Oblivious Immutable Memory

Ananya Appan
UIUC
aappan2@illinois.edu

David Heath
UIUC
daheath@illinois.edu

Abstract

An oblivious RAM (ORAM) compiler is a cryptographic tool that transforms a program $\mathcal{P}$ running in time n into an equivalent program $\tilde{\mathcal{P}}$, with the property that the sequence of memory addresses read from/written to by $\tilde{\mathcal{P}}$ reveal nothing about $\tilde{\mathcal{P}}$'s data (Goldreich and Ostrovsky, JACM'96). An efficient ORAM compiler $\mathcal{C}$ should achieve some combination of the following:

- **Low bandwidth blow-up.** $\tilde{\mathcal{P}}$ should read/write a similar amount of data as does $\mathcal{P}$.
- **Low latency.** $\tilde{\mathcal{P}}$ should incur a similar number of roundtrips to the memory as does $\mathcal{P}$.
- **Low space complexity.** $\tilde{\mathcal{P}}$ should run in as few words of local memory as possible.

It is well known that for a generic compiler (i.e. one that works for any RAM program $\mathcal{P}$), certain combinations of efficiencies are impossible. Any generic ORAM compiler must incur $\Omega(\log n)$ bandwidth blow-up, and any ORAM compiler with no latency blow-up must incur either $\Omega(\sqrt{n})$ bandwidth blow-up and/or local space. Moreover, while a $O(\log n)$ bandwidth blow-up compiler is known, it requires the assumption that one-way functions exist and incurs enormous constant factors.

To circumvent the above problems and improve efficiency of particular ORAM programs, we develop a compiler for a *specific class* of programs. Let $\mathcal{P}$ be a program that interacts with an *immutable memory*. Namely, $\mathcal{P}$ may write values to memory, then read them back, but it cannot change values that were already written. Using only information-theoretic techniques, we compile any such $\mathcal{P}$ into an oblivious form $\tilde{\mathcal{P}}$ with a combination of efficiencies that no generic ORAM compiler can achieve:

- $\tilde{\mathcal{P}}$ incurs $\Theta(\log n)$ amortized bandwidth blow-up.
- $\tilde{\mathcal{P}}$ incurs $O(1)$ amortized latency blow-up.
- $\tilde{\mathcal{P}}$ runs in $O(\lambda)$ words of local space ($\tilde{\mathcal{P}}$ incurs an error with probability $2^{-\Omega(\lambda)}$).

We show that this, for instance, implies that any pure functional program can be compiled with the same asymptotics.

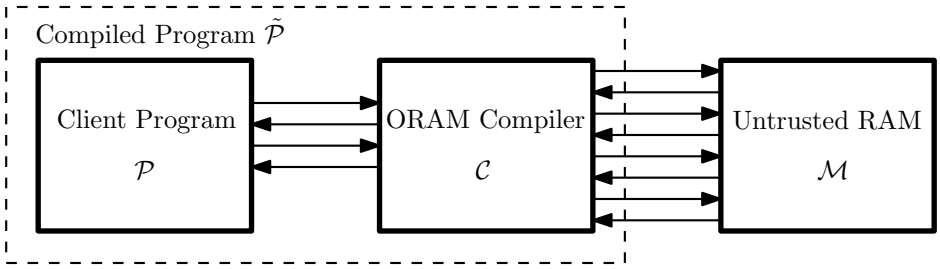
Our work builds on and is compatible with prior work (Appan et al., CCS'24) that showed similar results for pointer machine programs that manipulate objects with constant in-degree (i.e., the program may only maintain a constant number of pointers to any one memory cell; our immutable memory approach does not have this limitation). By combining techniques, we can consider programs that interact with a *mixed memory* that allows each memory cell to be updated until it is *frozen*, after which it becomes immutable, allowing further reads to be compiled with the above asymptotics, even when in-degree is high. Many useful algorithms/data structures can be naturally implemented as mixed memory programs, including suffix trees (powerful data structures used in computational biology) and deterministic finite automata (DFAs).

Keywords: ORAM, Oblivious Algorithms and Data Structures

Contents

1 Introduction

Oblivious RAM. An Oblivious RAM (ORAM) compiler is a cryptographic tool enabling a memory-constrained client to securely outsource its data storage to an untrusted server [GO96]. The challenge in constructing an ORAM compiler—and our focus in this work—is that the compiler must obfuscate the client's *memory access pattern*. An ORAM compiler provides a *generic* approach to obfuscating access patterns. Let $\mathcal{P}$ be an arbitrary client program that issues random access memory queries. An ORAM compiler $\mathcal{C}$ transforms $\mathcal{P}$ into a new program $\tilde{\mathcal{P}} = \mathcal{P} \leftrightarrow \mathcal{C}$:



$\tilde{\mathcal{P}}$ computes the same function as $\mathcal{P}$, but $\tilde{\mathcal{P}}$'s memory access pattern[1] hides client data, allowing the client to upload/download data from the server while preserving privacy. We refer to $\tilde{\mathcal{P}}$ as an *oblivious RAM program*.

Applications of Oblivious RAM. The ability to hide memory access patterns has found several important applications. Hardware enclaves, e.g. Intel SGX, leak memory access patterns that attackers can exploit to infer sensitive information about even encrypted data stored in memory [CGPR15, ZKP16, GMN+16]; an ORAM compiler can suppress this leakage [SGF17, MPC+18]. ORAM is also a central in the study of secure Multi Party Computation (MPC) protocols, which enable mutually untrusting parties to securely compute functions on their joint private data [Yao86, GMW87]. Classically, MPC protocols require that the parties encode their program as a boolean or arithmetic circuit, but by correctly incorporating an ORAM compiler, one can also achieve MPC protocols for arbitrary RAM programs. The combination of MPC and ORAM has been studied extensively; see [OS97, GKK+12, KM19, BKKO20, HV21, FNO22, VHG23, FNO24] and others.

Cost metrics and our setting. The efficiency of an ORAM compiler is measured in terms of two key metrics: bandwidth blowup and roundtrips. Bandwidth blowup measures the (amortized) number of RAM requests that $\tilde{\mathcal{P}}$ issues for each RAM request issued by $\mathcal{P}$. One roundtrip corresponds to $\tilde{\mathcal{P}}$ sending a batch of RAM requests, then receiving a batch of responses. We measure cost in terms of the number of roundtrips made per access.

ORAM is typically studied in the client-server setting where $\tilde{\mathcal{P}}$ runs on a client with small local memory and issues queries to an untrusted server that emulates a RAM. We, in particular, will consider compilers that run in at most $O(\lambda)$ words of space, where λ is a security parameter that will control failure probability. We focus on the classic ORAM setting where the server simply implements a read/write memory, and the server *does not* perform other useful computation, such as running operations under homomorphic encryption [DvDF+16].

The Challenge of Designing Efficient ORAM. Unfortunately, a generically-compiled oblivious RAM program $\tilde{\mathcal{P}}$ inherently requests more data from the server than the insecure RAM program $\mathcal{P}$, incurring considerable bandwidth blowup and roundtrips. Any generic ORAM compiler must incur a bandwidth blowup of $\Omega(\log n)$ [GO96, LN18]. A long line of work—e.g. [GO96, GM11, SCSL11, SvDS+13, PPRY18] and others—improved bandwidth blowup, ultimately leading to the remarkable achievement of OptORAMa [AKL+20], a generic ORAM compiler achieving the optimal $\Theta(\log n)$ bandwidth blow-up.

While OptORAMa achieves optimal generic compilation, it leaves much to be desired. OptORAMa suffers an enormous hidden constant in its bandwidth blow-up (≈ 9400, thanks to the significant improvement of [DO20]), and requires a cryptographic assumption—the existence of one-way functions. By contrast, the more practical Path ORAM [SvDS+13] incurs a small hidden constant, is used in practice by Signal for private contact discovery [Sig], and does not rely on cryptographic assumptions, but it incurs a sub-optimal bandwidth blowup of $O(\log^2 n)$ when $\mathcal{P}$

[1] Ultimately, the *content* of client data should, of course, also be hidden. In the client/server setting, it is sufficient to encrypt client data with any CPA-secure encryption scheme. In this work we ignore memory content and hide only the memory access pattern both because it is the far more interesting problem, and for generality. Indeed, in some settings encryption is not needed to hide client data, such as when ORAM is used as part of an MPC protocol and the data is secret-shared.

manipulates words of standard size $w = \Theta(\log n)$. The construction of an ORAM scheme with optimal bandwidth blowup without relying on cryptographic assumptions has remained open for decades.

Designing ORAM compilers with low roundtrips also proves challenging. Both OptORAMa and Path ORAM require $O(\log n)$ roundtrips. A lower bound [CDH20] shows that ORAM compilers that incur a single roundtrip either cause $O(\sqrt{n})$ bandwidth blowup or require $O(\sqrt{n})$ client storage, unless expensive server computation is used, indicating that the first ever ORAM scheme proposed [GO96] is in some sense optimal. Thus, reducing roundtrips comes with inherent tradeoffs.

Designing ORAM Compilers for Special-Case Problems. Since *generic* ORAM compilers necessarily incur high cost, it is natural to consider designing ORAM algorithms/compilers for *restricted* classes of programs. Prior work has shown that many useful programs can be handled obliviously with better efficiency than can be achieved by a generic ORAM compiler. Stacks, queues, linked lists, binary trees, and even particular graph algorithms can be handled with $O(\log n)$ bandwidth blow-up and $O(1)$ roundtrip blow-up as compared to their insecure baseline [ZE13, WNL$^+$14, AHR24].

Indeed, the most recent work on special-case oblivious RAM compilers showed that any *pointer-machine program* [KU58, Sch80] $\mathcal{P}$ can be compiled to an oblivious RAM program $\tilde{\mathcal{P}}$ with good efficiency using an oblivious pointer machine (OPM) compiler [AHR24]. The pointer machine is a natural model of computation where one can allocate objects in memory, then look those objects up from memory later by means of dereferencing a pointer. The restriction on pointer machine (PM) programs as compared to standard RAM programs is that one may not perform arbitrary operations on pointers. For instance, adding two pointers together is ill-defined.

The OPM compiler presented in [AHR24] builds on top of Path ORAM. Path ORAM continuously shuffles memory by mapping each RAM address to a uniformly random address that is re-mapped on every access. This mapping is recursively implemented using $O(\log n)$ smaller ORAMs—jointly called the *position map*—contributing a $O(\log n)$ factor to both bandwidth blowup and roundtrips. [AHR24]'s insight, which extends that of [WNL$^+$14], is that an OPM compiler need not maintain a position map since only a fixed number of addresses are reachable from a value for a pointer machine program, and the compiler has to keep track of the mapping only for these addresses.

To dereference a pointer, $\tilde{\mathcal{P}}$ incurs a bandwidth blowup of $O(\log n \log d)$ and $O(\log d)$ roundtrips, where d is the *fan-in* of the value being pointed to. Also called *degree*, this is the number of pointers that point to the value. A *constant-degree* pointer machine program ensures that each value in memory has constant degree. Since the compiler presented in [AHR24] is ultimately a simplification of Path ORAM, it is efficient both in theory and practice, without relying on cryptographic assumptions. While this implies significant improvement in cost for constant-degree PM programs, the bandwidth blowup and roundtrips incurred for programs that manipulate values with degree of $O(n)$ is no better than that of Path ORAM.

1.1 Our Contribution

Immutable Memory. In this work, we develop new techniques for obfuscating the memory access pattern of programs that manipulate an *immutable memory*. An immutable memory program may write fresh values to the memory, but once a memory slot is written, it cannot be subsequently changed [2]. Our main technical contribution is an oblivious immutable memory (OIM) compiler that converts any program that manipulates immutable memory into an oblivious RAM program. Our compiler incurs only $O(\log n)$ bandwidth blowup and requires $O(1)$ roundtrips for each request to immutable memory, *regardless of the degree* of the value being fetched. Existing lower bounds imply that this cost is optimal [JLN19].

Many algorithms can naturally be implemented as immutable memory programs, leading to new asymptotic results for oblivious algorithms. For instance, Church's foundational lambda calculus is straightforwardly implemented as an immutable memory program, so our work leads to a compiler from purely functional programs to oblivious RAM programs, where the oblivious RAM program incurs $O(\log n)$ bandwidth blowup and $O(1)$ roundtrips.

Lambda calculus and pure functional programming have proved indispensable to the study of programming language design; see e.g. [Hic25]. The lambda calculus is easily extensible, and indeed, several relatively mainstream programming languages are essentially minor extensions of the lambda calculus [M$^+$10, Mil97] . Our work implies an improved oblivious compiler for standard programs written in such languages.

All functional data structures [Oka98]—that is, those structures that can be constructed on top of an immutable memory—are automatically *persistent*, meaning that older versions of the data structure are available even after

[2]Unlike prior work [TJS19], we do not require that all contents of memory be decided at the start of program execution. New values can be written to memory even during program execution, but once written, cannot be changed.

Table 1: Amortized communication and roundtrips of oblivious algorithms based on immutable memory in comparison to state-of-the-art solutions for ORAM and [AHR24] for words of size $\Theta(\log n)$. The cost listed for [AHR24] assumes the worst case cost for values with arbitrary degree.

Algorithm	OptORAMa		Path ORAM / [AHR24]		This Work	
	Communication*	Rounds	Communication	Rounds	Communication	Rounds
λ-expression evaluation	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg^2 n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
Ukkonen's Algorithm	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg^2 n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
DFA Evaluation	$O(n \lg \mathcal{Q})$	$O(n \lg \mathcal{Q})$	$O(n \lg^2 \mathcal{Q})$	$O(n \lg \mathcal{Q})$	$O(n \lg \mathcal{Q})$	$O(n)$

* Involves a hidden constant ≈ 9400 [DO20].
n denotes - λ *evaluation:* The time required to evaluate the lambda expression, *Ukkonen's Algorithm:* The length of the string for which the suffix tree is built, *DFA Evaluation:* The length of the string input to the DFA.

updates. Using our compiler, all such data structures can be made oblivious at $O(\log n)$ blow-up. An oblivious lambda calculus compiler also has interesting use-cases for secure Multi Party Computation (MPC). Protocols that achieve MPC typically use the circuit model of computation. Our compiler enables MPC using lambda calculus instead. Due to its extensible nature, the lambda calculus is a compelling model of computation for MPC.

Mixed Memory. It is also interesting to consider programs that manipulate both pointer machine memory (i.e., mutable memory) and immutable memory. Namely, consider a single program that handles both mutable and immutable addresses. Pointees of mutable addresses can be updated; pointees of immutable addresses cannot. We give an obliviousness compiler for any such program, allowing one to combine the increased expressivity of mutability with the increased efficiency of immutability.

Formally, we consider a *mixed-memory programs* that allow for the allocation of a mutable address which can be made immutable using a `freeze` operation. This allows, for instance, the construction of a data structure with loops (e.g. directed cyclic graphs) using operations on mutable addresses. Once updates are no longer required, the data structure can be made more efficient by freezing it. We present an oblivious mixed memory (OMM) compiler that compiles the `freeze` operation into an oblivious RAM program that incurs $O(\log n)$ bandwidth blowup and $O(1)$ roundtrips. The cost of other operations depends on the type of address used: the cost of operations on mutable and immutable addresses are comparable to the cost of using an OPM and an OIM compiler respectively.

Mixed memory has multiple applications. Using our OMM compiler, any program that can be written as a combination of a constant-degree PM operations and immutable memory operations can be compiled into an oblivious RAM program with $O(\log n)$ bandwidth blowup and $O(1)$ roundtrip blowup. One important example is that of Ukkonen's algorithm [Ukk95]. This algorithm constructs the *suffix tree* for a string of length n in time $O(n)$. Suffix trees[3] are the basis for many string algorithms useful, for example, in computational biology. Example uses of suffix trees include identifying all occurrences of a set of substrings within a given string, detecting the longest repeated substring, and many others. Oblivious versions of these algorithms are useful when performed on sensitive data such as DNA sequences. Oblivious substring search is also useful for designing substring searchable encryptions schemes [CS15a] with access pattern privacy. We show that Ukkonen's algorithm can be interpreted as a mixed-memory program where all mutable memory values have constant degree, and hence we can compile the algorithm with $O(\log n)$ blow-up.

A deterministic finite automaton (DFA) can also be implemented as a mixed memory program. Specifically, once a DFA for a language L is constructed by performing $O(\texttt{poly}(\mathcal{Q}))$ operations on mutable addresses, the DFA can be frozen. From here, we can check whether a string of length n belongs to L using $O(n)$ operations on immutable addresses. This can be compiled into an oblivious RAM program with $O(\log \mathcal{Q})$ bandwidth blowup and $O(1)$ roundtrip blowup when the input string has large length, or when multiple strings are evaluated.

Of course, these are just the applications we have found, and there may be others. We feel that this collection of non-trivial applications demonstrates the flexibility of mixed-memory programs.

[3]While the data structure is called a suffix *tree*, the structure is in fact a *cyclic graph* where certain nodes may have in-degree as high as $O(n)$. In particular, a suffix tree is a tree with edges pointing both down the tree—from parents to children—and back up the tree. Nodes in this tree point into a string, and there is no bound (other than the trivial $O(n)$ bound) on the number of pointers to any particular index of the string.

1.2 Technical Overview

Recall that [AHR24] showed an efficient obliviousness compiler for constant-degree pointer-machine (PM) programs. We denote this compiler by `OPM`. Our oblivious immutable memory compiler `OIM` is constructed by first compiling the target immutable memory program into a constant-degree PM program. We denote our compiler from IM programs to PM programs by $\mathcal{C}_{IM}$. With $\mathcal{C}_{IM}$ constructed, we simply apply `OPM` to construct an obliviousness compiler `OIM` for arbitrary immutable memory programs:

$$\mathcal{P} \leftrightarrow \overbrace{\mathcal{C}_{IM} \leftrightarrow \texttt{OPM}}^{\texttt{OIM}} \leftrightarrow \texttt{RAM}$$

Our construction of $\mathcal{C}_{IM}$ centers on the following key idea: Because values in memory are *immutable*, we can safely make *copies* of values, without worrying that these different copies will become inconsistent. By using this insight, we can, with care, ensure that no particular copy has more than a constant number of incoming pointers.

In more detail, our construction copies an immutable memory value once its degree exceeds a constant threshold. Half of the pointers that pointed to the original value are made to point to the copy instead. This ensures that the degree of each value in memory remains constant, thereby allowing for an implementation that uses a constant-degree PM. There is one subtle but crucial point to this strategy: When copying a value, the copied value may itself contain pointers to other values. Thus, when we copy the value, we increase the degree of these other values, potentially resulting in a cascading construction of large numbers of copies throughout memory. Despite this, our amortized analysis (based on the potential method) shows that the cost of copying a pointer into immutable memory can be bounded to $O(1)$ PM operations.

Mixing mutable and immutable memory. The compiler presented in [AHR24] implements PM programs of non-constant degree by compiling them to PM programs of constant-degree using a compiler $\mathcal{C}_{PM}$. In this compilation step, multiple pointers are made to point to a value via a constant-degree balanced binary tree of pointers with the value at the root, and with each pointer pointing to a leaf. A pointer is dereferenced by traversing a path from the leaf to the root of this tree to fetch the value.

Because both arbitrary-degree pointer machines and immutable memory programs can be compiled to operations on a constant-degree pointer machine, it is possible to combine these compilation strategies and obtain obliviousness for a single program that utilizes *both* arbitrary-degree *mutable* pointers and arbitrary-degree *immutable* pointers. Dereferencing a mutable memory value with degree d incurs a number of constant-degree pointer machine operations scaling with $O(\log d)$; dereferencing an immutable value incurs $O(1)$ operations, regardless of the value's degree. To clarify, the entire execution of this *mixed-memory* (MM) program is ultimately made oblivious, and the server cannot tell whether the client is using mutable or immutable memory slots.

Our oblivious MM (OMM) compiler is implemented as a constant-degree PM program $\mathcal{C}_{MM}$ that simply invokes $\mathcal{C}_{PM}$ for operations on mutable addresses, and invokes $\mathcal{C}_{IM}$ for operations on immutable addresses. As a nontrivial detail, $\mathcal{C}_{MM}$ additionally implements a `freeze` operation, which converts a mutable value—and all values reachable from that value—into an immutable one. This is done by recursively re-arranging addresses and values from the format required for $\mathcal{C}_{PM}$ into the format required for $\mathcal{C}_{IM}$. Specifically, when a mutable address is frozen, a traversal of the tree of pointers pointing to the value is performed. Pointers that point to the leaves of this tree are instead made to point to copies of the value that are created based on the value's degree. Our amortized analysis shows that a mutable address can be frozen using $O(1)$ PM operations.

Organization. We discuss preliminaries in §2. §3 presents our OIM compiler and §4 presents our OMM compiler. We discuss applications in §5 and related work in §6.

2 Preliminaries

2.1 Notation

- λ denotes a security parameter that controls failure probability. We assume $\lambda > \log n$. Indeed, to achieve negligible failure probability it should be that $\lambda = \Omega(\log^{1+\epsilon} n)$ for some positive ϵ.
- $X \equiv Y$ denotes that X and Y are identically distributed.
- $X \approx Y$ denotes that probability ensembles X and Y are statistically close in λ.

- $\mathcal{P} \leftrightarrow \mathcal{Q}$ denotes an interaction between interactive randomized algorithms $\mathcal{P}$ and $\mathcal{Q}$, where $\mathcal{P}$ issues requests that are handled by $\mathcal{Q}$. We often view $\mathcal{P} \leftrightarrow \mathcal{Q}$ itself as an interactive randomized algorithm; the output of this composed algorithm is the output of $\mathcal{P}$ after interacting with $\mathcal{Q}$. The $\leftrightarrow$ operation is associative, so when composing multiple algorithms we omit brackets.
- A ***machine*** $\mathtt{M}$ refers to an interactive algorithm that handles memory requests. A memory request is issued using an *address* that is used to either write a value to or read a value from memory.
- A ***program*** $\mathcal{P}$ is an algorithm that issues memory requests to a machine. The memory requests that the program can issue are constrained by the kind of machine that the program interacts with. We write $\mathcal{P} \leftrightarrow \mathtt{M}$ to denote the interaction between the program and machine. The ***local space*** of a program refers to the memory used by the program outside of the memory provided by the machine it interacts with, and includes variables used by the program. We do not specify inputs and assume that inputs are hard-coded as part of the program. The costs of our compilers are based on the runtime of the *program*.
- A ***compiler*** $\mathcal{C}$ is used to compile requests issued by a program $\mathcal{P}$ to machine $\mathtt{M}_1$ into a program $\mathcal{P} \leftrightarrow \mathcal{C}$ that instead issues requests to a machine $\mathtt{M}_2$, while ensuring that $\mathcal{P} \leftrightarrow \mathtt{M}_1 \equiv \mathcal{P} \leftrightarrow \mathcal{C} \leftrightarrow \mathtt{M}_2$ holds.
- We write "*p points to v*" and "*v* is the ***pointee*** of *p*" to mean that a value v is written to ***address*** p. From here on, the term address is used for any machine, including pointer machines. The ***degree*** of a value is the number of addresses pointing to it.
- We assume a word size of $\Theta(\log n)$ bits, and that each value contains a constant number of words.

2.2 Definitions

Oblivious RAM Programs. An oblivious RAM program is a RAM program with a memory access sequence that can be simulated using just the length of the program.

Definition 1 (Oblivious RAM Program)**.** *A RAM program $\mathcal{P}$ that runs in n steps is an oblivious RAM program if there exists a poly-time simulator* $\mathtt{SIM}$ *that takes n as input and simulates the memory addresses accessed by $\mathcal{P}$ i.e.,*

$$\mathtt{addrs}(1^\lambda, \mathcal{P}) \approx \mathtt{SIM}(1^\lambda, n)$$

Here, $\mathtt{addrs}$ *denotes to the sequence of addresses requested by $\mathcal{P}$ while interacting with random access machine* $\mathtt{RAM}$*.*

The ORAM compilers most relevant to this work belong to a category of ORAMs called tree-based ORAMs [SCSL11, SvDS+13, WCS15, RFK+15]. Tree-based ORAMs store the memory content as a tree of nodes, with each node storing a small number of memory elements. Each memory address is mapped to a uniformly random leaf in the tree, with the invariant that the pointee lies somewhere on the path from the root to this leaf. This mapping is stored in a data structure called the position map, which is recursively implemented using $O(\log n)$ smaller ORAMs.

Opaque Address Programs. This work considers programs that are more restrictive than RAM programs, and that can hence be compiled into an oblivious RAM program more efficiently. To perform this compilation, it is necessary to restrict the types of operations that can be performed on memory addresses. For instance, we may restrain the program from performing arithmetic operations on addresses. One way to achieve this would be to completely specify a programming language in which programs can be legally written. Seeking a lighter approach, we instead only specify instructions that can be performed on *addresses* that are used to interact with the machine. Looking ahead, this is required since our compilers use the OPM compiler presented in [AHR24] that works only when operations performed on addresses are restricted. Thus, we define the notion of an *opaque address program*.

Definition 2 (Opaque Address Program)**.** *An* ***opaque address program*** *is one that (1) partitions its local space into memory addresses and (non address) data words and (2) performs operations on memory addresses only by sending them to the memory machine that it interacts with. In particular, an opaque address program* may not *alter the bits of an address or conditionally branch depending on the bits internal to an address.*

Pointer Machine Programs. A pointer machine program is an opaque address program that is restricted to use addresses that are either allocated in advance, or are copies of existing addresses. In particular, addresses cannot be decided arbitrarily, nor computed using arithmetic.

Definition 3 (Pointer Machine Program)**.** *A **pointer machine program** $\mathcal{P}$ is an opaque address program whose operations on addresses are restricted to the following instructions. An address can be initialized using the following:*

$p := \texttt{alloc}$	*allocate fresh address p; p points to the all-zeros tuple*
$q, r := \texttt{copy}\ p$	*return fresh addresses q and r that each point to p's pointee*

The following instructions can be performed on an initialized address:

$\texttt{write}\ p\ (x_1, ..., x_k)$	*overwrite p's pointee with $(x_1, ..., x_k)$*
$(x_1, ..., x_k) := \texttt{read}\ p$	*load p's pointee into variables $(x_1, ..., x_k)$*

We say that an address becomes **invalidated** *if it is used as the input address for a* `copy`*,* `write`*, or* `read` *instruction. We say that $\mathcal{P}$ is **valid** if it never passes an invalid address as an argument to* `read`*/*`write`*/*`copy`*. From here on, we only consider/define valid pointer machine programs. We define the **pointer machine*** `PM` *as the interactive algorithm that handles the above memory requests in the natural manner.*

We note that the above definition writes and reads tuples from pointer machine memory. This is used to generically represent an object that may contain a combination of values of varied data types, including addresses.

The above definition is slightly different from the traditional notion of a pointer machine program, since it restricts addresses to be used only once. We place this requirement because we need the program to explicitly label when it copies addresses; the input of such calls to `copy` will be passed as input to a compiler, which handles the copy operation in a black-box manner, then returns two fresh addresses.

Note that the requirement to explicitly label where addresses are copied is clearly without loss of generality because the ability to create copies indirectly allows for the repeated use of addresses. For instance, the program can copy an address, write to one copy, and keep the other copy for subsequent operations:

$p, q := \texttt{copy}\ p$	q is a copy of p
$\texttt{write}\ q\ (x_1, ..., x_k)$	overwrite q's pointee with $(x_1, ..., x_k)$

A `read` instruction that allows for the repeated use of addresses can be defined similarly.

A ***constant-degree PM program*** or an ***$O(1)$-PM program*** is a pointer machine program that ensures that the degree of any value written to PM memory is bounded by a constant.

Single Access Machine Programs. A single access machine (SAM) program [AHR24] is a pointer machine program that does not allow addresses to be copied, and restricts that each address can be read / written at most once[4], thereby truly enforcing single access. An OSAM compiler that compiles a SAM program into an oblivious RAM program can be implemented as a tree-based ORAM *without the position map* [AHR24], thus reducing bandwidth blowup and roundtrips by a factor of $\log n$. Recall that the position map is a data structure that maps RAM addresses (that may be used repeatedly) to a different uniformly random leaf in the ORAM tree each time the RAM address is used. At a high level, since SAM addresses are used only once, an OSAM compiler need not maintain this mapping; refer to [AHR24] for details.

Oblivious Pointer Machine Compilers. [AHR24] present compilers that compile any $O(1)$-PM program $\mathcal{P}$ into an oblivious RAM program by first compiling $\mathcal{P}$ to a SAM program, and then compiling the SAM program into an oblivious RAM program. Based on the underlying tree ORAM used to implement OSAM, two variants of compilers are obtained: `Path-OPM` that builds on Path ORAM [SvDS+13] and `Circuit-OPM` that builds on Circuit ORAM [WCS15], an ORAM compiler designed for MPC.

Theorem 1 (Oblivious PM [AHR24])**.** *There exists interactive RAM programs* `Path-OPM` *and* `Circuit-OPM` *that compile $O(1)$-PM programs into oblivious RAM programs. For every $O(1)$-PM program $\mathcal{P}$ running in time n and for statistical security parameter λ, the following hold:*

- ***Correctness.*** *The following are statistically close in λ:*

$$\mathcal{P} \leftrightarrow \texttt{PM} \approx \mathcal{P} \leftrightarrow \texttt{Path-OPM} \leftrightarrow \texttt{RAM}$$
$$\mathcal{P} \leftrightarrow \texttt{PM} \approx \mathcal{P} \leftrightarrow \texttt{Circuit-OPM} \leftrightarrow \texttt{RAM}$$

[4]More specifically, an *invalid* address is defined slightly differently for a SAM program: once an address is used in an operation, it cannot be used to perform the *same* operation again, but can be used in a different operation.

- ***Space complexity.*** `Path-OPM` *runs in* $O(\lambda)$ *local space;* `Circuit-OPM` *runs in* $O(1)$ *local space.*
- ***Communication complexity.***
  - *The interaction between compiled program* $\mathcal{P} \leftrightarrow$ `Path-OPM` *and random access memory* `RAM` *involves* $O(n)$ *roundtrips and* $O(n \lg n)$ *words of communication.*
  - *The interaction between compiled program* $\mathcal{P} \leftrightarrow$ `Circuit-OPM` *and random access memory* `RAM` *involves* $O(n\lambda)$ *roundtrips and* $O(n\lambda)$ *words of communication.*
- ***Obliviousness.*** $\mathcal{P} \leftrightarrow$ `Path-OPM` *and* $\mathcal{P} \leftrightarrow$ `Circuit-OPM` *are oblivious RAM programs (Definition 1).*

[AHR24] also presents a compiler that compiles any (non-constant-degree) PM program into a SAM program: the same implementation can also be perceived as a compiler to an $O(1)$-PM program, see §4 for details. The blowup incurred while reading/writing a PM address depends on the degree of the dereferenced pointee.

Theorem 2 (Compiling PM to $O(1)$-PM [AHR24])**.** *There exists an interactive,* $O(1)$*-PM program* $\mathcal{C}_{PM}$ *that implements the semantics of a pointer machine. In particular, for every PM program* $\mathcal{P}$ *running in time* n*, the following hold:*

- ***Correctness.*** *The following are identically distributed:*

$$\mathcal{P} \leftrightarrow \texttt{PM} \equiv \mathcal{P} \leftrightarrow \mathcal{C}_{PM} \leftrightarrow \texttt{O(1)-PM}$$

- ***Space complexity.*** $\mathcal{C}_{PM}$ *runs in* $O(1)$ *local space.*
- ***Runtime complexity.*** *The compiled program* $\mathcal{P} \leftrightarrow \mathcal{C}_{PM}$ *runs in time* $\tilde{n} = O\left(n \cdot \sum_{i=0}^{n} (\lg d_i + 1)\right)$*, where* d_i *denotes the degree of the value read at step* i *of* $\mathcal{P}$*'s computation (if the* i*-th operation is* `alloc`*, then we set* $d_i = 1$ *such that* $\lg d_i = 0$*).*

3 Immutable Memory

An immutable memory (IM) program restricts that an address, once initialized, can only be read. An address can be initialized by writing a value to memory or by copying an existing address.

Definition 4 (Immutable Memory Program)**.** *An* ***immutable memory program*** $\mathcal{P}$ *is an opaque address program whose operations on addresses are restricted to the following instructions. An address can be initialized using the following:*

$p := \texttt{write}(x_1, ..., x_k)$	*write a tuple to memory as the pointee of an address* p
$q, r := \texttt{copy}\ p$	*return fresh addresses* q *and* r *that each point to* p*'s pointee*

The following instructions can be performed on an initialized address:

$(x_1, ..., x_k) := \texttt{read}\ p$	*load* p*'s pointee into variables* $(x_1, ..., x_k)$

We say that an address becomes **invalidated** *if it is used as input in a* `copy` *or* `read` *instruction.* $\mathcal{P}$ *is* ***valid*** *if it never passes an invalid address as an argument to* `read`/`copy`*. From here on, we only consider/define valid immutable memory programs. An* ***immutable memory machine*** `IM` *is the interactive algorithm that handles the above memory requests in the natural manner.*

3.1 Building an Oblivious Immutable Memory Compiler

We present a compiler $\mathcal{C}_{IM}$ that compiles requests made to IM into requests to a $O(1)$-PM program. Our compiler makes only amortized $O(1)$ PM requests per IM request, even if values stored in IM are of arbitrary degree. The resulting PM program can be compiled to an oblivious RAM program using an OPM compiler (Theorem 1).

Approach. The key challenge faced is in implementing IM that stores values of arbitrary degree using a $O(1)$-PM program. $\mathcal{C}_{IM}$ does this by creating multiple copies of a value stored in memory, with each copy having only a constant-degree. Specifically, each time the IM program issues a `copy` request, if the degree of the pointee v exceeds a certain (constant) threshold, $\mathcal{C}_{IM}$ (1) makes a copy of v and (2) redirects half of v's incoming pointers to the copy. The degrees of the both pointee and its copy are now only half of the threshold value, allowing the IM program to issue further `copy` requests.

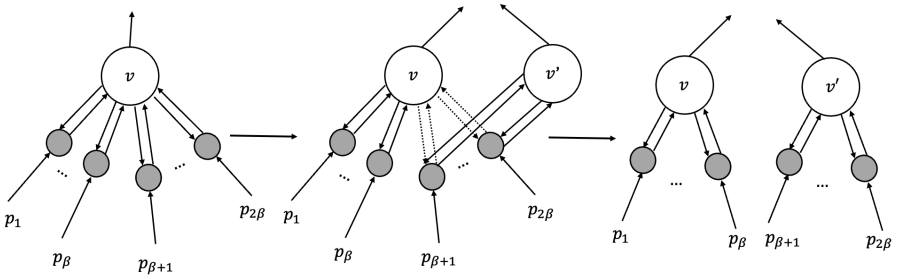


Figure 1: Example of how a value v is copied when its degree equals 2β. v holds addresses to intermediate addresses (stored in gray nodes) that are updated to point to v's copy.

The Importance of Immutability. For the above approach to work, it is crucial to assume that the memory is immutable. If updates to values in memory were allowed, then updating the value of one copy in memory would require all copies to also be updated, incurring too much overhead. Since memory is immutable, there is no semantic difference to the IM program between reading a value and its copy.

Challenges. The above simple approach has two subtle, but important issues. First is the issue of how an IM address p that points to a value v can be implemented. p cannot simply be a PM address that points to v. This is because, if an address copy causes v's degree to exceed the threshold, it is unclear how PM addresses that point to v can be updated to point to v's copy since these addresses are stored at varied locations in memory. Instead, we make p and v hold PM addresses that point to an intermediate address $p_{\texttt{int}}$ that points to v. The address stored in v can be used to update $p_{\texttt{int}}$ to instead point to v's copy (see Figure 1).

The second issue is that a copied value may itself contain IM addresses, and copying the value will recursively create copies of these addresses. This recursion can cascade, possibly creating more copies of other values. Despite this, we show that the amortized cost of a pointer copy can be made constant by choosing the right degree-threshold at which a value is copied. We follow the potential method of amortized analysis: the contribution of a value to the potential is proportional to how much its degree exceeds half of the threshold at which a copy is created. Copying a value causes the degree of both the value and its copy to drop to half. This causes a decrease in potential, accounting for the cost of recursive calls to copy.

Putting Everything Together. We present our compiler $\mathcal{C}_{IM}$ in Figure 2. We present $\mathcal{C}_{IM}$ as a PM program that uses operations that do not invalidate addresses; a program that explicitly creates copies is presented in Appendix A. Concretely, each IM instruction is implemented as follows.

- `write`$(x_1, ..., x_k)$: An intermediate address $p_{\texttt{int}}$ is initialized and made to point to the pointee $(x_1, ..., x_k)$. A copy of $p_{\texttt{int}}$ is stored along with the pointee in a list `ptrs` of addresses that point to intermediate addresses; this can be used to update $p_{\texttt{int}}$ to point elsewhere if needed. An address that points to $p_{\texttt{int}}$ is returned.
- `read` p: p is read first to obtain the intermediate address $p_{\texttt{int}}$. $p_{\texttt{int}}$ is then read to obtain the pointee.
- `copy` p: We first fetch the intermediate address $p_{\texttt{int}}$, then use $p_{\texttt{int}}$ to fetch the pointee v and the list `ptrs` of addresses to intermediate addresses. A copy $q_{\texttt{int}}$ of $p_{\texttt{int}}$ is created, and a new pointer q is made to point to $q_{\texttt{int}}$. A copy of q is also stored in `ptrs`. After creating this copy, if the degree of v is equal to a threshold 2β, a copy v' of v is created, and half of the addresses that pointed to v are made to point to v' using a procedure called `divert`. p is then updated to point to v stored alongside the updated list `ptrs`.

Theorem 3 (Compiling IM to $O(1)$-PM)**.** *There exists an interactive, $O(1)$-PM program $\mathcal{C}_{IM}$ that implements the semantics of a immutable memory. In particular, for every IM program $\mathcal{P}$ running in time n, the following hold:*

- ***Correctness.*** *The following are identically distributed:*

$$\mathcal{P} \leftrightarrow \texttt{IM} \equiv \mathcal{P} \leftrightarrow \mathcal{C}_{IM} \leftrightarrow \texttt{PM}$$

- ***Space complexity.*** *$\mathcal{C}_{IM}$ runs in $O(1)$ local space.*

```
def IM.write(x_1, ..., x_k) → p :
    ptrs := (⊥, ..., ⊥)  // Initialize a 2β-tuple
    p_int := PM.alloc( )  // Intermediate address that
      points to the value
    ptrs.add(p_int)
    p := PM.alloc( )  // address that points to the
      intermediate address
    PM.write(p, p_int)
    PM.write(p_int, ((x_1, ..., x_k), ptrs))
    return p

def IM.read(p) → (x_1, ..., x_k) :
    p_int := PM.read(p)
    (x_1, ..., x_k), ptrs := PM.read(p_int)
    return (x_1, ..., x_k)

def divert(ptrs, v) :
    p_int := alloc( )
    for each p in ptrs do
        PM.write(p, p_int)
    PM.write(p_int, (v, ptrs))
```

```
def IM.copy(p) → q :
    p_int := PM.read(p)
    v, ptrs := PM.read(p_int)
    q_int := PM.copy(p_int)
    q := PM.alloc( )
    PM.write(q, q_int)
    ptrs.add(q)
    if |ptrs| = 2β then
        v' := IM.copy(v)  // recursively copies
          addresses stored in v
        ptrs' := (⊥, ..., ⊥)  // Initialize a 2β-tuple
          filled with ⊥
        for each p in half(ptrs) do
            ptrs.remove(p)
            ptrs'.add(p)
        divert(ptrs', v')
    PM.write(p, (v, ptrs))
    return q
```

Figure 2: Implementation of IM that stores k-tuples using a $O(1)$-PM that stores $k + 2\beta$-tuples. A value is copied when its degree reaches 2β. The last 2β positions of a value in PM memory store addresses to intermediate addresses that point to the value. The procedure `divert` is used to make intermediate addresses point to a value's copy.

- ***Runtime complexity.*** *The compiled program $\mathcal{P} \leftrightarrow \mathcal{C}_{IM}$ runs in time $O(n)$.*

Proof. Correctness follows from inspection of Figure 2. In particular, $\mathcal{C}_{IM}$ is a $O(1)$-PM program provided that β is a constant. We show that by choosing any value $\beta > \texttt{out} + 1$, where `out` is the maximum number of addreses held in a value, performing `copy` invokes amortized $O(1)$ PM requests. It then follows from inspection of Figure 2 that each operation invokes $O(1)$ requests to PM.

We use the potential method to bound the cost of `copy`. Specifically, each value that has a degree $d > \beta$ contributes $c_1 \cdot (d - \beta)$ to the potential, where c_1 is a constant that we will determine later. Note that the cost of copying a value is dependent on $\beta + \texttt{out}$; let this cost be $c_2(\beta + \texttt{out})$ for some constant c_2. We now consider two cases:

- *Case 1: $R \geq 1$ values are copied:* For each value that is copied, the potential decreases by $c_1(2\beta - 1 - \beta) = c_1(\beta - 1)$. Then, the cost of copying an address is upper bounded by:

$R \cdot c_2 \cdot (\beta + \texttt{out})$	the cost of copying R values
$+ R \cdot c_1 \cdot \texttt{out}$	increase in potential by copying addresses contained in R values
$- R \cdot c_1 \cdot (\beta - 1)$	decrease in potential due to the decrease in degree of R values

  This only over-counts the cost since we assume that every address contained in a value is also copied without invoking further recursive calls (in addition to being copied recursively). This cost can be made 0 by carefully choosing c_1 and β such that the following holds:

$$c_2 \cdot (\beta + \texttt{out}) + c_1 \cdot \texttt{out} = c_1 \cdot (\beta - 1)$$

  Specifically, for some value of β, c_1 must be $\frac{c_2 \cdot (\beta + \texttt{out})}{\beta - \texttt{out} - 1}$. This is well defined as long as $\beta > \texttt{out} + 1$ holds.

- *Case 2: No values are copied:* Copying an address causes the degree of the value held at the pointee to increase by 1, leading to an increase in potential of at most c_1. In addition to this increase in potential, by inspection of Figure 2, the cost incurred is $O(1)$.

By inspection of Figure 2, $\mathcal{C}_{IM}$ runs in $O(1)$ local space. Note that recursive calls to `copy` can be implemented using a $O(1)$-PM program emulating a stack, thereby storing program state in PM machine memory. □

The following holds as an immediate consequence of Theorems 1 to 3:

Corollary 1 (Oblivious IM)**.** *There exists interactive RAM programs* `Path-OIM` *and* `Circuit-OIM` *such that for every IM program $\mathcal{P}$ running in time n and for statistical security parameter λ, the following hold:*

- ***Correctness.*** *The following are statistically close in λ:*

$$\mathcal{P} \leftrightarrow \texttt{IM} \approx \mathcal{P} \leftrightarrow \texttt{Path-OIM} \leftrightarrow \texttt{RAM}$$
$$\mathcal{P} \leftrightarrow \texttt{IM} \approx \mathcal{P} \leftrightarrow \texttt{Circuit-OIM} \leftrightarrow \texttt{RAM}$$

- ***Space complexity.*** `Path-OIM` *runs in $O(\lambda)$ local space and* `Circuit-OIM` *runs in $O(1)$ local space.*
- ***Communication complexity.***
  - *The interaction between compiled program $\mathcal{P} \leftrightarrow$* `Path-OIM` *and random access memory* `RAM` *involves $O(n)$ roundtrips and $O(n \lg n)$ words of communication.*
  - *The interaction between compiled program $\mathcal{P} \leftrightarrow$* `Circuit-OIM` *and random access memory* `RAM` *involves $O(n\lambda)$ roundtrips and $O(n\lambda)$ words of communication.*
- ***Obliviousness.*** *$\mathcal{P} \leftrightarrow$* `Path-OIM` *and $\mathcal{P} \leftrightarrow$* `Circuit-OIM` *are oblivious RAM programs.*

4 Mixing Mutable and Immutable Memory

A mixed memory (MM) program uses both mutable and immutable memory cells. A memory address is initialized either by allocating a mutable address, or by copying an existing address. The operations that can be performed on an address depends on whether that address is mutable or immutable. While all addresses can be read, only mutable addresses can be written to. The `freeze` operation changes a mutable address into an immutable one, and it also recursively freezes all memory reachable from the pointee. Operations on immutable addresses are more efficient than operations on a mutable ones by a $\log d$ factor, where d is the degree of the pointee.

Definition 5 (Mixed Memory Program)**.** *A **mixed memory program** $\mathcal{P}$ is an opaque address program whose operations on addresses are restricted to the following instructions. An address can be initialized using the following:*

$p :=$ `alloc`	*allocate a fresh address that is marked as mutable*
$q, r :=$ `copy` p	*return fresh addresses q and r that each point to p's pointee.*
	q and r are mutable if p is mutable and are immutable otherwise.

The following instructions can be performed on an initialized address:

$p :=$ `write`$(x_1, ..., x_k)$	*write a tuple to memory as the pointee of a mutable address p.*
$(x_1, ..., x_k) :=$ `read` p	*load p's pointee into variables $(x_1, ..., x_k)$.*
`freeze` p	*Given mutable address p that points to v, mark all addresses that point to v as immutable.*
	Recursively freeze mutable addresses stored in v.

We say that an address becomes **invalidated** *if it is used as input to a* `copy`*,* `write` *or* `read` *instruction. $\mathcal{P}$ is **valid** if it never passes an invalid address as an argument to* `read`*/*`write`*/*`copy`*. From here on, we only consider/define valid mixed memory programs. A **mixed memory machine** `MM` is the interactive algorithm that handles the above memory requests in the natural manner.*

4.1 Building an Oblivious Mixed Memory Compiler

We present a compiler $\mathcal{C}_{MM}$ that compiles requests made to MM into requests to a $O(1)$-PM program. [AHR24] presents a compiler that compiles a PM program into a SAM program. Their implementation can easily be seen as a compiler $\mathcal{C}_{PM}$ that compiles a PM program into a $O(1)$-PM program; we will briefly explain how this is done shortly. $\mathcal{C}_{MM}$ can be implemented simply by invoking $\mathcal{C}_{PM}$ for operations on mutable addresses and $\mathcal{C}_{IM}$ for operations on immutable addresses. The `freeze` instruction that changes a mutable address into an immutable one is unique to MM, and we discuss its implementation in detail. Our implementation of `freeze` restructures values in memory from the format required for $\mathcal{C}_{PM}$ into the format required for $\mathcal{C}_{IM}$.

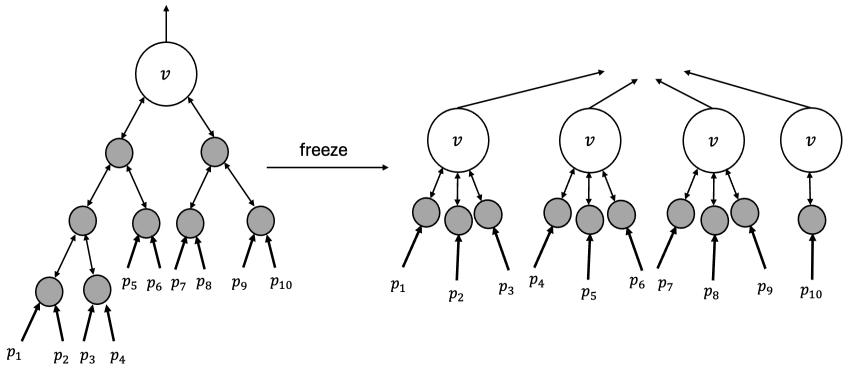


Figure 3: `freeze` rearranges addresses from the format required for PM into the format required for IM. An $\leftrightarrow$ arrow between two nodes indicates that both nodes have addresses pointing to each other. Here, we assume that $\beta = 3$.

Implementing PM as a $O(1)$-PM program. The implementation of `freeze` is based on how [AHR24] implements $\mathcal{C}_{PM}$. An address points to a pointee by means of a balanced binary tree with the pointee at its root (see Figure 3). Specifically, an address points to a leaf of this tree. Each node in this tree has addresses pointing to its children and its parent. Given an address, the program can read/write values by recursively climbing the tree until it reaches the root. Note that the cost to read/write a pointer thus scales with that pointer's distance from the root of the tree. Thus, [AHR24] must ensure that this distance does not grow too large; see next.

Copying an address is slightly more involved than reading/writing since, to keep costs low, we must maintain that the tree remains balanced. This balancing is achieved by ensuring that the tree is *complete*. An address is copied by first fetching the root, then traversing the path to the rightmost leaf in the last level of the tree by reading addresses to the appropriate child. The path to be traversed can be calculated using the degree of the pointee, which is stored at the root. The copy of the address is then created at this rightmost leaf.

Implementing `freeze`. Calling `freeze` on a mutable address rearranges the leaves of the above binary tree into intermediate addresses that point to various copies of the pointee, i.e the structure required for $\mathcal{C}_{IM}$. This is done by performing a traversal of the tree starting at the root, creating the required number of copies based on the degree of the pointee, and having the leaves of the tree store intermediate addresses that point to these copies. After this rearrangement is complete, mutable addresses held in the pointee are *recursively* frozen. We present an $O(1)$-PM program that implements $\mathcal{C}_{MM}$ in Figure 3.

The efficiency of `freeze`. Bounding the cost of `freeze` is non-trivial because `freeze` creates copies of values, and creating copies may cause the degrees of other values (in terms of $O(1)$-PM addresses) to increase, even if these values are frozen recursively. We first bound the cost of copying a value.

Claim 1. *Each of* `freeze`*'s calls to* `copy` *incurs only amortized $O(1)$ constant-degree PM operations.*

Proof. A value may contain both mutable and immutable addresses. From Theorem 3, copying an immutable address incurs $O(1)$-PM operations. A copy of a mutable address p during a `freeze` operation is performed using $O(1)$ PM operations by simply extending the path to the leaf that p points to. Specifically, a copy q of p that points to a leaf node n is performed by creating a new node n' with an address that points to n, and having p and q point to n'. □

Note that we have therefore modified the behavior of how a mutable pointer p is copied, but only for those copies created during `freeze`. This modification will not ensure that the tree of pointers to p's pointee is balanced, and above we explained that the tree must be balanced to keep the root-to-leaf distance small. However, it will not matter that trees may become unbalanced during `freeze`, since p will ultimately be frozen as well. As we will show next, during `freeze` the *arrangement* of these PM pointer trees is irrelevant to cost, and only their *size* matters:

Claim 2. *Let D be the total degree of all values in MM machine memory (including copies) in terms of $O(1)$-PM addresses. We say that a value v is frozen if all addresses pointing to it are immutable. Suppose that a call to* `freeze`

```
def MM.read p → (x_1, ..., x_k) :
    if p.mutable then return C_PM.read p
    else return C_IM.read p

def MM.write p (x_1, ..., x_k) :
    C_PM.write p (x_1, ..., x_k)

def MM.copy p → q :
    if p.mutable then return C_PM.copy p
    else return C_IM.copy p

def traverse p leaves → [ ] :
    v = read p
    if ¬v.left and ¬v.right then
        v.frozen = true
        write p v
        leaves.add(p)
    if v.left then
        leaves := traverse v.left leaves
    if v.right then
        leaves := traverse v.right leaves
    return leaves
```

```
def MM.freeze p :
    v := read p
    if v.frozen then
        return
    q := v.parent
    while ¬v = root do
        q := v.parent
        v := read q
    leaves := traverse(q, [ ])
    n := v.degree / 2β
    while n do
        v' := copy(v)  // copies a mutable address by
          extending the path to the leaf
        cnt = min(2β, |leaves|)
        n  = n − cnt
        ptrs := leaves.remove(cnt)
        divert(ptrs, v')
    for each p in v do
        freeze(p)
```

Figure 4: $O(1)$-PM program implementing $\mathcal{C}_{MM}$. The procedure `traverse` recursively traverses the binary tree used to implement $\mathcal{C}_{PM}$ and returns a list of pointers to the leaves of this tree. The `freeze` procedure uses this list to make intermediate addresses point to copies of the pointee. The implementation of `divert` follows from Figure 2.

causes values $v_1, \ldots, v_\ell$ with degrees $d_1, d_2, \ldots, d_\ell$ (in terms of mutable MM addresses) to be recursively frozen. If $\beta >$ `out`, the total increase in D is $O(d_1 + d_2 + \cdots + d_\ell)$.

Proof. Let c_i denote v_i's contribution to D. Note that v_i holds at most `out` addresses. We say that the pointees of these addresses are distance-1 from v_i. The degree of each of these distance-1 addresses will increase by factor $\lceil d_i/\beta \rceil$ as a result of the `freeze` operation, since we create $\lceil d_i/\beta \rceil$ total copies of v_i. However, because we also recursively freeze these distance-1 addresses, v_i indirectly increases the degree of those values at distance 2. In particular, v_i contributes at most $\lceil d_i/\beta^2 \rceil$ to the degree of each value at distance 2. More generally, each frozen value v_i contributes $\lceil d_i/\beta^k \rceil$ to each value at distance k. Since there are at most $\texttt{out}^k$ values at distance k, we can bound c_i as follows:

$$
\begin{aligned}
c_i &= O(\texttt{out} \cdot \frac{d_i}{\beta} + \texttt{out}^2 \cdot \frac{d_i}{\beta^2} + \texttt{out}^3 \cdot \frac{d_i}{\beta^3} + \ldots) \\
&= O\left(d_i \cdot \left(\left(\frac{\texttt{out}}{\beta}\right) + \left(\frac{\texttt{out}}{\beta}\right)^2 + \left(\frac{\texttt{out}}{\beta}\right)^3 + \ldots\right)\right) \\
&= O(d_i) && (\because \beta > \texttt{out})
\end{aligned}
$$

□

Theorem 4 (Compiling MM to $O(1)$-PM)**.** *There exists an interactive, $O(1)$-PM program $\mathcal{C}_{MM}$ that implements the semantics of a mixed memory. In particular, for every MM program $\mathcal{P}$ that makes $n_{\texttt{mut}}$ queries to mutable addresses and $n_{\texttt{im}}$ queries to immutable addresses, the following hold:*

- ***Correctness.*** *The following are identically distributed:*

$$\mathcal{P} \leftrightarrow \texttt{MM} \equiv \mathcal{P} \leftrightarrow \mathcal{C}_{MM} \leftrightarrow \texttt{PM}$$

- ***Space complexity.*** *$\mathcal{C}_{MM}$ runs in $O(1)$ local space.*

- ***Runtime complexity.*** *Let* $\tilde{n}_{\texttt{mut}} = O\left(n \cdot \sum_{i=0}^{n_{\texttt{mut}}} \lg d_i\right)$*, where* d_i *denotes the degree of the value read during the* i*th operation on a mutable address. The compiled program* $\mathcal{P} \leftrightarrow \mathcal{C}_{MM}$ *runs in time* $O(\tilde{n}_{\texttt{mut}} + n_{\texttt{im}})$*.*

Proof. Since `freeze` restructures memory into the format required for $\mathcal{C}_{IM}$, the correctness of $\mathcal{C}_{MM}$ follows from that of $\mathcal{C}_{PM}$ and $\mathcal{C}_{IM}$. We show that `freeze` is performed using $O(1)$ PM operations; the runtime of $\mathcal{C}_{MM}$ then follows from the runtime of $\mathcal{C}_{PM}$ and $\mathcal{C}_{IM}$.

Roughly, our proof shows that we can pay for the cost of freezing a mutable address at the time that mutable address is created. In an amortized sense, calls to `freeze` become free. Specifically, we use the potential method to bound the cost of `freeze`, with the potential function being $c \cdot m$, where m is the total number of mutable addresses in memory and c is a constant to be determined later.

Suppose that a call to `freeze` causes values $v_1, \ldots, v_\ell$ with degrees $d_1, \ldots, d_\ell$ to be frozen recursively. Let $d = d_1 + \cdots + d_\ell$. From Claims 1 and 2, and by inspection of Figure 4, $O(d) = c' \cdot d$ operations are incurred. The amortized cost of `freeze` is $c' \cdot d - c \cdot d$, since there is a decrease in potential of c for every address that points to a value that gets frozen. By setting $c = c'$, this cost is 0. Note that copying a mutable address causes an increase in potential of c. However, since this is a constant, the asymptotic cost of this operation does not change.

Note that `freeze` requires storing a list `leaves`. If this list is stored in pointer machine memory, and if recursive calls to `freeze` are implemented using a stack, by inspection of Figure 4, `freeze` requires $O(1)$ local space. Specifically, `leaves` can be implemented as a $O(1)$-PM program that emulates a linked list that writes values to and reads values from PM memory. □

The following holds as an immediate consequence of Theorems 1, 2 and 4.

Corollary 2 (Oblivious MM)**.** *There exists interactive RAM programs* `Path-OMM` *and* `Circuit-OMM` *such that for every MM program* $\mathcal{P}$ *running that makes* $n_{\texttt{mut}}$ *queries to mutable addresses and* $n_{\texttt{im}}$ *queries to immutable addresses and for statistical security parameter* λ*, the following hold:*

- ***Correctness.*** *The following are statistically close in* λ*:*

$$\mathcal{P} \leftrightarrow \texttt{MM} \approx \mathcal{P} \leftrightarrow \texttt{Path-OIM} \leftrightarrow \texttt{RAM}$$
$$\mathcal{P} \leftrightarrow \texttt{MM} \approx \mathcal{P} \leftrightarrow \texttt{Circuit-OIM} \leftrightarrow \texttt{RAM}$$

- ***Space complexity.*** `Path-OMM` *runs in* $O(\lambda)$ *local space and* `Circuit-OMM` *runs in* $O(1)$ *local space.*
- ***Communication complexity.*** *Let* $\tilde{n}_{\texttt{mut}} = O\left(n \cdot \sum_{i=0}^{n_{\texttt{mut}}} \lg d_i\right)$*, where* d_i *denotes the degree of the value read during the* i*th operation on a mutable address, and let* $n = n_{\texttt{mut}} + n_{\texttt{im}}$*.*
  - *The interaction between compiled program* $\mathcal{P} \leftrightarrow$ `Path-OMM` *and random access memory* `RAM` *involves* $O(\tilde{n}_{\texttt{mut}} + n_{\texttt{im}})$ *roundtrips and* $O((\tilde{n}_{\texttt{mut}} + n_{\texttt{im}}) \lg n)$ *words of communication.*
  - *The interaction between compiled program* $\mathcal{P} \leftrightarrow$ `Circuit-OIM` *and random access memory* `RAM` *involves* $O((\tilde{n} + n_{\texttt{im}})\lambda)$ *roundtrips and* $O((\tilde{n} + n_{\texttt{im}})\lambda)$ *words of communication.*
- ***Obliviousness.*** $\mathcal{P} \leftrightarrow$ `Path-OIM` *and* $\mathcal{P} \leftrightarrow$ `Circuit-OIM` *are oblivious RAM programs.*

5 Applications

Here, we state examples of existing algorithms that can naturally be written as immutable memory programs and mixed memory programs without being tweaked. Using our compilers, we obtain oblivious RAM programs for these algorithms with new asymptotic results; see Table 1 for a comparison with ORAM / [AHR24]. We compare our results with other specific constructions in §6.

Oblivious Evaluation of Lambda Expressions. Evaluation of arbitrary lambda calculus expressions can be easily achieved as an immutable memory program. While this is likely straightforward for those highly familiar with the lambda calculus, we present why this is so in Appendix B by arguing that a standard CEK machine is an immutable memory program.

Fact 1 (Reducing Lambda Calculus to Immutable Memory)**.** *Let* e *denote an arbitrary call-by-value lambda expression that evaluates to a value* ν *within* n *steps. There exists an* ***immutable memory program*** $\mathcal{P}$ *with* $O(1)$ *local space that computes from (a representation of)* e *(a representation of)* ν *within* $O(n)$ *time.*

The following holds as an immediate consequence of Fact 1 and corollary 1:

Corollary 3 (Oblivious Lambda Calculus)**.** *There exists oblivious RAM programs evaluating arbitrary call-by-value lambda calculus expressions with the following costs, where n is the runtime of the expression.*

- $O(n \lg n)$ *words of communication,* $O(n)$ *roundtrips and* $O(\lambda)$ *local space.*
- $O(n\lambda)$ *words of communication,* $O(n\lambda)$ *roundtrips and* $O(1)$ *local space.*

Oblivious Suffix Tree Construction. Ukkonen's algorithm [Ukk95] constructs from some string the *suffix tree* for that string. This algorithm can be interpreted as a mixed memory program, and hence it can be efficiently compiled to an oblivious program. Specifically, the suffix tree can be implemented using mutable addresses, with nodes of the tree holding immutable addresses that point to letters in the string. We note that while the degrees of values storing letters of the string can be arbitrary, Ukkonen's algorithm ensures that the degree of each node in the suffix tree is $O(1)$.

In a bit more detail, Ukkonen's algorithm works by maintaining a tree of constant-out-degree nodes. By inspection it can be seen that Ukkonen's algorithm is clearly a pointer-machine program, so all that needs to be considered is the in-degree of the nodes and their mutability. Characters of the target string can have arbitrary degree, but this string is fixed, and hence we can mark it as immutable. The nodes of the suffix tree itself must be mutable while running the algorithm, but they are clearly constant-degree. In particular, each tree node's address (1) is the child of one parent node, (2) is the target of at most one so-called "suffix link", and (3) might be currently held as an in client memory. Thus, the maximum in-degree of each tree node is at most three. In sum, all memory addresses in Ukkonen's algorithm can either be marked as immutable, or have constant in-degree. Hence, the algorithm can be compiled to a *constant-degree* pointer machine program.

The above discussion, combined with Corollary 2, implies the following:

Theorem 5 (Oblivious Suffix Trees)**.** *Let Σ denote a constant-sized alphabet. There exists an MM program that implements Ukkonen's algorithm for length-n strings over Σ in time $O(n)$. Accordingly, there exist oblivious RAM programs that implement Ukkonen's algorithm with the following costs.*

- $O(n \lg n)$ *words of communication,* $O(n)$ *roundtrips and* $O(\lambda)$ *local space.*
- $O(n\lambda)$ *words of communication,* $O(n\lambda)$ *roundtrips and* $O(1)$ *local space.*

Deterministic Finite Automaton. A DFA that accepts strings of a certain language can be constructed using a PM program. Once constructed, the states of a DFA remain immutable. Thus, it is possible to implement a mixed memory program that constructs a DFA and freezes it so that it can be efficiently used to accept / reject a string.

Theorem 6 (Oblivious Deterministic Finite Automaton)**.** *There exists an MM program that constructs a DFA $(\mathcal{Q}, \Sigma, \delta, q_0, F)$ using $\texttt{poly}(\mathcal{Q})$ operations on mutable addresses and checks whether a string s over Σ of length n is accepted using $O(n)$ operations on immutable addresses. Accordingly, there exist oblivious RAM programs that construct the DFA using $\texttt{poly}(\mathcal{Q})$ words of communication and roundtrips, and check whether s is accepted with the following costs:*

- $O(n \lg \mathcal{Q})$ *words of communication,* $O(n)$ *roundtrips and* $O(\lambda)$ *local space.*
- $O(n\lambda)$ *words of communication,* $O(n\lambda)$ *roundtrips and* $O(1)$ *local space.*

Applications for MPC. It is well known that one can implement an MPC protocol for any constant-sized RAM program with a fixed (constant-sized) instruction set, even those that allow arbitrary control flow, see e.g. [WGMK16]. This is done by carefully emulating the components of a CPU, such as the program counter (PC), registers, ALU, and RAM. Care should be taken to hide which step of the program is being executed (the value of the PC), as this leaks control flow. The value of the PC for each step is computed by performing a *linear scan* of the program. The output of a step is then computed by executing *all possible instructions* in the instruction set and selecting the output of the correct instruction using the PC. To handle RAM, the approach works by implementing the ORAM client via an MPC circuit. Note here that the relevant ORAM cost metric is the *circuit complexity* of the ORAM compiler. By using `Circuit-ORAM`, which is optimized for this particular metric, one can implement a semi-honest-secure MPC protocol for any RAM program at cost $O(n \log^2 n\lambda)$ bits of communication.

Because our techniques can be compiled to a simplification of `Circuit-ORAM`, all of our results can be translated to the MPC setting. As an example of what can be achieved, we state the following immediate result for MPC evaluation of lambda calculus programs, which integrates our handling of immutable memory into an MPC protocol.

Corollary 4. *There exists a* 3*-party MPC protocol that tolerates* 1 *semi-honest corruption and evaluates an arbitrary lambda expression. If the expression evaluates in time* n*, then the protocol consumes* $O(n \log n \lambda)$ *total bits (or* $O(n\lambda)$ *words) of communication.*

The above result is achieved by evaluating (via a CEK machine; see Section B) a lambda expression via immutable memory, and implementing the immutable memory client by means of a Boolean circuit. This result is not achievable by any prior MPC protocol. Of course, this is a bare-bones result: it implements in a straightforward way a stripped down computational model in the simplest MPC setting. However, it does illustrate that mixed memory programs are particularly interesting for the MPC setting.

Mixed memory also has the potential to aid MPC in handling programs with complex control flow. In particular, suppose we parameterize cost by the number of program instructions S. By using our techniques, we can bring down the overead of updating the program counter from $O(S)$, due to performing a linear scan of the program, to $O(\log(S+n))$, where n is the runtime of the program to be compiled. This is done by constructing and freezing an inter-procedural control flow graph (CFG) with each node being an instruction, and an edge between two nodes indicating possible control flow. Similar to Theorem 6, evaluating a program can be seen as a walk of the CFG using $O(n)$ operations on immutable addresses.

6 Other Related Work

We reviewed most of the relevant related work in Section 1. Here, we cover works related to the applications allowed by our new obliviousness compilers.

Secure Pattern Matching. Secure pattern matching allows a user to find all occurrences of a short string s (of size m) in a long text T (of size n) stored at a server while keeping both s and T hidden. Prior works that study this problem all require computation and communication costs of $O(m+n)$ for each string that is queried [HL08, HT10, BEDM$^+$13, YSK$^+$13, KRT18, ZML20]. Our oblivious suffix tree can help reduce these costs to $O((m+z)\log n)$ per string, where z is the number of occurrences of the queried string.

Substring Searchable Encryption. Secure pattern matching is also useful for designing substring searchable encryption schemes [CS15b] that hide access pattern leakage. The work of [MB15] presents a scheme that improves over using oblivious data structures [WNL$^+$14] by a $\log n$ factor, however, only when $m = \Omega(\log n)$. In contrast, our suffix tree algorithm improves over prior work by a log factor for any set of parameters. Other substring searchable encryption schemes with access pattern privacy require a server computation of $\Omega((m+o)\cdot n)$ [SNR16, IIY17, MBP21] or rely on trusted hardware [MSBP20]. We only require a server computation of $O((m+o)\cdot \log n)$.

Oblivious DFA Evaluation. Oblivious DFA evaluation is mainly studied in the context of MPC, where a client that holds a private input string of size n checks if the string is accepted by a private DFA held at a server. Prior works solve this problem with $\Omega(n \cdot \mathcal{Q})$ communication cost [Ker06, GHS10, MNSS12]: they rely on the server sending a permutation of the states of the DFA to the client, requiring a communication of $O(\mathcal{Q})$ for every string to be evaluated. While we study oblivious DFA evaluation in a different setting, our work can be extended to the MPC setting while incurring much less communication.

Lambda Calculus for MPC. While prior works propose extensions of the lambda calculus as languages using which programs for MPC can be written [DSLH19, SDH$^+$21], no prior work considers using lambda calculus as the model of computation for MPC. Roughly, prior work used lambda calculus to describe protocols; our result allows MPC protocols for programs in the lambda calculus.

7 Acknowledgments

We thank Ling Ren for several helpful discussions.

Appendices

A IM Compiler as a Constant-Degree PM Program

In §3, we presented $\mathcal{C}_{IM}$ that allowed operations on addresses without invalidating them. Figure 5 presents an implementation of $\mathcal{C}_{IM}$ that explicitly creates copies.

B Oblivious Lambda Calculus

The lambda calculus is a Turing-complete model of computation that, in its simplest form, is expressed and evaluated as follows.

Definition 6 ((Call by Value) Lambda Calculus)**.** *A **lambda expression** (or just **expression**) e is a string defined by the following context-free grammar:*

$$\begin{aligned} e ::= \lambda x.e & \qquad \textit{lambda abstraction} \\ \mid e\ e & \qquad \textit{function application} \\ \mid x & \qquad \textit{variable reference} \end{aligned}$$

*Here, x ranges over the set of program variable names. We say that for an expression $\lambda x.e$, variable x is **in scope** in the subexpression e. An expression is **well-formed** if all occurrences of variable references are in scope; from here on, we only consider well-formed expressions. We say that an expression e is a **value** if it is of form $\lambda x.e$. The goal of **evaluation** is to simplify an arbitrary expression e to a value. We for simplicity consider **call by value** evaluation semantics, which state that to evaluate a function application $e_0\ e_1$, first recursively evaluate both e_0 and e_1, then perform the following rewrite:*

$$(\lambda x.e)\ \nu \longrightarrow e[\nu/x] \qquad \textit{where subexpression } \nu \textit{ is a value}$$

Here $e[\nu/x]$ denotes a (capture-avoiding) substitution: occurrences of x in e are replaced by value ν.

There are some difficulties in assigning a precise cost to a given lambda expression, since substitution is a powerful operation that performs arbitrary numbers of replacements in one operation. We therefore take a direct approach by recalling a classic machine that evaluates lambda expressions, and simply defining cost as the time it takes that machine to run. Note that this is within constant factors of other cost models for the lambda calculus, e.g. the one defined by [BG95]:

Definition 7 (CEK Machine)**.** *A CEK machine is a state transition system defined on triples (e, E, K) where e is an expression tree called the* control*, E is a dictionary called the* environment*, and K is a stack called the* continuation*. More precisely, the environment E is defined recursively, and it maps program variable names to expression/environment pairs. The machine transitions by performing simple pattern matching/updating of its three components:*

$$\begin{aligned} x, E, K &\longrightarrow e', E', K && \textit{where } E[x] = (e', E') \\ (e_0, e_1), E, K &\longrightarrow e_0, E, \texttt{push}((\texttt{arg}, e_1, E), K) && \\ \lambda x.e_0, E, K &\longrightarrow e_1, E', \texttt{push}((\texttt{app}, (\lambda x.e_0), E), K') && \textit{where } (\texttt{arg}, e_1, E', K') = \texttt{pop}(K) \\ \lambda x.e_0, E, K &\longrightarrow e_1, E'[y \leftarrow \lambda x.e_0], K' && \textit{where } (\texttt{app}, \lambda y.e_1, E', K') = \texttt{pop}(K) \end{aligned}$$

Definition 8 (Runtime of Lambda Expression)**.** *Let e be a well-formed lambda expression. Consider the following CEK state:*

$$e, \texttt{empty-dictionary}, \texttt{empty-stack}$$

*We define the **runtime of** e as the number of state transitions taken by the CEK machine before the above initial state reaches a terminal state (one where no transitions can be applied).*

Fact 1 (Reducing Lambda Calculus to Immutable Memory)**.** *Let e denote an arbitrary lambda expression with runtime n. There exists an **immutable memory program** $\mathcal{P}$ with $O(1)$ local space that computes from (a representation of) e (a representation of) ν within $O(n)$ time.*

```
def divert(ptrs, v) :
    p_int := alloc()
    for each p in ptrs do
        PM.write(p, p_int)
    PM.write(p_int, (v, ptrs))

def IM.copy(p) → q :
    p_int := PM.read(p)
    v, ptrs := PM.read(p_int)
    q_int := PM.copy(p_int)
    q := PM.alloc()
    PM.write(q, q_int)
    ptrs.add(q)
    if |ptrs| = 2β then
        v' := IM.copy(v)  // recursively copies
          addresses stored in v
        ptrs' := (⊥, ..., ⊥)  // Initialize a 2β-tuple
          filled with ⊥
        for each p in half(ptrs) do
            ptrs.remove(p)
            ptrs'.add(p)
        divert(ptrs', v')
    PM.write(p, (v, ptrs))
    return q

def IM.write(x_1, ..., x_k) → p :
    ptrs := (⊥, ..., ⊥)  // Initialize a 2β-tuple filled
      with ⊥
    p_int := PM.alloc()  // Intermediate address that
      points to the value
    ptrs.add(p_int)
    p := PM.alloc()  // address that points to the
      intermediate address
    PM.write(p, p_int)
    PM.write(p_int, ((x_1, ..., x_k), ptrs))
    return p

def IM.read(p) → (x_1, ..., x_k) :
    p_int := PM.read(p)
    (x_1, ..., x_k), ptrs := PM.read(p_int)
    return (x_1, ..., x_k)
```

```
def divert(v, ptrs, v') :
    p'_int := PM.alloc()
    for each p in ptrs' do
        p, p' := PM.copy(p)
        p'_int, p"_int := PM.copy(p'_int)
        PM.write(p', p"_int)
    PM.write(p'_int, (v', ptrs'))

def IM.copy(p) → q, r :
    p, p' := PM.copy(p)
    p_int := PM.read(p')
    p_int, p'_int := PM.copy(p_int)
    v ⊔ ptrs := PM.read(p'_int)
    p_int, p'_int := PM.copy(p_int)
    q_int := PM.copy(p'_int)
    q := PM.alloc()
    q_int, q'_int := PM.copy(q_int)
    PM.write(q, q'_int)
    q, q' = PM.copy(q)
    ptrs.add(q')
    if |ptrs| = 2β then
        v' := PM.copy(v)
        ptrs' := (⊥, ..., ⊥)
        for each p in half(ptrs) do
            ptrs.remove(p)
            ptrs'.add(p)
        divert(v, ptrs, v')
    p, r := PM.copy(p)
    PM.write(p, (v, ptrs))
    return q, r

def IM.write(x_1, ..., x_k) → p :
    ptrs := (⊥, ..., ⊥)
    p_int := PM.alloc()
    p_int, p'_int := PM.copy(p_int)
    ptrs.add(p'_int)
    p := PM.alloc()
    p, p' := PM.copy(p)
    p_int, p'_int := PM.copy(p_int)
    PM.write(p', p'_int)
    PM.write(p_int, ((x_1 ..., x_k), ptrs))
    return p
```

Figure 5: Implementation of IM that stores k-tuples using a $O(1)$-PM that stores $k + 2\beta$-tuples. A value is copied when its degree reaches 2β. The last 2β positions of a value in PM memory store addresses to intermediate addresses that point to the value. The procedure `divert` is used to make intermediate addresses point to a value's copy. The procedures defined in the left half assume that addresses can be repeatedly used; procedures on the right are equivalent valid PM programs that explicitly create copies. The implementation of `read` is agnostic to whether the repeated use of addresses is allowed.

Proof. By observing the CEK machine can be implemented as an immutable memory program.

Note that each CEK transition performs at most a constant number of operations on each of the three data structures, as well as performing simple constant-time rearranging of data. All three data structures can be achieved with immutable memory via standard techniques known in the functional programming literature.

One subtle point is that immutable-memory-based dictionaries incur worst case logarithmic overhead in the number of entries. However, in the CEK machine, each dictionary has at most a constant number of entries, bounded by the fact that the evaluated program has only a constant number of variables, since the program is of constant size. Thus, reading/updating the environment (a constant-sized dictionary) is achievable within $O(1)$ immutable memory operations. Thus, there exists an immutable memory program that evaluates expression e in $O(n)$ time. □

References

[AHR24] Ananya Appan, David Heath, and Ling Ren. Oblivious single access machines - A new model for oblivious computation. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 3080–3094. ACM Press, October 2024.

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: optimal oblivious ram. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30*, pages 403–432. Springer, 2020.

[BEDM+13] Joshua Baron, Karim El Defrawy, Kirill Minkovich, Rafail Ostrovsky, and Eric Tressler. 5pm: Secure pattern matching. *Journal of computer security*, 21(5):601–625, 2013.

[BG95] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 226–237, 1995.

[BKKO20] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed ORAM. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 215–232. Springer, Cham, September 2020.

[CDH20] David Cash, Andrew Drucker, and Alexander Hoover. A lower bound for one-round oblivious RAM. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part I*, volume 12550 of *LNCS*, pages 457–485. Springer, Cham, November 2020.

[CGPR15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 668–679, 2015.

[CS15a] Melissa Chase and Emily Shen. Substring-searchable symmetric encryption. *PoPETs*, 2015(2):263–281, April 2015.

[CS15b] Melissa Chase and Emily Shen. Substring-searchable symmetric encryption. *Proceedings on Privacy Enhancing Technologies*, 2015.

[DO20] Samuel Dittmer and Rafail Ostrovsky. Oblivious tight compaction in O(n) time with smaller constant. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 253–274. Springer, Cham, September 2020.

[DSLH19] David Darais, Ian Sweet, Chang Liu, and Michael Hicks. A language for probabilistically oblivious computation. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, 2019.

[DvDF+16] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 145–174. Springer, Berlin, Heidelberg, January 2016.

[FNO22] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. 3-party distributed ORAM from oblivious set membership. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22*, volume 13409 of *LNCS*, pages 437–461. Springer, Cham, September 2022.

[FNO24] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. MetaDORAM: Info-theoretic distributed ORAM with less communication. Cryptology ePrint Archive, Report 2024/011, 2024.

[GHS10] Rosario Gennaro, Carmit Hazay, and Jeffrey S Sorensen. Automata evaluation and text search protocols with simulation based security. *Cryptology ePrint Archive*, 2010.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524. ACM Press, October 2012.

[GM11] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *International Colloquium on Automata, Languages, and Programming*, pages 576–587. Springer, 2011.

[GMN+16] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1353–1364, 2016.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[Hic25] Michael Hicks, editor. *Proceedings of the ACM on Programming Languages*, volume 9. Association for Computing Machinery, 2025.

[HL08] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Theory of Cryptography Conference*, pages 155–175. Springer, 2008.

[HT10] Carmit Hazay and Tomas Toft. Computationally secure pattern matching in the presence of malicious adversaries. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*, pages 195–212. Springer, 2010.

[HV21] Ariel Hamlin and Mayank Varia. Two-server distributed ORAM with sublinear computation and constant rounds. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 499–527. Springer, Cham, May 2021.

[IIY17] Yu Ishimaki, Hiroki Imabayashi, and Hayato Yamana. Private substring search on homomorphically encrypted data. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6. IEEE, 2017.

[JLN19] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2439–2447. SIAM, 2019.

[Ker06] Florian Kerschbaum. Practical private regular expression matching. In *IFIP International Information Security Conference*, pages 461–470. Springer, 2006.

[KM19] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious RAM with small block size. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part I*, volume 11442 of *LNCS*, pages 3–33. Springer, Cham, April 2019.

[KRT18] Vladimir Kolesnikov, Mike Rosulek, and Ni Trieu. Swim: Secure wildcard pattern matching from ot extension. In *International conference on financial cryptography and data security*, pages 222–240. Springer, 2018.

[KU58] Andrei Nikolaevich Kolmogorov and Vladimir Andreevich Uspenskii. On the definition of an algorithm. *Uspekhi Matematicheskikh Nauk*, 13(4):3–28, 1958.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 523–542. Springer, Cham, August 2018.

[M+10] Simon Marlow et al. Haskell 2010 language report. 2010.

[MB15] Tarik Moataz and Erik-Oliver Blass. Oblivious substring search with updates. Cryptology ePrint Archive, Report 2015/722, 2015.

[MBP21] Nicholas Mainardi, Alessandro Barenghi, and Gerardo Pelosi. Privacy-aware character pattern matching over outsourced encrypted data. *Digital Threats: Research and Practice (DTRAP)*, 3(1):1–38, 2021.

[Mil97] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

[MNSS12] Payman Mohassel, Salman Niksefat, Saeed Sadeghian, and Babak Sadeghiyan. An efficient protocol for oblivious dfa evaluation and applications. In *Topics in Cryptology–CT-RSA 2012: The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27–March 2, 2012. Proceedings*, pages 398–415. Springer, 2012.

[MPC+18] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE symposium on security and privacy (SP)*, pages 279–296. IEEE, 2018.

[MSBP20] Nicholas Mainardi, Davide Sampietro, Alessandro Barenghi, and Gerardo Pelosi. Efficient oblivious substring search via architectural support. In *Proceedings of the 36th Annual Computer Security Applications Conference*, pages 526–541, 2020.

[Oka98] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th ACM STOC*, pages 294–303. ACM Press, May 1997.

[PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In Mikkel Thorup, editor, *59th FOCS*, pages 871–882. IEEE Computer Society Press, October 2018.

[RFK+15] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 415–430. USENIX Association, August 2015.

[Sch80] Arnold Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, 1980.

[SCSL11] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o ((log n) 3) worst-case cost. In *International Conference on The Theory and Application of Cryptology and Information Security*, pages 197–214. Springer, 2011.

[SDH+21] Ian Sweet, David Darais, David Heath, Ryan Estes, William Harris, and Michael Hicks. Symphony: A concise language model for mpc. In *Informal Proceedings of the Workshop on Foundations on Computer Secuirty (FCS)*, 2021.

[SGF17] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. *Cryptology ePrint Archive*, 2017.

[Sig] Signal. Technology deep dive: Building a faster oram layer for enclaves. `https://signal.org/blog/building-faster-oram/`.

[SNR16] Kana Shimizu, Koji Nuida, and Gunnar Rätsch. Efficient privacy-preserving string search and an application in genomics. *Bioinformatics*, 32(11):1652–1661, 2016.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.

[TJS19] Shruti Tople, Yaoqi Jia, and Prateek Saxena. {PRO-ORAM}: Practical {Read-Only} oblivious {RAM}. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 197–211, 2019.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[VHG23] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3907–3924. USENIX Association, August 2023.

[WCS15] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861, 2015.

[WGMK16] Xiao Wang, S Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of mips machine code. In *European Symposium on Research in Computer Security*, pages 99–117. Springer, 2016.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 215–226. ACM Press, November 2014.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[YSK+13] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Secure pattern matching using somewhat homomorphic encryption. In *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*, pages 65–76, 2013.

[ZE13] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *2013 IEEE Symposium on Security and Privacy*, pages 493–507. IEEE Computer Society Press, May 2013.

[ZKP16] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: the power of {File-Injection} attacks on searchable encryption. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 707–720, 2016.

[ZML20] Maryam Zarezadeh, Hamid Mala, and Behrouz Tork Ladani. Efficient secure pattern matching with malicious adversaries. *IEEE Transactions on Dependable and Secure Computing*, 19(2):1407–1419, 2020.