with tools to facilitate memory-trace oblivious programming. libOblivious is one such tool.

4 libOblivious Overview

libOblivious is composed of four components to facilitate memory-trace oblivious programming in C and C++: (1) oblivious primitives, such as oblivious copy and swap operations; (2) oblivious container types, e.g. vectors, linked lists, and deque-like structures; (3) oblivious iterators which can be used to obliviously read/write from/to the oblivious container; and (4) oblivious algorithms. The next four subsections provide an overview of these features in more detail. The last subsection demonstrates how these components can be used in concert to implement a memory-trace oblivious k-NN algorithm in C++.

4.1 libOblivious Primitives

At the time of this writing, we provide four groups of oblivious primitives: oblivious copies, swaps, reads, and writes. An oblivious copy moves data from one of two sources, depending on a Boolean condition, to a single destination. The destination and sources could be any

C Prototype	Register Map	Implementation

`int64_t o_copy_i64(`
`  int cond,`
`  int64_t left,`
`  int64_t right);`
	cond ⇒ ecx
	left ⇒ rdx
	right ⇒ r8
		`mov rax, rdx`
		`test ecx, -1`
		`cmovz rax, r8`

`void o_swap_i64(`
`  int cond,`
`  int64_t *left,`
`  int64_t *right);`
	cond ⇒ ecx
	left ⇒ rdx
	right ⇒ r8
		`test ecx, -1`
		`mov r10, QWORD PTR [r8]`
		`mov r9, QWORD PTR [rdx]`
		`mov r11, r9`
		`cmovnz r9, r10`
		`cmovnz r10, r11`
		`mov QWORD PTR [rdx], r9`
		`mov QWORD PTR [r8], r10`

`int64_t o_read_i64(`
`  const void *src_base,`
`  size_t src_size,`
`  const int64_t *addr,`
`  bool base_aligned);`
	—	—

`void o_write_i64(`
`  void *dst_base,`
`  size_t dst_size,`
`  int64_t *addr,`
`  int64_t val,`
`  bool base_aligned);`
	—	—

Table 1: 64-bit libOblivious primitives

combination of memory locations or CPU registers. Semantically, an oblivious copy is like a ternary (`?:`) operator, except that the value of the Boolean condition does not leak through a side channel. An oblivious swap, like the oblivious copy, swaps the contents of two memory locations (or two registers) if the given Boolean condition is true, and it does not leak the value of that condition. An oblivious read/write from/to a region of memory does not leak the address within that region where the access was made. More complex algorithms such as those discussed in Section 4.4 can be built on top of these primitives.

Table 1 gives a summary of the 64-bit versions of these operations. For `o_copy_i64()` and `o_swap_i64()` we provide the x86-64 assembly implementations. This is our implementation for Windows, hence it uses Windows x86-64 procedure call conventions [8] and Microsoft macro assembler [7] (MASM) syntax (which itself uses Intel assembly syntax [3]). These operations utilize the x86 `cmov` family of instructions (as described in Section 2) to conditionally move data from one register to another—without requiring a branch operation.

The implementations of `o_read_i64()` and `o_write_i64()` are much more complex, but both use the `vpgather` instructions in a manner similar to that which was described

in Section 2. One difference is that for 64-bit reads and writes, we use the `vpgatherqq` instruction, which reads four 64-bit words from four (possibly) non-contiguous addresses in memory. As demonstrated later in Section 6, this modification doubles our read and write throughput.

Each oblivious write actually performs one read and one write per mask. In order for the write to be memory-trace oblivious, each mask in the given memory region(s) must be written to once. This is trivial for the mask containing the actual destination address for the write. But for every other mask, we need to perform a write without corrupting any data. Hence for every mask, we first read a value from a particular address in each mask, and then for each address which is not the destination address, we write back the same value which was read.

For backwards compatibility with older CPUs that do not support AVX2, libOblivious can be configured to perform scalar oblivious reads and writes instead of the vectorized versions. The scalar implementation is up to 2x slower than the vectorized implementation.

The oblivious copy and swap operations come in 8, 16, 32, 64, and 256-bit versions, along with generic `o_copy()` and `o_swap()` functions which copy or swap an arbitrary number of bytes in a single operation. Oblivious reads and writes come in 32 and 64-bit versions and also have generic `o_read()` and `o_write()` functions. For non-contiguous data structures such as linked lists and deques, libOblivious also provides `o_read_list()` and `o_write_list()` operations which accept a list of memory regions from which to obliviously read or write data.

When the number of bytes to be copied, swapped, read, or written is large, libOblivious vectorizes these operations, dramatically increasing throughput without violating the property of memory-trace obliviousness. Details on these optimizations are given in Section 5.3.

Of the four components of libOblivious, this is the only component which provides an interface for C programs. Moreover, since the C language does not use name mangling, other languages with a C foreign function interface (e.g. Python) can use these libOblivious primitives.

When libOblivious is linked into C++ code, additional generic templatized primitives are exposed: `o_copy_T()`, `o_swap_T()`, `o_read_T()`, `o_write_T()`, `o_read_list_T()`, and `o_write_list_T()`. These are easier to use than the pure C APIs, because they automatically deduce the correct number of bytes to copy/swap/read/write from the types of the function arguments.

4.2 libOblivious Containers

We provide oblivious wrappers for C++ STL containers. The containers supported by libOblivious include:

- Arrays (`std::array`)
- Singly linked lists (`std::forward_list`)
- Doubly linked lists (`std::list`)

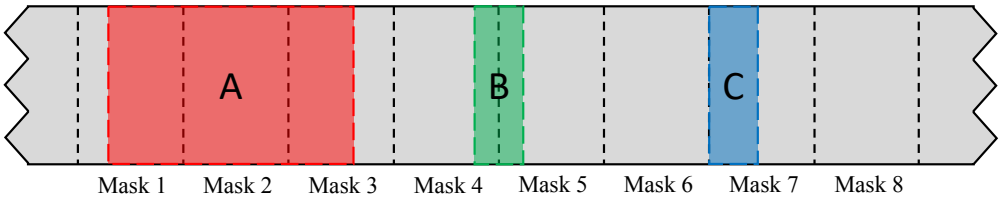

Figure 2: Objects in memory covering masks

- Deques (`std::deque`)
- Queues (`std::queue`)
- Stacks (`std::stack`)
- Vectors (`std::vector`)

Currently, the oblivious containers are just type aliases for the STL containers, except that they use stateful heap allocators to maintain a record of which regions of the heap they own. More details on the stateful heap allocators are given in Section 5.1. We are currently planning to provide memory-trace oblivious member functions for these containers (see Section 8).

Notice that all of the supported containers are sequential, i.e. containers that can be accessed by sequential iteration or random access. Associative containers such as maps (e.g. red-black trees) and unordered maps (e.g. hash tables) are not supported because these structures use key-value lookup to access container elements in a manner which is not memory trace oblivious.

It is still possible to simulate associative containers with sequence containers. For example, a singly linked list can store a key-value pair in each node (e.g. using the `std::pair` template). Associative lookup can be performed using the `ofind_if` algorithm (see Section 4.4), where the given predicate returns `true` when a node's key matches a given query. Unlike ordinary associative lookup in `std::map` and `std::unordered_map`, this operation has $O(n)$ complexity, as opposed to $O(log\ n)$ or $O(1)$ for the STL associative containers.

In fact, $O(n)$ complexity is the best we can hope to achieve for any search algorithm over an oblivious container when the search criteria are secret. The following theorem establishes this fact. We introduce some additional notation: $\mathcal{I}_{Secret}$ is the set of all possible sequences of secret inputs, and likewise $\mathcal{I}_{Public}$ for non-secret input sequences. We denote the *image* of a function $f : A \to B$ as:

$$f^{\rightarrow} = \{f(x) \mid x \in A\},$$

and similarly for any subset $S \subseteq A$:

$$f^{\rightarrow}(S) = \{f(x) \mid x \in S\}.$$

In the following theorem, function f serves as the lookup function, mapping program input to the address of a container element. We use the term *access* in this context to refer

to any combination any combination of memory reads or writes on an object. We say that an object in memory *covers* one or more masks (recall Definition 2) when any portion of the object's contiguous memory lies within those masks. For example, in Figure 2 object A covers masks 1, 2, and 3, B covers masks 4 and 5, and C covers mask 7.

Theorem 2 (Oblivious Lookup)**.** *Let C be a container with n elements. Let $f : \mathcal{I}_{Public} \to \mathcal{I}_{Secret} \to C$ be a function mapping program input to locations of elements in C. Let P be a memory-trace oblivious program which uses f to locate and then access some element in C. Let $I_{Public} \in \mathcal{I}_{Public}$, and let $f_{Secret} = f\ I_{Public}$ (i.e. $f_{Secret} : \mathcal{I}_{Secret} \to C$). For all $I_{Secret} \in \mathcal{I}_{Secret}$, if $\mathcal{M}(P, I_{Secret}, I_{Public}) \rightsquigarrow \tau$, then τ must have recorded at least $\Omega(\,|f^{\to}_{Secret}|\,)$ accesses on the masks covered by elements of C.*

Proof. Let $\{e_1, \ldots, e_m\} = f^{\to}_{Secret}$, where $m = |f^{\to}_{Secret}|$. Partition $\mathcal{I}_{Secret}$ into $\mathcal{I}^1_{Secret}, \ldots, \mathcal{I}^m_{Secret}$ such that for all i from 1 to m, $f^{\to}_{Secret}(\mathcal{I}^i_{Secret}) = \{e_i\}$. Let $I_{Secret}, I'_{Secret} \in \mathcal{I}_{Secret}$ be arbitrary, and note that there exist $i, j \in \{1, \ldots, m\}$ such that $I_{Secret} \in \mathcal{I}^i_{Secret}$ and $I'_{Secret} \in \mathcal{I}^j_{Secret}$. Suppose

$$\mathcal{M}(P, I_{Secret}, I_{Public}) \rightsquigarrow \tau \text{ and } \mathcal{M}(P, I'_{Secret}, I_{Public}) \rightsquigarrow \tau'.$$

Since P is memory-trace oblivious, we must have $\tau \approx_{Adv} \tau'$. Let g be the granularity of $\approx_{Adv}$, let s be the size of each element in C (container elements must be of uniform size in C++). If $i = j$, then τ and τ' both recorded accesses on at least $\left\lceil \frac{s}{g} \right\rceil$ masks covered by e_i. If $i \neq j$, then τ and τ' both recorded accesses on at least $\left\lceil \frac{2 \cdot s}{g} \right\rceil$ masks covered by e_i and e_j.

In general, the traces emitted by P over secret input sequences from k distinct subsets of the partition $\mathcal{I}^1_{Secret}, \ldots, \mathcal{I}^m_{Secret}$ must record accesses on at least $\left\lceil \frac{k \cdot s}{g} \right\rceil$ masks covered by k elements of C. Since we must consider accesses over the entire partition, each trace must record at least

$$\left\lceil \frac{m \cdot s}{g} \right\rceil = \left\lceil \frac{|f^{\to}_{Secret}| \cdot s}{g} \right\rceil \in \Omega(\,|f^{\to}_{Secret}|\,)$$

accesses on the masks covered by elements of C. □

Example 4. Consider the algorithm `ofind()`, an oblivious implementation of `std::find()`. The `ofind()` function takes as parameters two iterators into a container which define the range over which to search, and the value to find; it returns an iterator I to the first element in the container which matches the search query. Suppose that the search query value is secret, and that the iterators defining the range of the search cover only the first half of a vector of size n. If the caller then uses I to access the element that was looked up, Theorem 2 requires that $\Omega(n/2) = \Omega(n)$ additional elements in C must be accessed in order to maintain memory-trace obliviousness, and thus not leak the value of the secret query.

Example 5. Consider the function `std::list::begin()`, which returns an iterator I to the first element of a given list. `std::list::begin()` is actually a degenerate lookup function

whose image is a singleton set consisting of the first element of the list. By Theorem 2, if a caller uses I to access the first list element, then $\Omega(1)$ accesses must be performed. In fact, only the one access must be performed, hence `std::list::begin()` is invariably memory trace oblivious.

Corollary 1. *Let C be a container with n elements. Let $f : \mathcal{I}_{Secret} \to C$ be a surjective function mapping program input to locations of elements in C. Let P be a memory-trace oblivious program which uses f to locate and then access some element in C, and let $I_{Public} \in \mathcal{I}_{Public}$. For all $I_{Secret} \in \mathcal{I}_{Secret}$, if $\mathcal{M}(P, I_{Secret}, I_{Public}) \rightsquigarrow \tau$, then τ must have recorded at least $\Omega(n)$ accesses on the masks covered by elements of C.*

Proof. This is simply a special case of Theorem 2 with $f^{\rightarrow}_{Secret} = C$. □

Example 6 (Oblivious Read)**.** Consider the primitive subscript operation on an array, e.g. `arr[42]`. Assuming that out-of-bounds accesses are disallowed, the subscript operation is a lookup operation which maps an integral index to an array element, returning a reference to that element. If the array has n elements, then by Corollary 1 $\Omega(n)$ accesses are required to access the returned reference, assuming the index is secret.

On a particular concrete machine and with a particular adversary model, we can make the argument in Example 6 more precise. Recall the discussion about oblivious reads/writes from Section 2. For an adversary whose observational power $\approx_{Adv}$ has cache block granularity on x86-64, our strategy was to read one element (e.g. a 32-bit integer) from each cache block to obfuscate the access at a secret address or index. Recall from the proof of Theorem 2 that we constructed a precise lower bound on the number of accesses required for a memory-trace oblivious lookup: $\left\lceil \frac{n \cdot s}{g} \right\rceil$ when the lookup function's image is the entire container holding n elements. Thus to defeat $\approx_{Adv}$, a 32-bit oblivious read on an array of size n must make at least $\left\lceil \frac{n \cdot 4}{64} \right\rceil = \left\lceil \frac{n}{16} \right\rceil$ accesses. The oblivious memory access strategy we employ for `o_read_32()` and `o_write_32()` performs precisely this many accesses for any array of size n.

4.3 libOblivious Iterators

When operating on elements in a container, C++ conventions recommend the use of iterators, rather than pointers [48]. An iterator is typically implemented as a wrapper around a pointer, but with semantics more appropriate for its associated container. For instance, an iterator to a linked list may overload the `++` operator so that instead of incrementing the underlying pointer, the underlying pointer is advanced to the next link in the list.

All iterators must define the indirection (`*`) operator, which is used to access the container element to which the iterator points [25]. Random access iterators, which operator over sequential containers that support random access operations, also provide a subscript (`[]`) operator. Random access iterators support all of the arithmetic operations (addition, pointer difference, etc.) that are supported by ordinary pointers.

libOblivious provides its own iterators, which are themselves wrappers around C++ STL iterators or ordinary pointers. The `O` template wraps an iterator or a pointer, and endows

```
1  // return the first value less than `x` in `list`, or -1 if
2  // there is no such value
3  int find_less_than(const olist<int> &list, int x) {
4    auto finder = [x](int val) { return val < x; };
5    O i = ofind_if(list.begin(), list.end(), finder, &list);
6    if (i == list.end()) // value less than `x` not found
7      return -1;
8    else // return the value we found
9      return *i; // performs an oblivious read
10 }
```

Listing 3: Using an `O` iterator to obliviously read an `int`

it with memory-trace oblivious read and write operations to the container into which the iterator points. The `O` template also must wrap a reference to the container, effectively binding each `O` iterator to a specific container. The reason for this design decision is discussed later in Section 5.

Listing 3 shows an example which uses an `O` iterator to obliviously read from a linked list element whose address was discovered by an oblivious find (`ofind_if()` is described in the next section). Failure to obliviously read from the iterator `i` could leak the value of `x` and/or the values of elements in `list`.

In general, `O` iterators conform to the C++ standard requirements for iterators, though there are two noteworthy exceptions. First, section 27.2.5 the C++17 standard states that the `reference` type member of a forward iterator (an input iterator that can be used in multipass algorithms) must be a reference to the value type of the associated container [25]. When a forward iterator is dereferenced using the `*` operator, the return type is `reference`. The `O` template uses the temporary proxy pattern [5], a topic which we cover in greater detail in Section 5.2. One feature of the temporary proxy pattern is that it uses a proxy object—rather than a reference—to access container elements. So the `*` and `[]` operators for `O` iterators do not conform to the `reference` type requirement for forward iterators.

Second, section 27.2.3 of the C++17 standard requires that for an iterator `a` which satisfies the requirements for an input iterator (an iterator that can read from the pointed-to element), the expression `a->m` must be valid with semantics `(*a).m` [25]. The `O` template does not support the `->` operator.

4.4 libOblivious Algorithms

The goal of the libOblivious algorithms library is to provide memory-trace oblivious implementations of all algorithms supported by the C++ STL algorithms library [1, 25]. These algorithms are built on top of the memory-trace oblivious primitives that were introduced in Section 4.1. Algorithms which use oblivious iterators (see: the `O` template, Section 4.3) can only be applied over the libOblivious containers (Section 4.2). All other oblivious algorithms can be applied to any sequential container in the C++ STL.

```
1  template <class InputIt, class UnaryPredicate,
2            class ContainerT>
3  O<InputIt, ContainerT>
4  ofind_if(InputIt first, InputIt last,
5      UnaryPredicate p, ContainerT *container) {
6    InputIt ret = last;
7    for (; first != last; ++first) {
8      o_copy_T(ret, p(*first) & (ret == last), first, ret);
9    }
10   return {ret, container};
11 }
```

Listing 4: `ofind_if` implementation

Listing 4 shows the implementation of `ofind_if()`, one of the foundational search algorithms of the libOblivious algorithms library. Its semantics correspond to those of `std::find_if()` [25]. The interface is identical to that of `std::find_if()`, except that `ofind_if()` also requires a reference to the container to be searched. This is necessary in order to construct the oblivious iterator return value, as discussed in Section 4.3. We use `o_copy_T()` to obliviously update the return value when the first match is found. Also noteworthy is the use of a bitwise AND (`&`) instead of the logical AND (`&&`) in the copy condition. This subtle distinction is important because the logical AND in C and C++ has short-circuiting behavior which may or may not compile into a branch: the second operand will only be evaluated if the first operand evaluates to `true`. Thus, if the first operand has been influenced by secret data, that information could be leaked. Finally, notice that the `for` loop does not break when a match is found. The algorithm always examines every single element in the container once. Other algorithms such as `ofind()` and `oany_of()` are implemented on top of `ofind_if()`.

Table 2 summarizes the algorithms which we have so far completed and tested. We discuss our future plans for the oblivious algorithms library in Section 8.

4.5 An Oblivious k-NN Implementation

We now revisit the k-NN example introduced in Section 3. libOblivious can be used to craft a memory-trace oblivious k-NN implementation. This section describes the security-critical portions of the `classify_entry_oblivious()` implementation listed in Appendix A.

To ensure memory-trace obliviousness of the implementation, all flow-of-control patterns and memory access patterns which depend on input categories and attributes must be obfuscated. There are three places in our implementation of k-NN where this can happen.

First, the training set data points are sorted by their proximity to the given test set data point. The proximity was computed using the data point attributes. These attributes must not be leaked. The C++ STL's `std::sort()` algorithm is not oblivious, hence it may leak information about the values in the container being sorted. libOblivious provides the

Name	Description	Complexity
`osort`	Sort a container which support random access operations.	$O(n^2)$
`omax_element`	Find the greatest element in a container, and return an oblivious iterator to it. If several elements are equal to the greatest element, return an oblivious iterator to the first such element.	$O(n)$
`ofind_if`	Searches for an element in a container for which a given predicate P is valid. Return an oblivious iterator to the first such element found.	$O(n)$
`ofind`	Searches for an element in a container which is equal to a given element. Return an oblivious iterator to the first such element found in the container.	$O(n)$

Table 2: libOblivious algorithms

`oblivious::osort()` function to obliviously sort data. We use this instead of `std::sort()`:

```
35    oblivious::osort(neighbors, neighbors + training_set_size, cmp);
```

Second, tallying the votes for each class can also leak information. To perform the tally, we use a direct address table (implemented as a vector) indexed by class, e.g. 0, 1, 2, and 3 correspond to the first four classes. Whenever one of the k nearest neighbors "votes" for its class, that class's entry is incremented in the table. This increment operation—a read followed by a write—may leak the address in memory where the value is being updated. To perform this update obliviously, we use the `oblivious::O` template from libOblivious. `O` is a wrapper around an iterator which performs memory accesses obliviously:

```
37    oblivious::ovector<unsigned> class_votes(num_categories);
38    oblivious::O optr{class_votes.begin(), &class_votes};
39    for (int i = 0; i < k; ++i) {
40      int category = neighbors[i].entry->category;
41      optr[category] = optr[category] + 1;
42    }
```

A subscript `operator[]()` access on an `O` iterator will not leak the value of the subscript argument, nor will it leak the value of the iterator itself.

Third, just as `std::sort()` was not oblivious, `std::max_element()` is also not memory trace oblivious. libOblivious provides a solution:

```
43    entry->category =
44      oblivious::omax_element(class_votes.begin(), class_votes.end(),
45                              &class_votes) - class_votes.begin();
```

5 libOblivious Implementation

Section 4 presented an overview of the capabilities and APIs of libOblivious from the API user's perspective. This section discusses the implementation of libOblivious in much more

detail, especially the innovations which have not been presented in prior works.

5.1 The libOblivious Heap Allocators

The libOblivious primitives are adequate for facilitating memory-trace oblivious computation on their own. However, the primitive APIs are not elegant or easy to use. The `o_write()` API, for example, has 7 parameters. To make oblivious operations easier to use for the programmer, we added oblivious iterators.

Many aspects of the oblivious iterator `O` template were tricky to implement, but none more so than the following. Since an iterator is really just a pointer to a container element, how can our oblivious iterator determine the entire range of memory covered by the container? On any given oblivious read or write, the oblivious iterator will need to know this range so that it can touch each mask covered by the range. To make matters more difficult, a non-contiguous container such as a linked list may cover many non-contiguous memory regions. The oblivious iterator will somehow need to determine the base address and size of each region covered by the non-contiguous container. Our generalized solution for this problem exploits a seldom used feature of C++ [35]: user-defined heap allocators.

The C++ standard library has a category of containers called *allocator-aware containers* [25]. That is, containers which use a heap allocator to store values. Examples of allocator-aware containers include linked lists, maps, and vectors, but not statically-allocated or stack-allocated arrays. For any C++ STL allocator-aware container, C++ allows the default allocator to be substituted by a user-defined allocator, assuming the user-defined allocator conforms to the C++ standard's interface and behavioral requirements for a heap allocator [25].

Our general strategy in libOblivious is to define an allocator which maintains bookkeeping information for all of the memory currently owned by its parent container. When an oblivious iterator performs a read or a write, it queries the oblivious container's allocator to determine which memory region(s) it must touch. Yet the implementation is not so simple.

The requirements for user-defined allocators changed substantially from C++03 to C++11 [24, 48]. In particular, prior to C++11 all allocators were required to be stateless. C++11 relaxed this requirement. However, the need to maintain backwards compatibility meant that the new interface requirements for stateful user-defined allocators could not be defined cleanly [35]. For example, allocator-aware containers have always been required to provide a `get_allocator()` member function which returns a copy-constructed replica of that container's internal heap allocator [25]. Because copy operations are often expensive, C++ accessor methods typically return a reference to the internal member. But because user-defined allocators were previously required to be stateless, copy operations were free; thus the return-by-value semantics of `get_allocator()` was justifiable.

Unfortunately, the `get_allocator()` function is the only member function exposed by C++ STL containers which allows access to the container's internal allocator. The return-by-value semantics poses two problems for our design. First, the heap memory ownership bookkeeping is maintained in a linked list, which can potentially become quite large. Copying the entire list on each oblivious read or write would be extraordinarily inefficient. Second, it

may be possible for a container to allocate new memory after an oblivious iterator obtains a copy of the allocator, but before the oblivious iterator uses the copy to perform an oblivious read or write. In this scenario, the oblivious iterator would use stale information to determine which regions it should touch.

We could not find a perfect solution to this problem. Our compromise was to implement the heap allocator's internal state as a shared pointer [25, 48] (i.e. a reference-counted pointer) to the linked list containing the bookkeeping information. Thus when the oblivious iterator calls `get_allocator()` on the oblivious container to which it is bound, it receives a copy of the shared pointer, through which it can access the (fresh) bookkeeping information. We take care to ensure that the lifetime of the shared pointer copy does not extend beyond the current read or write operation, so as to not prolong the lifetime of the potentially large bookkeeping list after its parent container is destroyed.

The libOblivious heap allocator also strategically allocates memory to improve performance of oblivious reads and writes. Suppose that the underlying libc `malloc()` routine on a particular platform tends to scatter each successive heap allocation request across memory. In the worst case, consider a linked list where each node occupies 16 bytes, and each call to `malloc()` returns 16 bytes located in a mask not already occupied by any other node allocated in this list so far. If the linked list has n nodes, then an oblivious read will need to access all n nodes in order to touch every mask covered by the container. However, if the nodes have all been allocated contiguously, then only $\left\lceil \frac{16 \cdot n}{mask\ size} \right\rceil$ nodes will need to be touched. For an adversary who has observational power at cache block granularity, this optimization reduces the work load by as much as a factor of 4.

The libOblivious heap allocator has several implementations which are optimized for various data structures. The characteristics of each implementation are not particularly interesting. However, all implementations do share in common at least two attributes. First, they try to contiguously allocate the container elements as much as possible, with minimal internal and external fragmentation. Second, all memory regions covered by the allocators are guaranteed to be at least mask aligned. This enables yet another optimization, as described in Section 5.3.

5.2 The O Template

The `O` template wraps an iterator and a reference to the container into which the iterator points. As mentioned earlier, an oblivious iterator (an instance of the `O` template) behaves similarly to the iterator which it wraps, but with several important exceptions. The primary purpose of the oblivious iterator is to override the read and write semantics of C++, thus replacing conventional memory accesses on containers with memory-trace oblivious accesses. We demonstrate that although this is possible in C++, there are some limitations to our technique. This section describes the novel details of the `O` template implementation.

If `p` is a pointer into an array with element type `T`, then the default behavior in C++ for the indirection operation `*` is to return a reference to the pointed-to element, i.e. `T &`, and similarly for the subscript operator `[]`. Thus these operations behave predictably when used in expressions. The expression `T x = *p;` will assign the value of the reference returned by `*p`

to the new variable `x`. Similarly, `*p = x;` will assign the value of the variable `x` to the object referenced by `*p`, assuming that `p` is a non-`const` pointer: invoking the indirection operator on a `const` pointer will return a `const` (immutable) reference.

The indirection and subscript operations can be overloaded in C++ [25, 48]. This is how the indirection and subscript operations are defined for C++ STL iterators. The overload will typically behave similarly to the default operators: a dereference operation on an iterator will return a reference to the pointed-to element in the container. But the C++ standard does not mandate this as a requirement for all classes.

The temporary proxy idiom [5] exploits the flexibility of the indirection and subscript overloads to allow the developer to change the way in which container elements are accessed through an iterator. The basic mechanics of the idiom are as follows. Instead of returning a reference to the pointed-to element, the indirection/subscript operator can return a proxy object with one or both of the following operators overloaded:

```
operator T() { ... }
void operator=(const T &) { ... }
```

The first operator is known as the user-defined conversion operator [25, 48]. For a given class `U`, a user-defined conversion operator can specify either an implicit or explicit conversion to any other type `T`. Without the `explicit` keyword in the declaration, the conversion is assumed to be implicit. The second operator defines assignment. For example, when an object of the given class is assigned (via `=`) a value of type `T`, this operator will be called to perform the assignment.

The temporary proxy idiom uses the user-defined conversion operator to "read" from the pointed-to element, and it likewise uses the assignment operator to "write" to the pointed-to element. The process works as follows. Suppose that `i` is an iterator which points to elements of type `T`, and `i`'s class uses the temporary proxy idiom. In the statement `T x = *i;`, The indirection `*i` will construct and return a proxy object which, at minimum, knows the address of the pointed-to element. The proxy object and `x` do not share the same type, so the compiler will search for a valid conversion sequence from the type of the proxy object to `T`. The proxy object's user-defined conversion `operator T()` satisfies this requirement, and thus is selected to perform the conversion. The operator can simply return a copy of the pointed-to element, or it can do something more interesting. The process for a write is similar. The statement `*i = x;` will invoke the proxy object's assignment operator for a parameter of type `T`, thus allowing the user to customize the write behavior of the iterator. Note that the lifetime of the proxy object should not extend beyond the evaluation of the statement, which is why the idiom is called temporary proxy.

The oblivious read/write mechanism of the `O` template is implemented using the temporary proxy idiom. If `o` is an oblivious iterator, then the statement `T x = *o;` is evaluated as follows:

1. The evaluation of `*o` returns the proxy *oblivious accessor* object, which essentially wraps a reference to `o`.

2. Overload resolution for a conversion sequence from the type of the oblivious accessor to `T` selects the oblivious accessor's implicit conversion operator, which performs the following sequence of operations:

(a) Call `get_allocator()` on `o`'s container reference to retrieve a shared pointer to the container's oblivious allocator.

(b) Call `get_regions()` on the oblivious allocator, which returns a list of ⟨base address, size⟩ pairs of all heap memory regions covered by the container.

(c) Call `o_read_list_T()` to obliviously read the value of type `T` pointed to by `o` from the given list of regions.

3. The value returned by this last call is assigned to `x`.

The process is similar for an oblivious write through an oblivious iterator, except that the assignment operator on the oblivious accessor is called, and we use `o_write_list_T()` in step 2(c) above.

Unfortunately, the temporary proxy object does not always behave as expected. If we instead declare a variable with the C++ `auto` keyword, as in `auto x = *o;`, then the C++ type inference rules will infer the type of `x` to be the type of the value being assigned to `x`—in this case, the proxy object itself. Hence instead of triggering a call to the proxy object's conversion operator, the compiler will use the proxy object's copy constructor to instantiate `x` as a copy of the proxy object. In certain situations, this may actually be what the programmer would want. For example, the copy of the proxy object can act like a reference to the pointed-to element in the container, allowing repeated oblivious accesses to it. But it is perhaps more likely that the copy would be created accidentally.

One way to prevent these accidental copies would be to delete the proxy object's copy constructor. However, as of C++17 this fix is no longer workable due to guaranteed copy elision [18, 25].

A similar consequence of the temporary proxy idiom arises when calling template functions. Given a template function `foo()` with prototype

```
template <typename T> void foo(const T &arg);
```

the function call `foo(*o);` (where `o` is an oblivious iterator) will instantiate the template parameter `T` as the type of the proxy object. In this case, the lifetime of the temporary proxy object will be extended, and `arg` will be a `const` reference to the proxy object. Again, this is most likely not the behavior that the user would intend.

It would be inconvenient for the user if he/she were always required to specify the element type when reading from an oblivious iterator. Therefore, the iterators and the oblivious accessor both expose a member type `value_type`, which is the type of the elements in the container. For instance,

```
typename decltype(o)::value_type x = *o;
typename decltype(*o)::value_type x = *o;
```

The first statement uses the oblivious iterator's `value_type`. The second statement uses the oblivious accessor's `value_type`. Both statements are equivalent.

One other substantial limitation of the template proxy idiom is that it does not provide any strategy for similarly modifying the behavior of the member access (`->`) operator. The member access operator is typically used to implement wrappers around pointers, such as iterators and smart pointers. However it is less customizable than the dereference and

subscript operators. According to section 16.5.6 of the C++ standard, "An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x` of type `T` if `T::operator->()` exists and if the operator is selected as the best match function by the overload resolution mechanism" [25]. Hence any overload of the `->` operator must either return a raw pointer or another object with an overloaded `->` operator. There is simply no way to return a proxy object without inviting an infinite recursion.

The current workaround in libOblivious is to first obliviously read an object from a container (i.e. create a copy of it on the stack) and then use the `x.m` member access to read a member.

5.3 Optimizations for libOblivious Primitives

We made several observations about the execution of libOblivious primitives which led us to introduce a series of performance optimizations. To the best of our knowledge, these optimizations for data-oblivious or memory-trace oblivious programs have not been reported in related works. Section 6 demonstrates the observed performance benefits of some of these optimizations.

Our first observation pertains to the `o_copy()` and `o_swap()` primitives, both of which move chunks of contiguous data in memory.

Observation 1. Loops which operate sequentially on pairs or tuples of data can often be optimized into parallel vectorized operations using single instruction, multiple data (SIMD) instructions.

The use of multimedia SIMD instructions in commercially available desktop, server, and laptop CPUs to achieve superword level parallelism (SLP) optimizations was first postulated by Larson and Amarisinghe [28, 29]. This strategy has since been adopted by numerous optimizing compilers such as GCC and Clang.

Copy and swap operations sequentially operate on tuples of data in a manner which can be vectorized. However, since we had to write these operations manually in assembly (to prevent the compiler from using optimizations that would violate obliviousness) we also effectively disabled all SLP optimizations. Our solution is to manually vectorize the oblivious copy and swap operations when the number of bytes to copy or swap is sufficiently large enough to fill an Intel AVX2 vector [15], specifically, 256 bytes. The code which vectorizes memory-trace oblivious copy in libOblivious for 256 contiguous bytes is given in Listing 5.

Instead of inline assembly, we use Intel's convenient Streaming SIMD Extensions (SSE) compiler intrinsics [11], which are available on popular compilers such as GCC, Clang, and MSVC for supported platforms. For example, the `_mm256_loadu_si256()` intrinsic compiles to an appropriately sized `vmov` instruction. The `o_copy_i256()` function unconditionally loads 256 bytes from both the `left` and `right` pointer operands. It then uses a vector blend operation—a kind of piece-wise ternary operation for vectors—to store either the `ltmp` or `rtmp` vector into the `result` vector, which is then written to the destination `dst` in memory.

The generic `o_copy()` and `o_swap` operations dynamically determine the vectorization depending on the number of bytes requested for copy or swap at runtime. The C++-only

```
1  void o_copy_i256(__m256i *dst, int cond,
2                   const __m256i *left, const __m256i *right,
3                   size_t offset) {
4    const __m256i mask = _mm256_set1_epi32(!!cond - 1);
5    const __m256i ltmp = _mm256_loadu_si256(left + offset);
6    const __m256i rtmp = _mm256_loadu_si256(right + offset);
7    const __m256i result =
8        _mm256_blendv_epi8(ltmp, rtmp, mask);
9    _mm256_storeu_si256(dst + offset, result);
10 }
```

Listing 5: `o_copy_i256()` implementation uses `vpblendvb`

`o_copy_T()` and `o_swap_T()` operations are able to infer the number of bytes to copy/swap at compile time, since the number of bytes is simply the size of the object(s) to be copied or swapped. Thus the vectorization can be inlined, and possibly loop-unrolled, hence achieving even better performance without losing memory-trace obliviousness.

This optimization allows our memory-trace oblivious copy and swap operations to perform comparatively well against the analogous primitives in C++, as demonstrated later in Section 6. In fact, as of this writing, libstdc++ and libc++ both do not vectorize the `std::swap()` operation. So our oblivious `oswap()` operation is actually faster than `std::swap()` for larger objects.

Observation 2. When obliviously reading or writing an amount of data greater than or equal to the size of the mask, the read/write can be performed faster by an oblivious copy.

Section 4.1 described our basic approach for obliviously reading data from a memory region. When reading 4 bytes, we use the `vpgatherdd` instruction, and we use `vpgatherqq` to read 8 bytes, which increases throughput. To read more (contiguous) bytes, we simply repeat this process, reading 4 or 8 bytes at a time. Figure 3 illustrates this kind of memory striping pattern to read data. Reads shown in red use `vpgatherdd`, and reads in blue use `vpgatherqq`.

The striped reads offer reasonable throughput because they allow us to read data obliviously, without having to read from every single address within a memory region. But once the size of the read reaches the size of the mask (for our adversary model, a cache block), it becomes necessary to read from every address—otherwise the accesses on cache blocks would not be uniformly distributed. In this case, instead of striping we can adopt a more efficient strategy.

In the last diagram in Figure 3, each green box indicates an oblivious copy. Thus, when an oblivious read is requested with n bytes where $n \geq \textit{mask size}$, the `o_read()` primitive iterates over the memory in n-sized chunks, invoking `o_copy()` for n bytes on each chunk. The condition for `o_copy()` is false for every chunk except for the chunk whose beginning address matches the given `addr` parameter. Hence only the desired value is actually written

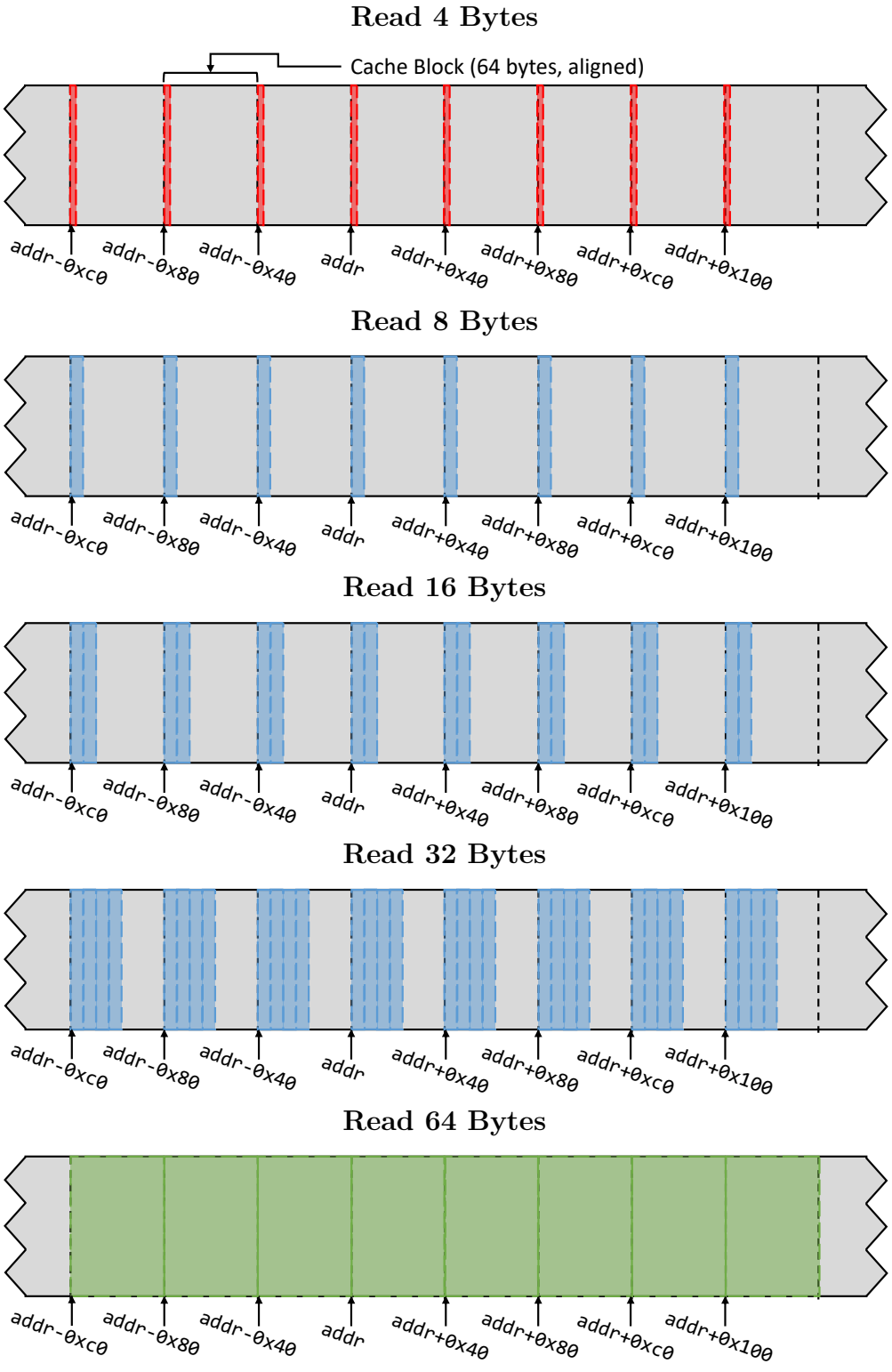

Figure 3: Obliviously reading n bytes from an array

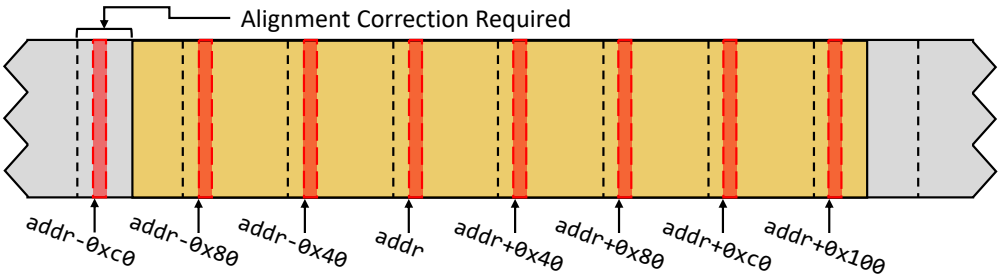

Figure 4: An correction is required when the memory region is not aligned to the mask.

to the destination. This optimization causes the throughput of `o_read()` to scale linearly with the size of the memory region.

Observation 3. If a memory region being read/written from/to fits entirely within a mask, then an oblivious read/write is not necessary.

This observation may sound trivial, but it can be used to improve performance in certain scenarios. For instance, the oblivious k-NN implementation in Section 4.5 used an `O` iterator to perform oblivious accesses on a direct address table containing the tally for each class. If the number of classes is small enough—specifically, less than 16—then the entire data structure may fit within the mask. Indeed the oblivious containers will guarantee that this is the case because all of the memory they allocate for data is at least mask aligned.

The oblivious read and write primitives inline a check to determine whether the target memory region is entirely contained within a mask. If so, the given address is simply read/written from/to through the pointer. Otherwise, a call is made into the libOblivious DLL to perform an oblivious read or write. For a single operation, this optimization may make very little difference. But in the k-NN example, where potentially many random accesses are performed in a loop, eliding repeated calls into a DLL can achieve a noticeable performance improvement.

The last optimization we present also pertains to alignment. Our discussion thus far has not considered the case where either the beginning or the beginning or the end of a memory region is not aligned to a mask. Consider the example in Figure 4. The memory region from which to obliviously read (shaded in orange) is not aligned to the mask at either boundary. To ensure memory-trace obliviousness for all accesses, we must always touch these masks.

The easiest solution is to treat the memory region as though it extends all the way to the beginning and end of all of the masks it covers. Computationally, this is as simple as computing the alignment correction at the beginning of the region (depicted in Figure 4), subtracting the correction from the base of the region, and adding the correction to the end of the region. Similarly, we compute the correction at the end of the region, and add this correction to the region's size.

Observation 4. If we know at compile time that a memory region will be aligned to the mask, then we can skip the alignment check and adjustment at run time.

All of the oblivious read and write primitives have a `bool` parameter called `is_aligned`, which asserts that the given memory region will be aligned to the mask. The alignment check and adjustment described above is an inlined check. So of an API call asserts `true` for `is_aligned`, the check will be entirely optimized away by the compiler. This is precisely what the `O` template does when it is defined over an oblivious container, because libOblivious containers always allocate memory with mask alignment.

5.4 Shallow vs. Deep Copying in libOblivious

Just about every object-oriented or imperative programming language supports some kind of mechanism to copy an object from one location in memory to another. For instance, consider this structure definition of a string in C:

```
struct MyString {
  size_t size;
  char *data;
};
```

If `s1` is a `MyString`, then the statement `MyString s2 = s1;` copies the value of `s1` to the memory occupied by `s2`. Note that the string pointed to by `s1.data` is not actually copied by this operation. Only the `s1.size` member and the `s1.data` pointer itself will be copied, 16 bytes in total on a 64-bit platform. Hence, after the copy `s1` and `s2` will share the same state: any modification to `s1`'s string will be visible through `s2`, and vice-versa.

In C++, this kind of copy is commonly referred to as a *shallow copy* [48]. Unlike C, C++ provides support for making *deep copies*, copies which completely copy the state of the source object. Deep copies are supported via overload of the copy constructor. For instance, a deep copy constructor for `MyString` might look like

```
MyString(const MyString &other)
    : size(other.size), data(new char[size]) {
  memcpy(data, other.data, size);
}
```

Without going into too much detail about the syntax of C++ constructors, this example creates a copy of a `MyString` by first copying its `size` member, and then instantiating its own `data` member with a pointer to `size` freshly allocated `char`s on the heap. Then it copies the string state to the fresh heap memory.

Currently, libOblivious primitives are only able to make shallow copies of objects. The reason should be obvious. A deep copy requires knowledge of the meaning of an object, and what exactly constitutes its deeper state. Only the author of the `MyString` structure would know that the `size` member should refer to the number of characters pointed to by the data member; an algorithm cannot simply infer this. Hence all of the libOblivious C++ primitives statically check that the types of the arguments are all trivially copyable, meaning that they use the default copy constructor. That is, only shallow copies of the object can be made using copy semantics [25].

We do know that it is possible to support deep copies with libOblivious. But, this would require a lot of extra work on account of the developer. One solution could be to support a

static interface for obliviously (deep) copyable objects, e.g.

```
template <typename T> struct ODeepCopy {
  static T o_deep_copy(bool c, const T &left, const T &right);
};
```

If the developer wants one of her classes to be obliviously deep copyable, she can specialize the `ODeepCopy` struct for that class and defining an appropriate `o_deep_copy()` operator, recursively invoking `o_copy()`s or `o_deep_copy()`s as needed. The libOblivious C++ primitives could first check whether the class is trivially copyable and, if not, check whether the class specializes `ODeepCopy`. If so, the primitives will use the user-defined `o_deep_copy()` instead of `o_copy()`.

6 Evaluation

This section describes our experiments to test the performance of libOblivious' primitives and algorithms. Our results demonstrate that the libOblivious read and write primitives offer a substantial performance increase over the naïve solution described in Section 2. Other oblivious primitives compare favorably to their C++ and C++ STL counterparts, and in one case an oblivious primitive outperforms its STL counterpart. Our algorithms generally perform well in comparison to the C++ STL, with some exceptions. We discuss the reasons for this poor performance, and suggest how it may be addressed.

6.1 Test Setup

We performed our tests on a laptop machine with an off-the-shelf dual-core Intel SkyLake CPU, with a base frequency of 2.9 GHz. The CPU has a 32 KB L1 data cache, a 256 KB L2 cache (per core), and a 4 MB L3 cache (shared). It has 16 GB of 2133 MHz LPDDR3 RAM. All tests were performed on the host operating system, macOS 10.13. Individual tests were compiled by GCC version 8.1.0. We use libstdc++ as the C++ standard library.

Before running each test, we pre-allocate and lock (e.g. using `mlock()`) each region of memory required by the test to preclude paging. Each test is run 100,000 times for each value of each of the test parameter(s) (e.g. the size of the data structure being operated on). The reported result is an average of the 100,000 iterations. One exception to this rule is the sorting test, which we run for only 1,000 iterations. The result of the first iteration for each test is always discarded because the instruction and data caches are not yet hot, thus the first iteration is always slower.

For tests which require input parameter(s), such as an index from which to read or write, or a value to find in a container, we randomize the parameter for each iteration.

6.2 libOblivious Primitive Results

libOblivious currently provides four categories of primitives, upon which more complicated algorithms can be built. The four categories are reads, writes, swaps, and ternary-like copies.

For each primitive, we analyze its performance with respect to the size of the data and/or the memory region over which the primitive is being applied.

Figure 5 shows our performance results for the libOblivious primitives. The top row of plots shows the performance of the `vpgatherdd`-based oblivious read and write primitives, `o_read_i32()` and `o_write_i32()`, respectively. Both scalar and vectorized versions of `o_read_i32()` substantially outperform the naïve solution. In general, the scalar solution offers a roughly 10-14x performance improvement. The vectorized solution performs best when applied to data already in the CPU's L1 and L2 caches. Beyond the L2 cache, performance gains against the scalar solution gradually taper off. In particular, we observed at best a 32x improvement over the naïve solution when operating on data within the L1 cache, a 21x improvement on data in the L2 cache, and an 11x improvement within the L3 cache. This last result is slightly faster than our observation for the scalar read.

Theoretically, the best performance improvement we could have hoped to achieve over the naïve solution with our vector or scalar solutions would have been 16x. This is because the vector and scalar solutions read just 4 bytes for every 64 bytes reads by the naïve solution. Yet beyond the L2 cache, the boost bottoms out at 10-11x. We can most likely attribute this discrepancy to cache misses. When the array is larger than 256 KB, the first memory access in a cache block is always more expensive than each subsequent access, because the first access is always an L2 cache miss. So for the naïve solution, each access in a cache block after the first is cheaper. For the scalar and vector solutions, every access should be an (expensive) L2 cache miss. Hence the 16x improvement beyond the L2 cache is almost certainly unattainable. We consider our 10-11x improvement to be very close to optimal.

The results for `o_write_i32()` are less favorable. As we mentioned earlier in Section 5, at the time of this writing we did not have access to a newer Intel-based machine with AVX512 instructions, which include the `vpscatter` family of instructions. We believe that these instructions could be used to drastically improve the performance of the oblivious write primitives, in the same manner as `vpgatherdd` does for oblivious reads. At best, we achieved a 12x performance improvement in the L1 cache, 11x in the L2 cache, and 7x in the L3 cache. We plan to revisit this issue in the near future.

The second row of plots in Figure 5 shows the performance of the `o_read()`/`o_write()` primitives against the number of bytes being read/written from an array of fixed size, 1 MB. Each plot also shows the throughput (number of MB read/written per second) of the operation. Throughput is at its worst when reading or writing only 4 bytes. In this case, `o_read_i32()` and `o_write_i32()` are being called by `o_read()` and `o_write()`, respectively. When the test parameter is increased to 8 bytes, `o_read()` instead calls `o_read_i64()`, and likewise for `o_write()`. These 64-bit primitives, as discussed in Section 5, instead use the `vpgatherqq` instruction, which gathers four 64-bit integers instead of eight 32-bit integers. The execution time is nearly identical, and thus the throughput is doubled.

Throughput remains steady until the parameter size reaches 64 bytes: the size of a cache block. At this point, `o_read()` and `o_write()` instead use `o_copy()`. The reasons for this were discussed in Section 5. Consequently, for reads and writes exceeding 64 bytes the throughput increases linearly with respect to the number of bytes being read or written.

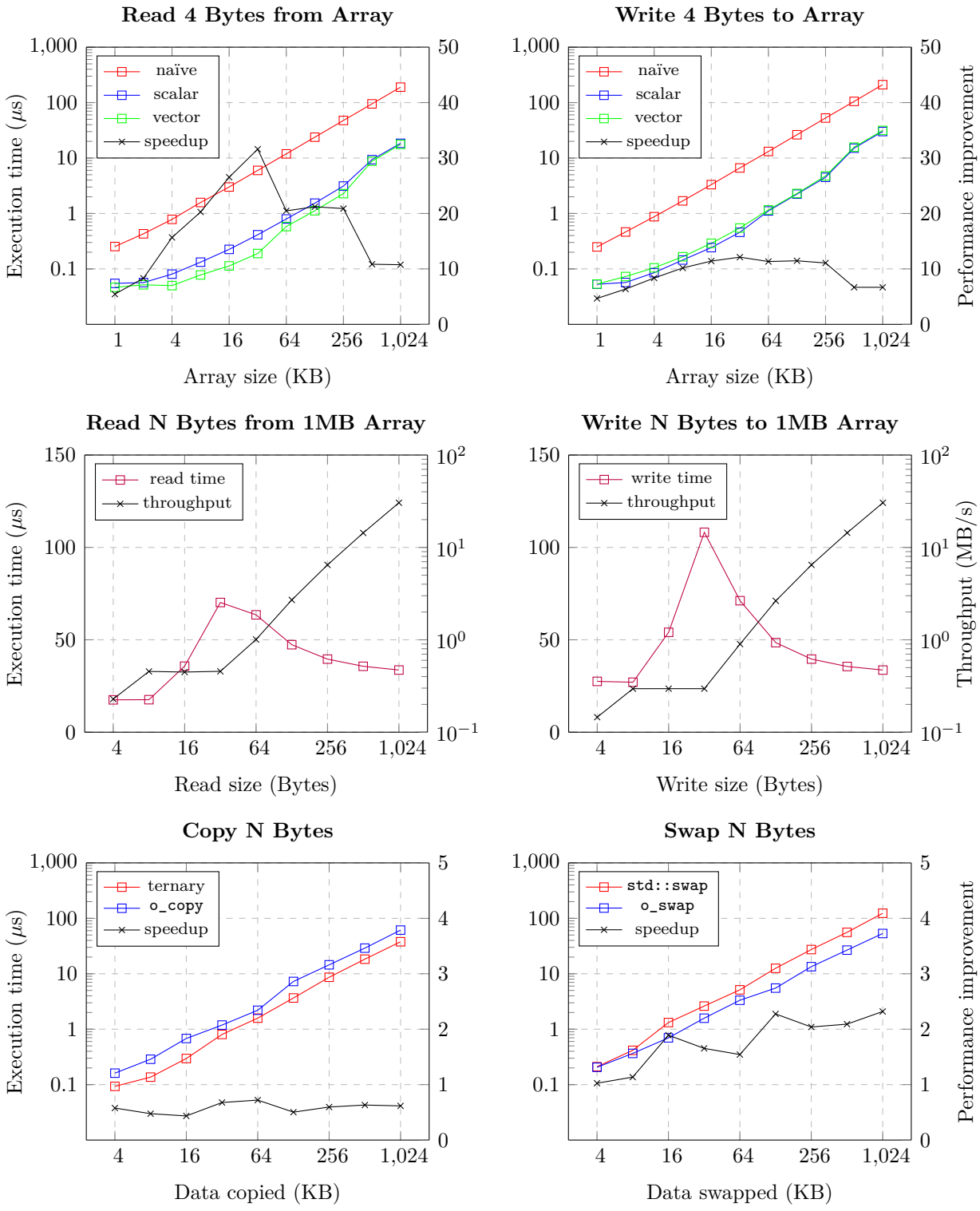

Figure 5: Performance results for libOblivious primitives

The bottom row in Figure 5 shows our results for the `o_copy()` and `o_swap()` primitives, respectively. The first plot compares the performance of `o_copy()` against the C++ ternary `?:` operator. As the number of bytes copied increases, the slowdown of `o_swap()` converges toward 1.5x. This result should not be surprising. When the statement

```
*d = c ? *x : *y;
```

is compiled (and the pointee type is sufficiently large), the compiler will produce a branch depending on `c`, which will invoke a `memcpy()` either from `*x` to `*d` or from `*y` to `*d`. If the libc `memcpy()` is optimized for AVX2, then it will repeatedly invoke vectorized `vmov` instructions to load from either `*x` or `*y`, and then store in `*d`. The analogous libOblivious operation would be

```
*d = o_copy_T(c, *x, *y);
```

which would similarly invoke `vmov` to load from both `*x` and `*y`, but then use `vpblendvb` to select the correct vector depending on `c`, and then write that result to `*d` with a third `vmov`. So the oblivious copy is performing exactly three vectorized memory accesses for each two vectorized memory accesses performed by `memcpy()`, which explains the 1.5x slowdown.

The performance results for `o_swap()` are much more impressive. Our oblivious swap implementation actually outperforms the C++ STL's `std::swap()` implementation by up to 2.3x. This is because `std::swap()` is not optimized with AVX2 to vectorize swaps on sufficiently large parameters, at least in libstdc++ 8.1.0. At a glance, this might seem like a surprising oversight. However, it is conventional in C and C++ to swap values in this manner only when the types of the values are sufficiently small, e.g. when they can fit into a general purpose register. When the values are larger, the convention is to instead swap pointers to these values, which is obviously more efficient.

6.3 libOblivious Algorithm Results

For each libOblivious algorithm, we test it over one or more oblivious containers and vary the containers's sizes. The results are shown in Figure 6.

The first row of plots test our `ofind()` implementation against `std::find()` over a vector and over a linked list. The container is initialized with a sequence of integers $\{0, \ldots, n-1\}$, where n is the number of elements which can fit into the container of its given size. Each test searches for a random element in the range $[0, n)$. Hence, on average we would expect the oblivious find to require twice as many memory accesses as the standard library find (because on average, the randomly chosen element will be located at the middle of the range). Our results demonstrate a roughly 10-12x slowdown over the oblivious vector container, and 2.7x over the oblivious forward list. We discuss the reasons for this discrepancy in the next section.

Our oblivious sort is the worst performing algorithm, when compared to its STL counterpart. Over an array of size 64 KB, `osort()` runs nearly three orders of magnitude slower, on average. And this discrepancy only increases as the size of the container increases, because

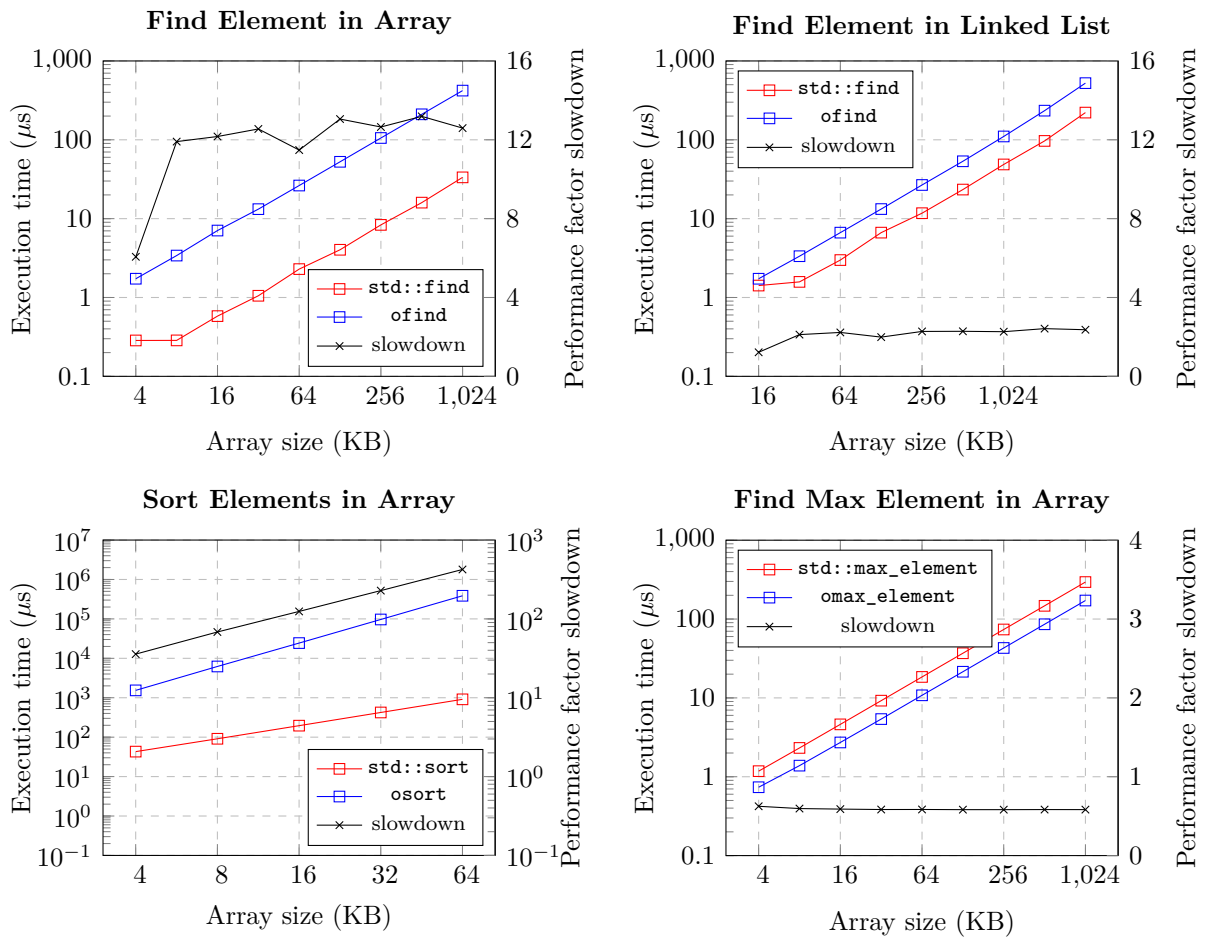

Figure 6: Performance results for libOblivious algorithms

the asymptotic complexity of the oblivious sort is $O(n^2)$, as opposed to $O(n\,log(n))$ for `std::sort()`.

Of the three algorithms discussed in this section, only `omax_element()` actually outperforms its STL counterpart, improving performance by nearly a factor of two. However, we believe this may be related to a known optimization bug in GCC[2]. When both `omax_element()` and `std::max_element()` are compiled by Clang, we observe a 3.4x slowdown over a 1MB array. Because the entire container must always be searched to find the maximum element, an optimal result would have been no slowdown, or very little slowdown.

6.4 Discussion

With the exception of oblivious writes (due to lack of vector scatter support on our test platform), our oblivious primitives perform exceptionally well. However, our oblivious algorithms—based on those same primitives—do not always perform as well.

As we discussed earlier in Section 5, the oblivious copy and swap primitives needed to be implemented in assembly so as to avoid compiler optimizations that would trigger a branch based on the (potentially) secret copy/swap condition. An unfortunate side effect of this approach is that the presence of inline assembly in any compiler basic block disables many optimizations for the rest of that basic block [6]. If the basic block is encountered infrequently, then the slowdown may be unnoticeable. But if the basic block is part of a loop, then the slowdown will be multiplicative. And in the case of our oblivious sort, the inline assembly is actually nested within two loops.

This is precisely what we have observed by manually inspecting the compiled binaries of the oblivious algorithms. For example, compared to `std::find()`, we find the compiler output for `ofind` over a vector to be more complex, and unnecessarily so. The performance discrepancy for `ofind` over a linked list is smaller, in part because each iteration of a traversal over a linked list is partially dominated by reading the address of the next link from memory. There is still overhead caused by the disabled optimizations, but it is less noticeable.

One solution would be to manually write the loop in assembly, for each loop which contains an oblivious copy or swap primitive. Unfortunately it is not possible to do this in a manner which is entirely generic for each algorithm. For instance, `std::find()` is parameterized by the type of its iterator arguments. The compiler uses this type to determine whic increment operator, dereference operator, and value equality comparison operator to use. Each of these is called within the main loop. But these operators are almost always inlined by the compiler. Thus, for any two invocations of `std::find()` which differ only in container type, iterator type, or value type, the compiler may produce different output.

It is not possible to manually write enough inlined assembly procedures to account for all combinations of type parameters, especially for every oblivious algorithm. It may be possible to write assembly procedures which call the dependent operators, instead of inlining them. But this would likely produce code that also performs poorly.

[2]bug reported here: `https://stackoverflow.com/questions/25622109/why-is-c-stdmax-element-so-slow`

7 Related Work

The closest relative to libOblivious is the work done by Ohrimenko et. al. at Microsoft Research, who used memory-oblivious primitives to implement several popular machine learning algorithms [37]. At a low level, their approach is very similar to ours. They use x86 instructions such as `cmov` and `vpgatherdd` to build oblivious primitives, upon which they constructed several popular machine learning algorithms. Their implementation was tested on Intel SGX.

Our work on libOblivious differs primarily in its purpose. Whereas the goal of the work done at Microsoft Research was to build machine learning algorithms for multi-party computation on trusted hardware, our goal is to build a software library which can facilitate memory-trace oblivious computation over a wide variety of problem domains, and on multiple platforms. However libOblivious was influenced by other work as well.

7.1 ORAM

Oblivious RAM [22, 23, 47] (ORAM) is a related technique for obfuscating memory access patterns, and can be used to defeat side-channel attacks. This technique is appealing because it operates under the hood, and thus places no additional burden on the programmer. But it does incur a rather steep $O(log\ n)$ cost for each memory access, where n is the size of the entire program memory. Thus protecting all program secrets in a single ORAM has been has been shown to be inefficient [19, 46, 51].

ZeroTrace is an Intel SGX enclave runtime which uses an ORAM-based memory controller to enforce data obliviousness [43]. The ZeroTrace usage model assumes that a remote client needs to make remote queries on a large (e.g. $>$ 10 GB) dataset. These queries could include reads, writes, key-value lookups, etc. Within an SGX enclave, ZeroTrace uses a Path ORAM [47] library to look up the ORAM leaf containing the query. Within each ORAM leaf, the lookup can be performed by an untrusted party, so the final request is forwarded to a non-enclave fetch/store controller, which can execute with less overhead. This design has demonstrated favorable performance for queries over large datasets.

Several works have specifically targeted the design of oblivious data structures using ORAM variations. Wang et. al. proposed a framework for created ORAM-based oblivious data structures [49]. Their work reduces access time overhead on oblivious data structures from $O(n)$ to $O(logn)$. However, the data structure storage scheme requires $O(nlog(n))$ storage. By specifically optimizing their implementation for each data structure, they were able to achieve a 10-15x speedup over naïve ORAM for moderately sized ORAMs. Keller and Scholl also implemented optimized data structures over ORAMs [26]. Their work compared specifically considered the problem of secure multi-party computation (SMPC) using oblivious data structures. Their analysis also compared the performance of data structures using Path ORAM vs. Tree ORAM, and also against naïve scalar accesses.

ORAM itself also does not protect against timing side-channel attacks, such as an attacker inferring a loop termination condition for the number of loop iterations. This is a problem which is addressed by libOblivious' algorithms library. For relatively small (e.g. $<$ 100 MB)

data structures, ORAM has been demonstrated to be significantly slower than the kind of memory scanning techniques employed by libOblivious [41].

7.2 Language-based Oblivious Computation

Other solutions have aimed to implement data-oblivious or memory-trace oblivious protection at the source language level. Liu, Hicks, and Shi have provided foundational work in programming language theory for constructing a programming language with the property of memory-trace obliviousness [30]. Furthermore, their work exceeds memory-trace obliviousness to also cover termination-sensitive noninterference, i.e. secret program inputs cannot influence public program outputs. Memory accesses are obfuscated using a maximal partition of ORAM banks over all secret program data structures, and also conditionally-executed code, where the condition is secret.

ObliVM [32] is a domain-specific language for memory-trace oblivious computation, built on the theoretical foundation established by Liu, Hicks, and Shi described above. The ObliVM-lang is a C++-like language with features such as structures, generics, loops, etc. Each variable in ObliVM-lang is annotated as either “public" or “secret." Memory-trace obliviousness with respect to the secret values is enforced by the type system. Programs written in ObliVM-lang are compiled into ObliVM-GC, a Java-based garbled circuit implementation. This effectively allows a computation to run on an untrusted platform in encrypted form.

Obliv-C [53] is an extension (i.e. strict superset) to the C language which adds a category of data called “obliv," similar to the “secret" annotation in ObliVM. Similar to ObliVM, typing in Obliv-C enforces memory-trace obliviousness for obliv-annotated values. The implementation is simply a wrapper around GCC which performs the oblivious type checking against a rule set, and then translates the Obliv-C code into ordinary C code, which is then passed in to GCC.

7.3 Program Transformation

Program transformation techniques can be used before or during the compilation process to analyze and modify control flow patterns and memory accesses, for the purpose of achieving some degree of obliviousness.

Raccoon [41] is a kind of source code preprocessing tool which transforms C or C++ programs into a form that is memory-trace oblivious. Raccoon only requires the programmer to annotate secret variable in the source code. The tool performs inter-procedural taint analysis on the secret variables to determine precisely where a secret may influence either control flow or a memory access. When secret data is found to influence a branch, Raccoon obfuscates the control flow by forcing execution of both paths, replacing memory writes in each path with an oblivious store operation, similar to the oblivious ternary `o_copy()` described in this paper. Raccoon uses software Path ORAM to obfuscate memory accesses which depend on program secrets.

Dr. SGX [12] is an IR-level instrumentation tool and library which obfuscates heap memory accesses to make an SGX enclave program data oblivious at cache block granularity.

Unlike Raccoon, Dr. SGX does not require any additional code annotation by the user. It obfuscates heap memory accesses by first producing a randomized heap memory layout, and then constantly re-randomizing the layout, subject to a configurable time window. Dr. SGX uses an LLVM IR pass to instrument the input program, replacing ordinary heap memory accesses with calls to the Dr. SGX library. Each call consults the current iteration of the pseudo-random memory permutation function to obtain the correct randomized address. Although this solution requires less work from the user, it also cannot distinguish between public and private data. Thus every access is obfuscated, which may introduce unnecessary overhead.

SGX Lapd [21] specifically targets the adversary using forced page eviction to observe enclave memory traces at page granularity, as in [52]. However, instead of promising full prevention of memory-based side-channel attacks, SGX Lapd is simply a mitigation technique which makes it much more difficult for the adversary to infer enclave secrets from a memory trace. Specifically, SGX Lapd consists of two components: (1) an untrusted kernel module which provides 2 MB pages to the enclave, and (2) a program transformation tool which inserts instrumentation code into the enclave program. Whenever enclave program control flow or data accesses cross a 4 KB-aligned boundary, the instrumentation code dynamically checks whether a forced 4 KB page fault has occurred. If so, this would likely indicate a malicious OS feeding 4 KB pages to the enclave and forcibly evicting them. Like Dr. SGX, SGX Lapd uses an LLVM IR pass to perform the enclave code transformation. This technique effectively increases the granularity of the adversary's visibility by a factor of 2^9.

7.4 Other Approaches

The GhostRider [31] project features a co-designed assembly language, compiler, and hardware architecture for memory-trace oblivious computation. Program analysis of the assembly allows GhostRider to partition program data into one of three hardware RAMs depending on its security classification and usage: (unencrypted) RAM, (encrypted) ERAM, or ORAM. The usage scenario for GhostRider involves an untrusted host system communicating with a GhostRider co-processor, both of which share the same RAM/ERAM/ORAM banks. The authors simulated their design in software, and on an FPGA system.

8 Future Work

At the time of this writing, libOblivious is still a work in progress. Each of the four components in libOblivious has aspects which can be expanded and improved upon.

We plan to complement the oblivious read and write operations with new oblivious update operations. At present, an oblivious update to a value in a container or memory region is unnecessarily expensive. Suppose that the programmer wants to obliviously increment a value in an array. With the current API, the programmer would have to call an `o_read()` to obtain the value, then increment it appropriately, and finally use `o_write()` to record the

updated value to memory. Since each `o_write()` function performs both a read and a write, the sequence

`v = o_read(...);` $\longrightarrow$ update `v;` $\longrightarrow$ `o_write(v);`

actually performs two oblivious reads, followed by one oblivious write. A semantically equivalent memory-trace oblivious operation could be performed by a single primitive, say `o_update()`, which would perform a single read, then increment/decrement the value by a given amount, and finally write the updated value back to memory.

This limitation among the libOblivious primitives also limits other components of libOblivious. For this same reason, the `O` template does not support arithmetic assignment operations on dereferenced values, such as `+=`, `-=`, `*=` etc. The addition of an oblivious update operation would allow oblivious iterators to support efficient value assignment operators on dereferenced values.

As shown in Table 2, we still have many algorithms remaining to implement. We have so far implemented 4 oblivious algorithms. We believe that there are roughly 40-50 algorithms in the C++ algorithms library [25] that are not memory-trace oblivious, but which can be made oblivious.

As discussed in Section 6, several of our oblivious algorithms do not perform as well as they should, when compared against their C++ STL counterparts. Depending on the algorithm and the data structure being operated on, we have observed as much as a 10-12x slowdown for linear algorithms (the result for `ofind()` over a vector of 32-bit integers), whereas the theoretical minimum slowdown would have been roughly 2x. At this time, we are unaware of any feasible strategy for improving the performance of these algorithms, other than manually writing optimized loops in assembly.

We have also recorded an enormous slowdown for our oblivious sort algorithm. Our current implementation uses a simple variation on bubble sort, but with oblivious swaps. A better solution would be to use something like Batcher's sorting network [10], which has complexity $O(n\,log^2\,n)$. This approach has been employed elsewhere for memory-trace oblivious computation [37].

We mentioned in Section 4.2 that the oblivious containers are simply type aliases for STL containers, but with the oblivious heap allocator substituted for the default C++ STL heap allocator. As such, the member operations on libOblivious containers are identical to the member operations on the STL containers. Because some STL container operations are not memory trace oblivious (e.g. `sort()`, `merge()`, `find()`, etc.), this means that the same operations on the libOblivious containers will be unsafe. One solution is to instead have each oblivious container inherit the public interface of its corresponding STL container. For each inherited operation that is not oblivious, we override it with a semantically equivalent oblivious operation. If any operation cannot be made memory-trace oblivious, we disable it. All other operations can be inherited as-is. This design would allow libOblivious containers to be used in exactly the same way as STL containers, except for the few operations which cannot be supported safely.

Recent advances in hardware and other low-level protections against cache-based side-channel attacks have made these attacks more difficult [14, 33]. In the near future, cached-

based side-channel attacks may no longer be feasible. However, side-channel attacks at page granularity will still be an issue. We are currently planning to update libOblivious with optimizations to facilitate memory-trace oblivious computation at page granularity and with substantially improved performance.

Finally, as discussed in [41] and [37], the `vpgather`-based memory scanning technique currently employed in libOblivious is a competitive alternative to ORAM-based implementations. In general, the scanning technique can be faster for smaller structures, while ORAM is faster for larger structures (e.g. $>$ 100 MB). This is because ORAM execution time scales logarithmically, with a large constant factor, whereas scanning scales linearly with a small constant factor. But there are other tradeoffs under certain application scenarios. For example, Ohrimenko et. al. found the scanning technique to be generally preferable for machine learning algorithms [37]. We may at some point wish to add back-end support to libOblivious for a Path ORAM implementation, and allow the user to select between ORAM and oblivious scanning, depending on the usage scenario.

9 Conclusion

This report introduced libOblivious, a software library which exposes C and C++ APIs to facilitate memory-trace oblivious computation. libOblivious provides efficient memory-trace oblivious primitives, upon which more complex oblivious algorithms can be built. The library currently exposes some basic algorithms, and we hope to add more in the future. We also introduced the `O` template, a wrapper which transforms an ordinary C++ STL iterator into an oblivious iterator. This makes it easy for developers to seamlessly integrate libOblivious into legacy C++ codebases.

The performance of libOblivious' primitive operations compares favorably against naïve scanning implementations, and also against the C++ standard library. The libOblivious algorithms do not perform as well as their C++ STL counterparts, but most of them do not perform poorly. We plan to address larger performance gaps—such as with oblivious sorting—in future work.

References

[1] Algorithms library. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm. Accessed: 6 August 2018.

[2] Intel® Software Guard Extensions (Intel® SGX). [Online]. Available: https://software.intel.com/en-us/sgx/details. Accessed: 9 June 2018.

[3] Introduction to x64 assembly. [Online]. Available: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Introduction_to_x64_Assembly.pdf. Accessed: 5 August 2018.

[4] Security on ARM TrustZone. [Online]. Available: https://www.arm.com/products/security-on-arm/trustzone. Accessed: 9 June 2018.

[5] Temporary proxy. [Online]. Available: https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Temporary_Proxy. Accessed: 1 August 2018.

[6] DontUseInlineAsm. [Online]. Available: https://gcc.gnu.org/wiki/DontUseInlineAsm, April 2016. Accessed: 3 August 2018.

[7] Microsoft macro assembler reference. [Online], November 2016. Available: https://docs.microsoft.com/en-us/cpp/assembler/masm/microsoft-macro-assembler-reference. Accessed: 5 August 2018.

[8] Overview of x64 calling conventions. [Online], November 2016. Available: https://docs.microsoft.com/en-us/cpp/build/overview-of-x64-calling-conventions. Accessed: 5 August 2018.

[9] M. A. N. Abrishamchi, A. H. Abdullah, A. David Cheok, and K. S. Bielawski. Side channel attacks on smart home systems: A short overview. In *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 8144–8149, Oct 2017.

[10] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[11] Leonardo Borges. Program optimization through loop vectorization. [Online]. Available: https://software.intel.com/en-us/articles/program-optimization-through-loop-vectorization, January 2014. Accessed: 11 August 2018.

[12] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, 2017.

[13] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge aes against cache-based software side channel vulnerabilities, 2006. jean-pierre.seifert@intel.com 13192 received 13 Feb 2006.

[14] Catalin Cimpanu. New intel cpu cache architecture boosts protection against side-channel attacks. [Online]. Available: `https://www.bleepingcomputer.com/news/security/new-intel-cpu-cache-architecture-boosts-protection-against-side-channel-attacks/`, July 2017. Accessed: 13 August 2018.

[15] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual—combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c*, 2013. No. 325462-048.

[16] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, January 1967.

[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[18] Jonas Devlieghere. Guaranteed copy elision. [Online]. Available: `https://jonasdevlieghere.com/guaranteed-copy-elision/`, November 2016. Accessed: 13 August 2018.

[19] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12, pages 3–8, New York, NY, USA, 2012. ACM.

[20] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 357–380, Cham, 2017. Springer International Publishing.

[21] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In *RAID*, 2017.

[22] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 182–194, New York, NY, USA, 1987. ACM.

[23] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[24] Howard Hinnant. Allocator boilerplate. [Online]. Available: `https://howardhinnant.github.io/allocator_boilerplate.html`, August 2016. Accessed: 12 August 2018.

[25] ISO/IEC. Programming Language C++ (C++17 final draft). Technical Report N4659, International Organization for Standardization (ISO), Geneva, Switzerland, March 2017. Retrieved from `https://isocpp.org/std/the-standard`.

[26] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 506–525, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[27] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[28] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM.

[29] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Not.*, 35(5):145–156, May 2000.

[30] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 51–65, June 2013.

[31] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 87–101, New York, NY, USA, 2015. ACM.

[32] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 359–376, Washington, DC, USA, 2015. IEEE Computer Society.

[33] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, March 2016.

[34] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[35] Jonathan Müller. AllocatorAwareContainer: Introduction and pitfalls of propagate_on_container_XXX defaults. [Online]. Available: `https://foonathan.net/blog/2015/10/05/allocatorawarecontainer-propagation-pitfalls.html`, October 2015. Accessed: 12 August 2018.

[36] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1570–1581, New York, NY, USA, 2015. ACM.

[37] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, Austin, TX, 2016. USENIX Association.

[38] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[39] Kevin Parrish. Microsoft's latest windows 10 patch will address spectre variant 2 cpu flaw. [Online], March 2018. Available: `https://www.digitaltrends.com/computing/microsoft-windows-patch-skylake-spectre-variant-2/`. Accessed: 4 August 2018.

[40] Aaron Pressman. Why your web browser may be most vulnerable to spectre and what to do about it. [Online], January 2018. Available: `http://fortune.com/2018/01/05/spectre-safari-chrome-firefox-internet-explorer/`. Accessed: 4 August 2018.

[41] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., 2015. USENIX Association.

[42] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[43] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. *IACR Cryptology ePrint Archive*, 2017:549, 2017. `https://eprint.iacr.org/2017/549`.

[44] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54, May 2015.

[45] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. [Online]. Available: `https://misc0110.net/web/files/netspectre.pdf`, July 2018.

[46] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. *CoRR*, abs/1106.3652, 2011.

[47] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In

Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[48] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[49] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 215–226, New York, NY, USA, 2014. ACM.

[50] Tom Warren. Intel processors are being redesigned to protect against spectre. [Online], March 2018. Available: `www.theverge.com/2018/3/15/17123610/intel-new-processors-protection-spectre-vulnerability`. Accessed: 4 August 2018.

[51] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 977–988, New York, NY, USA, 2012. ACM.

[52] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society.

[53] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. *IACR Cryptology ePrint Archive*, 2015:1153, 2015.

A Oblivious k-NN Implementation

```
typedef struct KNN_Entry {
  int category;
  unsigned int num_attributes;
  double *attributes;
} KNN_Entry_t;

static void classify_entry_oblivious(unsigned k, unsigned num_categories,
                                     KNN_Entry_t *entry,
                                     const KNN_Entry_t *training_set,
                                     unsigned int training_set_size) {
  struct pair {
    const KNN_Entry_t *entry;
    double distance;
  };

  auto euclidean_distance = [](auto x, auto y, std::size_t len) {
    double distance = 0;
    for (int i = 0; i < len; ++i) {
      distance += (x[i] - y[i]) * (x[i] - y[i]);
    }
    // safe on x86; std::sqrt() uses the vsqrtsd instruction
    return std::sqrt(distance);
  };

  pair *neighbors = new pair[training_set_size];
  for (unsigned int i = 0; i < training_set_size; ++i) {
    neighbors[i] = {training_set + i,
                    euclidean_distance(training_set[i].attributes,
                                       entry->attributes,
                                       entry->num_attributes)};
  }
  auto cmp = [](const pair &p1, const pair &p2) {
    return p1.distance < p2.distance;
  };
  oblivious::osort(neighbors, neighbors + training_set_size, cmp);

  oblivious::ovector<unsigned> class_votes(num_categories);
  oblivious::O optr{class_votes.begin(), &class_votes};
  for (int i = 0; i < k; ++i) {
    int category = neighbors[i].entry->category;
    optr[category] = optr[category] + 1;
  }
  entry->category =
    oblivious::omax_element(class_votes.begin(), class_votes.end(),
                            &class_votes) - class_votes.begin();

  delete[] neighbors;
}
```