

On-the-Fly Memory Programming for Largely Unmodified Cryptographic Applications

Alice Yeh



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-130

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-130.html>

May 17, 2024

Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

On-the-Fly Memory Programming for Largely Unmodified Cryptographic Applications

by

Alice Yeh

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Raluca Ada Popa, Chair

Professor Natacha Crooks

Spring 2024

On-the-Fly Memory Programming for Largely Unmodified Cryptographic Applications

by Alice Yeh

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements for the
degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

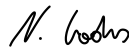


Professor Raluca Ada Popa
Research Advisor

May 13, 2024

(Date)

* * * * *



Professor Natacha Crooks
Second Reader

05/13/2024

(Date)

On-the-Fly Memory Programming for Largely Unmodified Cryptographic Applications

Copyright 2024

by

Alice Yeh

Abstract

On-the-Fly Memory Programming for Largely Unmodified Cryptographic Applications

by

Alice Yeh

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Raluca Ada Popa, Chair

Secure computation (SC) is a family of cryptographic primitives with the potential to enable transformative applications, such as secure sharing of patient data and fraud detection across financial institutions. Unfortunately, SC’s high memory overhead makes it difficult to realize such applications in practice. To address this overhead, recent work has proposed memory programming, a technique that leverages the determinism and obliviousness of SC’s memory access patterns to efficiently perform demand paging for SC. However, state-of-the-art memory programming requires application and library developers to rewrite their code into a special DSL or framework, so that the system has the necessary visibility into the program’s memory accesses.

We propose Osprey, a system that enables memory programming without requiring significant code changes. Osprey integrates with SC libraries with minimal code changes, and in many cases, requires no modifications at all to applications written against those libraries. We adapt three cryptographic frameworks to our system, all of which require code modifications that account for less than 1.1% of library code for integration. Evaluating our system on two distinct and data-intensive SC workloads shows reductions in page faults and context switches compared to classical OS paging. We additionally observe favorable timing performance for one workload and opportunities for optimizations from the other.

Contents

Contents	i
List of Figures	iii
List of Tables	iv
1 Introduction	1
2 Background	4
2.1 Secure Computation Building Blocks	5
2.2 Advances in Secure Computation	5
2.3 Memory Overhead of Secure Computation	6
3 Preliminary Experiments	7
3.1 Limitations in Existing Memory Programming Work	7
3.2 Improving Over Existing Work	8
3.3 Validating Hint-Passing Through Simulation	9
4 Design	12
4.1 Generating Page Fault Traces & Informed Paging	12
4.2 Content-Oblivious Memory Allocation	13
4.3 File I/O & Networking Overrides	15
4.4 Interfacing with External Libraries	15
4.5 Implementation	16
5 Adapting Cryptographic Frameworks	17
5.1 emp-toolkit	17
5.2 Microsoft SEAL	18
5.3 MP-SPDZ	18
5.4 Analysis of Adoption Overheads	19
6 Evaluation	20
6.1 Workloads	20

6.2	Benchmarks	21
6.3	Performance Analysis	24
7	Discussion	25
7.1	Related Work	25
7.2	Future Work	26
8	Conclusion	28
	Bibliography	29

List of Figures

2.1	Banks using secure computation to detect fraud without sharing their proprietary data.	4
3.1	Creating a memory management plan to simulate hint-passing.	8
3.2	Normalized execution performance of a system with unbounded physical memory, MAGE, our hint-passing simulation, and classical OS paging across 10 workloads with a 1 GiB physical memory limit.	9
3.3	Breakdown of execution performance by time spent in user space, kernel space, and blocked. Bars from left to right represent a system with unbounded physical memory (no pattern), MAGE (cross pattern), our hint-passing simulation (circle pattern), and classical OS paging (diagonal line pattern) in that order.	10
4.1	Slab allocator in action.	14
4.2	Annotating a <code>Ciphertext</code> class.	16
6.1	Major and minor page faults incurred. The bar on the left (no pattern) represents a system with unbounded memory, the middle bar (circle pattern) represents Osprey’s programmed pass, and the bar on the right (diagonal line pattern) represents a system using classical OS paging. The y-axis is logarithmic.	21
6.2	Voluntary and involuntary context switches during program execution. The bar on the left (no pattern) represents a system with unbounded memory, the middle bar (circle pattern) represents Osprey’s programmed pass, and the bar on the right (diagonal line pattern) represents a system using classical OS paging.	22
6.3	Normalized execution performance for merge and rstats. The bar on the left (blue) represents a system with unbounded memory, the middle bar (dark blue) represents Osprey’s programmed pass, and the bar on the right (white with diagonal line pattern) represents a system using classical OS paging.	23
6.4	Breakdown of execution performance for merge and rstats by time spent in user space, kernel space, and blocked. The bar on the left (no pattern) represents a system with unbounded memory, the middle bar (circle pattern) represents Osprey’s programmed pass, and the bar on the right (diagonal line pattern) represents a system using classical OS paging.	23

List of Tables

- 5.1 Complexity of integrating our system into external crypto libraries. The number of lines for Microsoft SEAL is in the worst case and can be further optimized. MP-SPDZ modifications are for the Shamir secret sharing protocol only and do not include networking overrides, which we leave for future work. 19

Acknowledgments

I would like to thank my advisor Professor Raluca Ada Popa for her guidance throughout my time at Berkeley and for introducing me to the world of security research. I would also like to thank Sam Kumar for his mentorship throughout my research journey—I have truly learned so much about the research process from him. I am also grateful for the students, faculty, and staff in Sky Lab that have fostered an incredibly supportive research community—I am endlessly inspired by so many of these people. Finally, I would like to thank my family for their love and support, as I would not be where I am today without them.

Chapter 1

Introduction

With the rapid development and adoption of digital technologies, more and more data is being created—by 2025, an estimated 463 *exabytes* of data will be created daily [8]. This data is invaluable to corporations for driving business analytics and generating intelligent insights; however, many organizations face the challenge of having to collect and compute on data in a privacy-preserving way, thanks in part to privacy regulations, like the GDPR in the EU and HIPAA for healthcare in the US, as well as a reluctance to share data to maintain a competitive edge.

Secure computation has emerged as a powerful cryptographic primitive that enables computation over encrypted data, where given n parties with individual inputs x_i , a joint function $f(x_1, \dots, x_n)$ can be computed over the inputs without revealing anything other than the function output. A popular use case of SC is secure multi-party computation, which allows parties to collaboratively compute on their shared data without any party learning the private inputs of another. Another is homomorphic encryption, which enables mathematical operations, such as additions and multiplications, to be performed on encrypted data.

The use cases of secure computation are plentiful. In the context of the healthcare industry, which is tightly regulated by HIPAA, hospitals can use secure computation to collaboratively compute on patient data to produce more accurate diagnoses, without revealing sensitive patient information to each other. Similarly, in the financial industry, where companies may be hesitant to share proprietary business data, secure computation can enable collaborative analytics for fraud detection.

Secure computation schemes have enabled applications of shared computation on encrypted data across research and industry, including to secure digital assets [10] and to collect analytics information in a privacy-preserving way [1]. However, the high memory overhead associated with secure computation has been a continuous obstacle to its widespread adoption and application towards practical use cases in industry—a number of cryptographic systems have noted that available memory is often quickly exhausted and that subsequent reliance on OS paging makes computation slow and inefficient [23, 26].

Recent work have produced systems that can run memory-intensive SC computations at speeds similar to machines with unbounded physical memory [18]. They make the key

observation that memory access patterns of SC protocols can be computed ahead of time, due to the fact that SC protocols must have data dependent memory accesses—a property known as *obliviousness*. This enables memory management plans to be produced ahead of time that can be used to efficiently perform demand paging, in a technique known as *memory programming*. Despite the performance gains that these systems achieve, they face a large usability barrier, as they require significant domain expertise to adopt. Additionally, there is room for performance improvements, as these systems offload work to expensive planning phases.

To address these usability challenges, we design Osprey, a system that manages the memory overhead of secure computation with minimal code changes for integration with existing cryptographic frameworks and no application modifications for usage. We present a solution that runs a lightweight speculative pass that records a trace of the program’s page faults and a subsequent programmed execution pass that leverages the trace to learn about upcoming page accesses for informed paging. To enhance portability, we make only user space modifications for tracing and informed paging.

To design a lightweight speculative pass that is fast and memory efficient, we observe that we don’t require the speculative pass to produce the *correct* program output, so long as the memory access patterns are unaffected, allowing us to trade off correctness for speed and memory efficiency. We design a *content-oblivious* memory allocator to allocate memory for data that does not influence memory access patterns and map the resultant content-oblivious memory region to a small set of physical pages. In addition, we design file I/O and networking overrides for the speculative pass to ensure that speculation does not pollute the file system and to enable asynchronous speculation for each party in protocols that involve communication between parties.

To ensure usability and portability, we design Osprey to integrate with cryptographic frameworks with minimal code changes. We implement an integration layer through template specialization to enable developers to easily annotate objects in their libraries to be allocated obliviously. We additionally provide code stubs for networking overrides that preserve memory access patterns. We adapt three major cryptographic frameworks, namely emp-toolkit [27], Microsoft SEAL [24], and MP-SPDZ [15], to work with our system to determine the complexity of integration. We find that all three libraries required less than 1.1% of library modifications for integration and that most libraries required fewer than 100 lines of code changes, which accounts for less than 0.1% of the library.

We evaluate Osprey on two distinct, data-intensive SC workloads and compare the page faults incurred, context switches during program execution, and timing to a system with unbounded physical memory and one using classical OS paging. Our system shows a $52.3\times$ and $12.9\times$ reduction in major page faults for the two workloads, respectively, compared to classical OS paging, and a $6.4\times$ reduction in voluntary context switches for both workloads. We find that for timing, our system performs $1.67\times$ faster than OS for one workload but performs $1.35\times$ slower for the other workload, which we attribute to the need for further optimizations to our user space centered prefetching approach and informed paging.

We structure the paper as follows. We introduce the building blocks of our system in

Section 2 and motivate our design with simulation results in Section 3. We detail Osprey's design in Section 4 and present case studies applying our system to cryptographic frameworks in Section 5. We discuss evaluation results in Section 6. We talk about our system in the context of the research space and look towards future work in Section 7. Finally, we conclude in Section 8.

Chapter 2

Background

With secure computation, given n parties each with their individual inputs x_i , a joint function $f(x_1, \dots, x_n)$ is computed over the inputs, and each party learns nothing more than the output of the function. Figure 2.1 depicts a commonly cited use case of secure computation, which involves enabling financial institutions to pool their data to detect fraud, which can feature complex patterns that involve multiple institutions, without having any of their competitors learn the data that they have shared.

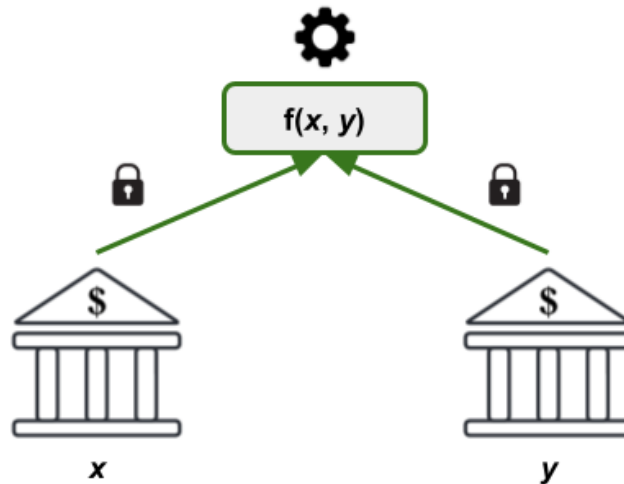


Figure 2.1: Banks using secure computation to detect fraud without sharing their proprietary data.

2.1 Secure Computation Building Blocks

Garbled Circuits

One common way that secure computation protocols are implemented is using Yao’s garbled circuits [28], where two parties, a garbler and an evaluator, jointly compute over a shared function f . Using a circuit compiler, a function f can be converted to a boolean circuit representation made up of AND and XOR gates. From here, the garbler will generate a garbled circuit, where for each gate, an obfuscated boolean gate truth table is generated. This garbled circuit is sent to the evaluator, who will evaluate the garbled circuit. To execute the circuit, the evaluator will need to get the appropriate input labels that correspond to each party’s private inputs. This can be achieved by having the two parties run an oblivious transfer protocol, which leaks no information about the private inputs. While the round complexity of garbling protocols is constant, the communication costs are high, as the amount of data that needs to be sent is quite large.

Obliviousness

Since secure computation protocols cannot leak any information about the data that is being computed on through their memory access patterns, many SC programs are *oblivious*—their memory accesses are data independent. This property of obliviousness means that the memory access patterns for multiple runs of the same program must be deterministic.

2.2 Advances in Secure Computation

The joint function f computed under a secure computation protocol cannot leak any information about its inputs, which means that the program’s memory access pattern must be independent of its input data. For garbled circuits, this can mean that for even simple functions, these circuit representations can be very large. For example, an RSA-1024 signature function contains more than 42 billion gates [17]. Since the first full implementations of secure computation protocols in the mid-2000s, advances in implementation techniques have contributed to 3-4 order of magnitude of improvement in performance [9].

Reducing Garbling Costs

One line of work focuses on tackling the high costs of garbled circuits, which comes from the bandwidth needed to transmit garbled gates and the computation required for evaluating garbled tables. To reduce the number of ciphertexts that need to be transmitted per gate, Naor et al. [22] proposed garbled row reduction (GRR), which observes that we can pick wire labels such that one of the ciphertexts is 0, enabling one less ciphertext to be transmitted for each gate. Kolesnikov and Schneider’s free XOR technique [16] enables XOR gates to be evaluated “for free,” without the need for garbled tables or expensive key operations, by

fixing the relationship between wire labels, which allows for new wire labels to be calculated by taking the XOR of the input labels. Zahur et al. [29] introduced a garbling technique that only requires two ciphertexts per AND gate and supports free XOR. They observe that an AND gate can be represented as an XOR of two half gates, which are AND gates where one of the inputs is known to one of the parties. With garbled row reduction, only one ciphertext needs to be transmitted for a half gate.

Circuit Optimizations

Another line of work looks to reducing the circuit size to reduce the execution costs of MPC protocols. For more widely used circuits, manual circuit design techniques have been used to discover opportunities to reduce circuit size. One example of this is Kolesnikov and Schneider’s [16] design for a conditional swapper, a basic component for an oblivious permutation, that used the free XOR technique to reduce the garbled table to two rows. Automated tools for producing efficient circuits have also been a major research focus, with techniques that focus on logic compaction [25] and minimization of non-free gates [11].

Systems Optimizations

Generating and storing the entire garbled circuit for a protocol requires large amounts of memory, so working to eliminate the concurrent memory overhead of secure computation has been a large research focus as well. Huang et al. [13] makes the insight that there is not a need for the garbler or the evaluator to be holding the entire circuit in memory and that generation and evaluation can be overlapped and pipelined. Gates are sent by the garbler as they are generated and evaluated by the evaluator as they and their inputs are received, then immediately discarded. In addition to pipelining, approaches to compress circuits by reusing circuit components for code blocks like loops [17] have also been a topic of research.

2.3 Memory Overhead of Secure Computation

Despite major advances in secure computation techniques that have reduced the order of growth of SC programs to plaintext factors, the constant factors of SC protocols still contribute to high memory overheads that cause practical deployments to become prohibitively slow once systems run out of memory. In garbled circuits, because wire values are necessarily encrypted, ciphertext expansion factors can cause a large blow-up in memory usage. In the case of circuits that use 128-bit block ciphers, each wire representing just 1 plaintext bit is 16 bytes, which is a 128x expansion factor [18]. In the case of other secure computation protocols like CKKS homomorphic encryption, ciphertexts can expand to orders of hundreds of times larger than plaintexts. In fact, for a short message, encrypting a single bit can produce a ciphertext that is a few megabytes in size [7]. The high memory demands of secure computation necessitate new approaches to help manage the memory overhead of SC.

Chapter 3

Preliminary Experiments

3.1 Limitations in Existing Memory Programming Work

Recent work towards reducing the memory overhead of secure computation have presented promising insights and results but suffer in terms of usability.

MAGE

One notable system is MAGE [18], which is able to run memory-intensive secure computations at speeds similar to machines with unbounded physical memory by taking advantage of the obliviousness property of secure computation schemes. Due to this observation that memory access patterns are not data dependent, the system is able to create memory management plans ahead of time and incorporate optimal access patterns, like Bélády’s algorithm, that are much more efficient than OS paging or heuristics.

Despite being able to perform at speeds that nearly match systems with unbounded memory, MAGE is not without its flaws. For one, MAGE’s planning phase can take a long time to run since it must go through the entire program to prescribe a memory management plan that can be used in the subsequent execution phase. In fact, in one of MAGE’s workloads, planning takes more time than the execution phase. Though the memory management plan produced from the planning phase can be reused with multiple runs of the execution phase, this can still prove inefficient when there are frequent changes to the original program or if we require only one (or few) executions of a single program, which would require a re-run of the planning phase. Additionally, the produced memory management plan can be very large as it tracks each instruction in the program. From a usability perspective, MAGE’s planning phase requires a DSL program input, which presents increased developer overhead, as developers must select a compatible DSL for the selected protocol and engine and write DSL code in a distributed way that explicitly indicates asynchronous network operations that are needed to transfer data between different workers. This requirement means that

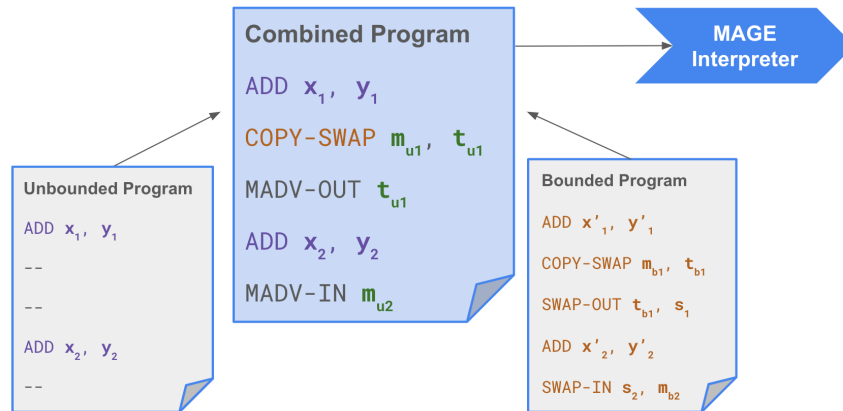


Figure 3.1: Creating a memory management plan to simulate hint-passing.

MAGE may be more difficult for non-experts to use, which makes widespread adoption of MAGE for secure computation schemes even more difficult, especially in industry. Further, the DSLs may not optimize the resulting circuit and may expose low-level SC operations.

3PO

3PO [2] builds on MAGE’s idea of taking advantage of obliviousness for prefetching memory, though in the context of far memory. 3PO introduces an in-kernel tracer to generate a tape of page accesses and accelerates tracing by recording accesses in batches of microsets that omit information about the exact sequence of accesses within the set. During execution, 3PO synchronizes the prefetcher with the application through selecting certain key pages that will page fault in the main execution and having the prefetcher resynchronize and bring in the next batch of pages when these faults are triggered. We draw several ideas for tracing and synchronizing hint-passing with the main run execution from 3PO but favor an approach that requires only user-space modifications.

3.2 Improving Over Existing Work

The pain points that we found in MAGE are (1) the slow planning phase that generates large memory management plans and (2) the DSL input. To address both problems, we propose a solution that takes the original program as input and runs a lightweight speculative pass that is fast and generates small traces, followed by an execution pass. We borrow from 3PO’s techniques of generating a trace of page accesses in the speculative pass and subsequently passing hints to the execution pass via the program trace about upcoming page accesses. However, this introduces a new pain point—the need for kernel modifications to generate the program trace and to prefetch pages, which greatly erodes usability. We solve this

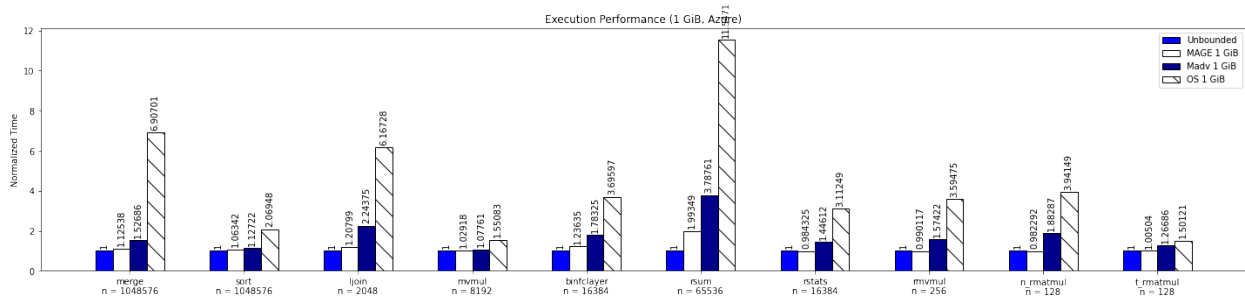


Figure 3.2: Normalized execution performance of a system with unbounded physical memory, MAGE, our hint-passing simulation, and classical OS paging across 10 workloads with a 1 GiB physical memory limit.

problem by recording and handling page faults in user space and passing hints the the OS about upcoming page accesses using syscalls. The remaining major unknown was whether using this type of user space approach to memory management would erode performance. This required us to validate that user space interfaces to memory management still perform comparably to the more direct paging approaches of prior systems.

3.3 Validating Hint-Passing Through Simulation

To validate that passing hints to the kernel about upcoming page accesses can still yield significant performance gains over OS paging, we simulate the execution performance by adapting MAGE. We treat MAGE’s planner as a black box and generate memory management plans from running the planner with unbounded memory and with 1 GiB of bounded memory. From these two plans, we generate a combined memory management plan that incorporates the instructions and addresses from the unbounded plan and the swap directives found in the bounded plan. We substitute the swap directives for the corresponding `madvise` syscall, which is used to give advice to the kernel about upcoming page accesses over an address range. This combined memory management plan is then fed into MAGE’s interpreter to retrieve timing and page fault information. Figure 3.1 shows this workflow on a simplified program.

Benchmarks were run on 10 workloads with varying access patterns using Microsoft Azure against baselines of a system with unbounded physical memory and a system using classical OS paging. We also compare our system with MAGE, which should still perform better as it uses more proactive swap instructions. 1 GiB `cgroups` were used to bound physical memory in all cases other than the unbounded physical memory case.

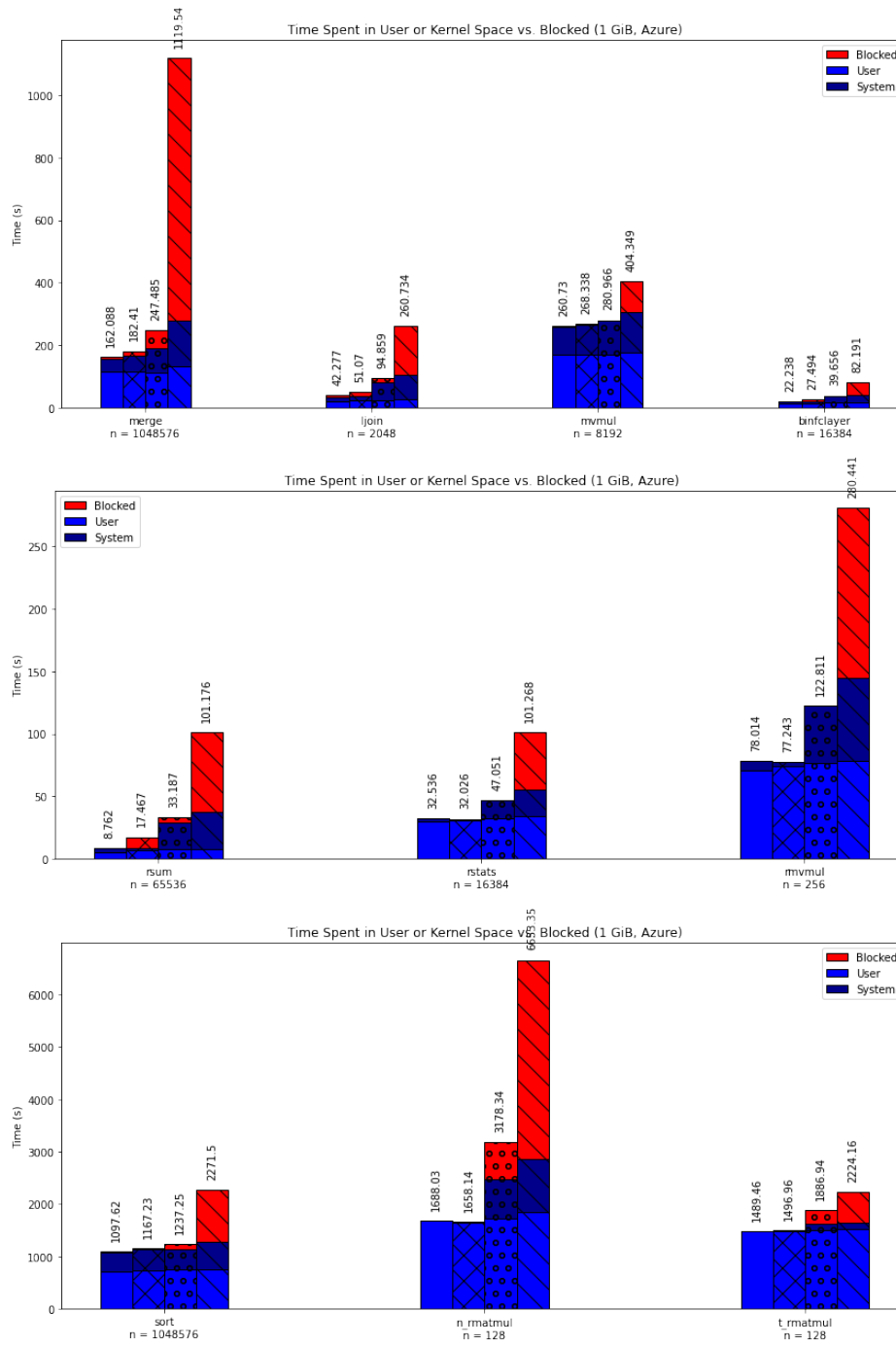


Figure 3.3: Breakdown of execution performance by time spent in user space, kernel space, and blocked. Bars from left to right represent a system with unbounded physical memory (no pattern), MAGE (cross pattern), our hint-passing simulation (circle pattern), and classical OS paging (diagonal line pattern) in that order.

Simulation Results

The results for running these workloads on a system with unbounded physical, with MAGE, with hint-passing with `madvise`, and with classical OS paging are shown in Figure 3.2. We find that the simulated system using `madvise` performs better than a system using classical OS paging for all workloads, outperforming the classical system by 2 - 4.5 \times in 7 of the workloads. Additionally, the system performs within 2 \times of MAGE for all workloads and performs within 1.5 \times of MAGE for 6 of the workloads. This shows that the OS is responsive to hint-passing and that using `madvise` calls for informed paging is a feasible alternative to injecting swap directives.

Figure 3.3 shows the breakdown of execution performance based on time spent in user space, kernel space, and blocked. Our simulated `madvise` system reduces the time spent blocked on page faults of the classical OS paging system, as we incur only a minor page fault instead of a major page fault. This is especially significant in workloads like `merge`, `rmvmul`, and `n_rmatmul`. The simulated system, however, does feature a slight increase in time spent in kernel space, compared to both MAGE and a system with unbounded physical memory, but this is expected due to the added syscalls.

From these simulation results, we find that a user space approach to memory management yields results that consistently outperform classical OS paging and perform comparably to direct paging approaches, validating the efficacy of a user space centered design.

Chapter 4

Design

The goal of Osprey is to reduce the memory overhead of a target program by speculatively generating page fault traces and informing the operating system about upcoming page accesses through hint-passing. On a high level, the system workflow consists of two separate passes over a target program, with a lightweight speculative pass that generates a trace of page accesses and a subsequent programmed pass that receives hints about upcoming page accesses and executes the program correctly.

The key design tenet for our system is usability. We achieve this with two overarching design considerations: using only user space modifications and ensuring minimal library changes for integration. We describe the individual systems components in more detail in the following sections. In particular, we detail how we can generate page fault traces and enable informed paging using only user-space modifications. We further detail how we can make the speculative process memory-efficient and fast through our design of a specialized oblivious memory allocator. Finally, we make additional design considerations required for ease of use and for integration with SC libraries.

4.1 Generating Page Fault Traces & Informed Paging

The purpose of the speculative pass is to collect a trace of page accesses that can be used to speed up the programmed pass via hint passing to the OS about upcoming memory accesses. We generate page fault traces without kernel modifications by using `userfaultfd` to intercept page faults and add the faulting address to a trace. To ensure that the trace is concise, we borrow 3PO's techniques of tracing on a page-level granularity and collecting page faults in batches that will be prefetched together, which enables us to remove duplicate faults within the batch and reduce the trace size. To synchronize the trace with the programmed pass's execution, we adapt the idea of using key pages that are guaranteed to generate a page fault when accessed as synchronization points and prefetch pages in batches, as well as prefault select pages to populate the page table ahead of time as an additional optimization. For both prefetching and prefaulting, as well as marking pages that are no longer needed, we

use `madvise` to inform the OS about future memory usage, with the advice values `MADV_WILLNEED` for prefetching, `MADV_DONTNEED` for dropping pages, and `MADV_POPULATE_READ` and `MADV_POPULATE_WRITE` for pre-populating page tables.

4.2 Content-Oblivious Memory Allocation

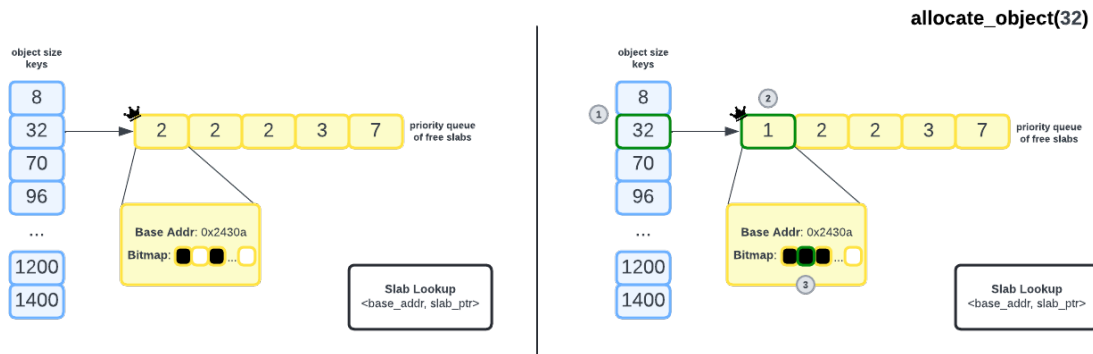
In order for the speculative pass to be efficient, we design the speculative process to have a small memory footprint and to run quickly. To do so, we observe that we can actually trade off correctness for memory efficiency and speed, so long as the memory access patterns of the speculative process remain unaffected, as we do not need to generate the correct program outputs in the speculative pass—we only need to this in the programmed pass. Given this, we elect to construct a specialized content-oblivious memory allocator that allocates memory in a separate memory region that we map to a small set of physical pages but has its own data structures stored in the regular portion of memory. Data structures that do not impact the memory access patterns of the program, notably ciphertexts, are then allocated using this allocator. We provide more details about the structures and implementation of the slab allocator below.

To configure the allocator for each respective pass, we create two allocation modes, regular and speculative, which determine whether memory is mapped normally or to the small set of physical pages respectively. The latter mode is used in the speculative pass to reduce memory usage while the former is used in the main run to guarantee correctness in program output. We reserve the entire content-oblivious memory region upfront to ensure that the address space for the content-oblivious data structures is contiguous.

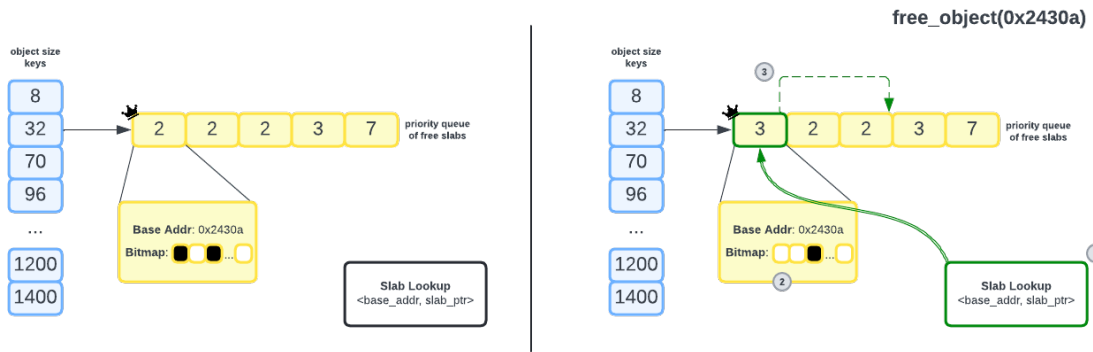
Slab Allocation

To allocate memory from the content-oblivious memory region, we favor a slab allocation design for two reasons. The first stems from our observation of a pattern of repeated allocations of identically-sized oblivious objects through our experience with popular cryptographic frameworks. By using an allocation scheme that allocates and frees memory such that objects of the same size are stored contiguously, memory fragmentation can be greatly reduced. Second, since the allocator’s data structures are stored in the regular portion of memory, we need to ensure that allocations are tracked in a memory efficient manner, which is not the case with a naive one-to-one mapping of address to object. Instead, we maintain “slabs” that track multiple objects with the use of a bitmap, enabling us to maintain mappings of address to slabs, which minimizes the size of regular memory data structures.

Concretely, we keep a mapping of object sizes to respective priority queue-like structures of unfilled slabs that prioritize filling the most filled slab first to improve memory efficiency. Further, we maintain a separate mapping of `<base_addr, slab_ptr>` key value pairs for looking up the address range that a slab tracks.



(a) Call to `allocate_object()`.



(b) Call to `free_object()`.

Figure 4.1: Slab allocator in action.

On calls to `allocate_object()`, the allocator will first check for unfilled slabs to allocate from before going to the content-oblivious memory region for additional memory. Upon retrieving an appropriate slab, the next available address is calculated using the tracked base address and bitmap and returned. The free slot count key for the priority queue is then updated to reflect the newest allocation. Figure 4.1a shows an example of a call to `allocate_object()` to allocate an object of size 32.

On calls to `free_object()`, allocated memory is not freed immediately and is instead simply marked as available, in anticipation of a subsequent allocation of an object of the same size. The slab that tracks the memory address corresponding to the memory being freed is found using a map that stores `<base_addr, slab_ptr>` key value pairs, and the bitmap and free slot count of the slab are updated. Figure 4.1b shows an example of a call to `free_object()`.

4.3 File I/O & Networking Overrides

We observe two additional opportunities to improve the usability of the speculative pass, which we detail below.

File I/O

The file outputs generated by the speculative pass are no longer needed after speculation, as in the best case, they would simply be duplicates of those generated by the programmed pass, and in other cases, they may contain garbage data since the speculative pass does not guarantee correctness. It can be laborious and difficult, however, for a user to manually go and delete these generated files, as programs may generate many files that can be complexly nested within the file system. To prevent these files from polluting the file system, we create and mount an overlay file system for the speculative process, which creates a separate overlay file system for speculation logic. This ensures that speculative process's file interactions do not alter the actual file system. After program execution, the overlay file system can be cleanly deleted, as the execution outputs of the speculative process are not used and thus no longer needed.

Networking

There are a number of SC protocols that involve multiple parties that communicate with each other, with one example being oblivious transfer in multi-party computation. This requires parties to be run synchronously in a normal program execution. However, since the content of these network interactions do not interfere with the memory access patterns of an oblivious program nor do we need program correctness in the speculative pass, we can enable asynchronous speculation by eliminating network I/O in the speculative pass. We do so by providing `fread` and `fwrite` override functions, namely `speculative_fread` and `speculative_fwrite`, that preserve the memory access patterns of the syscalls by touching the actual data without performing any I/O. This enables the speculative process to continue to generate the correct program trace without needing to be run synchronously with other parties involved in the protocol.

4.4 Interfacing with External Libraries

Annotating Oblivious Data Structures

To reduce the developer overhead of integrating oblivious allocation functionality into their libraries, we implement a base class that calls the oblivious allocator and use template specialization (C++ 20 feature) to override the underlying allocation calls for objects that are annotated to inherit from this base class. External libraries that wish to mark data structures to be allocated obliviously will simply annotate these objects to inherit from this

```
1 #include "osprey/lib/annotation.hpp"
2
3 class Ciphertext : public ContentObliviousStructureBase {
4     ...
5 }
```

Figure 4.2: Annotating a Ciphertext class.

base class. Future constructions and deconstructions of the annotated objects will call the oblivious allocator under the hood and return a pointer to an address in the content-oblivious memory region. Figure 4.2 showcases how a Ciphertext class can be annotated.

Integrating Overrides

For file I/O overrides, developers do not need to make any changes to their library and the overlay file system will be automatically set up for the speculative pass on program execution with our system. For libraries with networking functionality, networking overrides can be integrated by wrapping networking code with a conditional that checks whether our system is running in speculative mode. Calls to `fread` and `fwrite` should also be replaced with calls to `speculative_fread` and `speculative_fwrite`, which preserve the memory access patterns of the syscalls but do not perform the network interactions.

Running a Program

A target program can be run with our system without rewriting it to a low level or DSL program. To run a program with our system, we provide an executable and command line flags that can be used to set program options. An example execution of a speculative pass looks as follows:

```
$ ./osprey --speculative-only --trace-file=run.trace --lookahead=1000 ./my_prog
```

4.5 Implementation

We implement a prototype of Osprey in C++, which consists of $\approx 3,000$ lines of code, which excludes comments and blank lines (measured using `cloc`). We build our system with `clang++` 14.0.0 with compiler flag `-std=c++20` to use C++ 20 features. We compile a shared library file for integration with external libraries.

Chapter 5

Adapting Cryptographic Frameworks

We integrate Osprey with three external cryptographic frameworks to determine the developer overhead associated with using our system. We selected these libraries since they are frequently used in MPC development and research, with `emp-toolkit` [27] having nearly 200 stars on GitHub and 250+ citations on Google Scholar, Microsoft SEAL [24] nearly 3.5k stars, and MP-SPDZ [15] having 800+ stars and 400+ citations. They also have fundamental differences in complexity and implementation that enable us to fully test out our system, particularly the oblivious allocator and networking overrides. In integrating with these libraries, we also survey the lines of code changed to assess the ease of integration and usability of our system.

5.1 `emp-toolkit`

`emp-toolkit` [27] contains a set of MPC frameworks implemented as garbled circuits and was built with the aim of allowing researchers to quickly prototype protocols to assess their efficiency and of black-boxing cryptographic techniques.

Annotations

A `Bit` object underlies `emp-toolkit`'s `Integer` and `Float` representations, allowing us to annotate just the `Bit` object to enable oblivious allocation.

Networking Overrides

We implement networking overrides for `emp-toolkit` since the garbler and evaluator communicate in the garbled circuits protocol. To disable networking solely in the speculative pass, we wrap sections of `emp-toolkit`'s `NetIO` code in a conditional that checks whether the code is being executed by the speculative pass or the programmed pass. This includes socket connection logic, where the program listens for and accepts connections on a socket and initiates

connections on a socket. We similarly add a conditional check to where the code interacts with the network buffer and call the `speculative_fread` and `speculative_fwrite` in the speculative pass to preserve the memory access patterns of `fread` and `fwrite`.

5.2 Microsoft SEAL

Microsoft SEAL [24] is a homomorphic encryption library that allows additions and multiplications directly on encrypted integers and real numbers, with the goal of making HE primitives available to a wider audience. SEAL provides two homomorphic encryption schemes, BFV and BGV, which allow modular arithmetic on encrypted integers, and CKKS, which enables additions and multiplications on encrypted real or complex numbers.

Annotations

SEAL contains an internal memory manager representation that features logic via memory manager profiles that determine which virtual memory pools to allocate memory from. These memory pools make underlying calls to `SEAL_MALLOC` and `SEAL_FREE` macros, which allocate `seal_byte` pointers using `malloc` and `free`. SEAL’s ciphertext and plaintext objects both have their memory allocated from these memory pools. To allocate SEAL’s ciphertext objects obviously, we create new `SEAL_MALLOC_OBLIVIOUS` and `SEAL_FREE_OBLIVIOUS` macros that call the respective oblivious `malloc` and `free` allocation functions from our library. We then create an oblivious memory pool and profile that make underlying calls to these newly created macros and modify the `Ciphertext` class to retrieve memory from the oblivious memory pool.

Networking Overrides

Microsoft SEAL does not perform any networking. Thus, overrides for networking are not necessary.

5.3 MP-SPDZ

Multi-Protocol SPDZ (MP-SPDZ) [15] extends an implementation of the MPC protocol to 34 variants that cover honest/dishonest majority and semi-honest/malicious corruption security models and both binary and arithmetic circuits.

Annotations

High-level programs can make use of 9 basic types that enable secret and cleartext values, as well as container types like arrays and matrices. Basic types use “registers” in the virtual machine, which are allocated on an ongoing basis and thread-specific while container types

	Total files mod.	Total lines mod.	% of library lines mod.
emp-toolkit	5	76	0.058%
Microsoft SEAL	11	721	1.088%
MP-SPDZ (Shamir)	2	10	0.010%

Table 5.1: Complexity of integrating our system into external crypto libraries. The number of lines for Microsoft SEAL is in the worst case and can be further optimized. MP-SPDZ modifications are for the Shamir secret sharing protocol only and do not include networking overrides, which we leave for future work.

use “memory,” which is allocated statically and shared between threads. Since MP-SPDZ supports 34 protocol variants, each with different object types but similarly structured implementation, we focus on just one protocol—the Shamir secret sharing protocol; however, these techniques should apply similarly to the other protocols as well. For the Shamir secret sharing protocol, the secret type is `ShamirShare<T>` and the cleartext is of type `T`, which means that we can simply annotate the `ShamirShare` object that underlies the secret type to be allocated obliviously.

Networking Overrides

We do not incorporate networking overrides for MP-SPDZ and elect to save this for future work.

5.4 Analysis of Adoption Overheads

In both `emp-toolkit` and `MP-SPDZ`, we find that the number of lines of code that need to be modified is fewer than 100 lines of code, which accounts for less than 0.1% of the total lines of code in these libraries. Microsoft SEAL requires roughly 700 lines of code changes due to its internal memory manager representation, which required additional structures to be created for oblivious allocation. With further optimizations, this number should be able to be brought down to less than one hundred lines of code changes, though even at its worst case, the total lines modified still accounts for less than 1.1% of the total lines of code. Table 5.1 quantifies the file and line changes for adapting each library.

For library developers, these modifications look to be feasible, especially since these developers would be in tune with the ciphertext objects and networking logic in their own code. Additionally, most applications that use these frameworks will not need to make any modifications and can use them out-of-the-box as usual, meaning that integrating our system will not increase the complexity of usage for application developers.

Chapter 6

Evaluation

We evaluate the performance of Osprey on two distinct SC workloads and compare to a system with unbounded physical memory and a system using classical OS paging.

6.1 Workloads

We evaluate on two data-intensive SC workloads that are used in SC applications, as we have access to native implementations of these two workloads in the cryptographic frameworks that we have adapted from MAGE. We save implementing and evaluating on the remaining workloads from Chapter 3 for future work, as they were written in MAGE’s DSL and cannot be easily run independently of MAGE.

Merge

One SMPC application is federated data analytics, where a central party wants to learn about some property of the data distributed across multiple clients, with the constraint that the data cannot be centrally collected. A commonly used operation in federated analytics is a federated equi-join, which combines relations held by distinct clients without revealing any information about each client’s individual data. One method of implementing federated equi-joins is sort-merge, which entails merging sorted lists—this is our first benchmark workload, which we refer to as **merge**.

Real Statistics

Homomorphic encryption is an SC application that enables homomorphic operations on encrypted data without decryption and requires only one party to encrypt and decrypt data, which is useful in cases where there are limited number of participants. The CKKS HE scheme [6] enables additions and multiplications on vectors of real and complex numbers and yields approximate results, consisting of add-multiply circuits. Our second workload involves computing the mean and variance of real numbers, which we refer to as **rstats**.

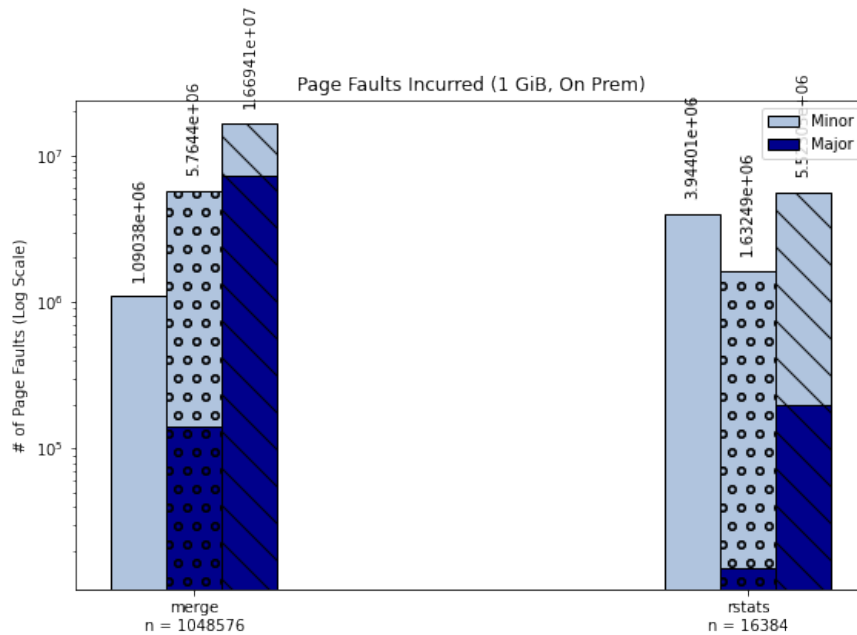


Figure 6.1: Major and minor page faults incurred. The bar on the left (no pattern) represents a system with unbounded memory, the middle bar (circle pattern) represents Osprey’s programmed pass, and the bar on the right (diagonal line pattern) represents a system using classical OS paging. The y-axis is logarithmic.

6.2 Benchmarks

We run benchmarks on merge and rstats using on-prem machines against baselines of a system with unbounded physical memory, which we refer to as **unbounded** and a system using classical OS paging, which we refer to as **OS**. We create 1 GiB cgroups, which we use to bound program memory for Osprey and for the classical OS paging baseline, and use `/usr/bin/time -v` to collect metrics. We run the speculative and programmed passes separately and disable networking overrides, as they have not yet been optimized.

Page Fault Count

We measure the impact of informed paging from user space by comparing the major and minor page fault counts of our programmed pass with the unbounded, which we expect to have no major page faults, and OS baselines. A major page fault reflects a fault where the page has to be read in from disk, and a minor fault is one in which the page has been read into memory but has not yet been mapped into the page table. Figure 6.1 shows the major and minor page fault counts on a logarithmic scale. Our system shows a $52.3\times$ reduction in major page faults compared to the OS baseline for merge and a $12.9\times$ reduction for rstats,

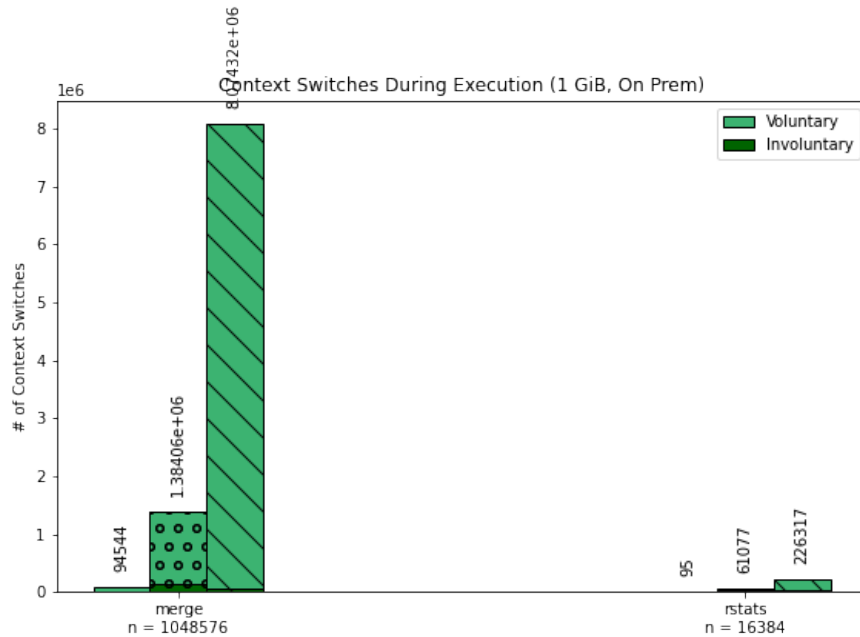


Figure 6.2: Voluntary and involuntary context switches during program execution. The bar on the left (no pattern) represents a system with unbounded memory, the middle bar (circle pattern) represents Osprey’s programmed pass, and the bar on the right (diagonal line pattern) represents a system using classical OS paging.

which highlights the effectiveness of informed paging.

Context Switches

We compare the number of context switches during program execution, focusing mainly on voluntary context switches, and find that our system voluntarily context switches $6.4\times$ less than OS for both merge and rstats, indicating a reduced overhead from waiting for data from disk. Figure 6.2 shows the number of context switches during program execution for each workload.

Timing

The timing results for unbounded, the programmed pass of our system, and OS are shown in Figure 6.3. We find that while our system performs $1.67\times$ faster than OS for merge, it performs $1.35\times$ slower for rstats. Looking at the breakdown of time spent in user space, kernel space, and blocked in Figure 6.4, both workloads see an increased amount of time spent in user space and kernel space with the programmed pass, which can be attributed to the user space centered prefetching approach and the `madvise` syscalls made for informed

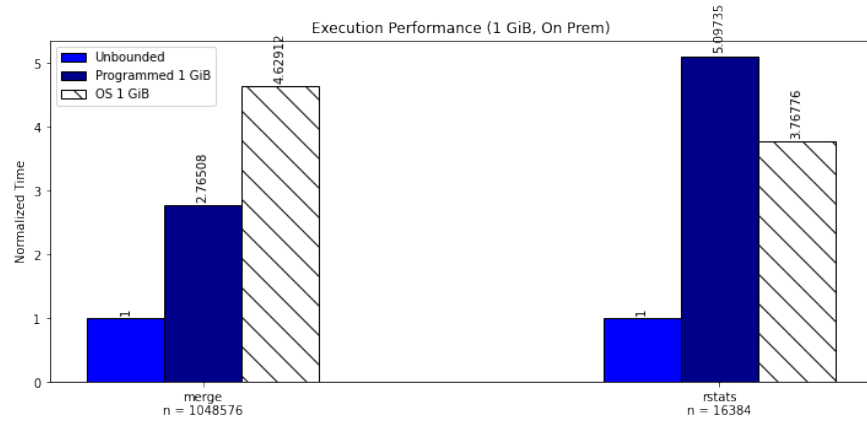


Figure 6.3: Normalized execution performance for merge and rstats. The bar on the left (blue) represents a system with unbounded memory, the middle bar (dark blue) represents Osprey’s programmed pass, and the bar on the right (white with diagonal line pattern) represents a system using classical OS paging.

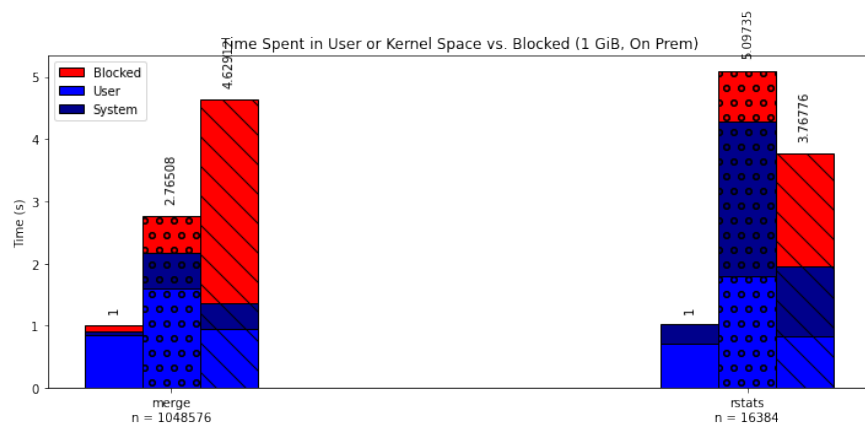


Figure 6.4: Breakdown of execution performance for merge and rstats by time spent in user space, kernel space, and blocked. The bar on the left (no pattern) represents a system with unbounded memory, the middle bar (circle pattern) represents Osprey’s programmed pass, and the bar on the right (diagonal line pattern) represents a system using classical OS paging.

paging. With further optimizations to our design, the time spent in user and kernel space should be able to be brought down to be closer to those seen in the simulation results in Chapter 3.

6.3 Performance Analysis

The page fault count and context switching results, as well as results for the merge workload, show that our system performs favorably compared to OS and is effective in managing the memory overhead of data-intensive secure computations. With the incorporation of the optimizations mentioned in Section 6.2, the performance of our system should be able to approach those seen in the simulation results of the paper. We focus on this as immediate future work.

Chapter 7

Discussion

7.1 Related Work

Predictive Paging

I/O Speculation

A number of works have used speculative execution to predict an application's future data accesses to speed up operations. Chang and Gibson [5] propose an automatic prefetching technique that involves speculatively executing an application while stalled waiting for disk and analyzing future read accesses to issue hints accordingly. They introduce a speculative thread that executes when the main thread is blocked and issues hint calls and use binary modifications to transform applications to speculate, showing that it is possible to speculate and provide hints about future read accesses with little observable overhead. Fraser and Chang [12] implement a speculative process and achieve correctness and performance by ensuring that the speculative process does not produce or change the output of the original application and that the speculative process's resource utilization does not hurt the performance of normal processes. This speculative process works to issue prefetches on behalf of its parent process and runs ahead of the parent by continuing to run without waiting for non-resident data. Li et al. [19] design a competitive prefetching strategy based on balancing the tradeoffs between conservative prefetching, whose cost can be dominated by high I/O switch overhead, and aggressive prefetching, which may result in wasted I/O bandwidth when prefetching unnecessary data. They also discuss adaptive prefetching strategies to deal with situations where there is high memory contention, which is the case with secure computation applications. We draw on a number of principles of prior I/O speculation works in designing Osprey's speculative pass, with a prominent one being ensuring that speculation occurs with minimal additional program overhead. These works also influence our plans for future work of concurrently speculating and running the programmed pass, which we discuss in more depth in Section 7.2.

Tracing

Capturing traces is a long-standing method for debugging and performance evaluation, with a notable use case for such traces being in trace re-execution for performance prediction. Recording traces without imposing high overheads to the program or requiring substantial modifications to the system are key goals for tracing applications. Jones [14] focuses on improving from low-level mechanisms for handling system calls that would require reimplementing large portions of the system interface, developing a toolkit that simplifies interposing user code between applications and system interfaces. Burton and Kelly [4] design ULTra, a user mode trace mechanism that intercepts systems calls and records traces to a file with minimal interference to the system. They extend ULTra to capture a workload’s paging activity [3] and apply optimizations to reduce the size of the captured trace. 3PO [2] implements tracing for prefetching, using an in-kernel tracer to record page accesses in the page fault handler, with a focus on collecting a concise trace efficiently. They accomplish this by tracing at the granularity of a page and batching faults into microsets to avoid redundant recordings of accesses, which are then prefetched together during re-execution. We similarly focus on generating traces efficiently and concisely and pull many of 3PO’s techniques in generating a concise trace but diverge from prior works in our user-space approach to trace collection.

Compilers for Secure Computation

A separate line of work to make secure computation more efficient and accessible to non-experts is general-purpose compilers for executing multi-party computation on arbitrary functions. These compilers focus on reducing the developer burden of having to design custom MPC protocols, instead enabling users to write high-level descriptions that can be compiled into MPC protocols. Fairplay [21] is an early work that enables developers to write code in a high-level Secure Function Definition Language (SFDL) and compile it down to a boolean circuit. Later works build on the angles of efficiency and usability, like OblivM [20], which provides a more intuitive programming language and user-friendly oblivious programming abstractions that compile into efficient SC representations. TinyGarble [25] is another that generates optimized boolean circuits for secure computation with a focus on compactness and scalability, which reduces the memory footprint of circuit operations.

7.2 Future Work

While currently, the speculative and programmed passes are run separately, the eventual goal for Osprey is for speculation to occur concurrently with execution. The main challenges of concurrently speculating is threefold: the speculative pass must (1) run quickly to ensure that it is consistently ahead of the programmed pass, (2) maintain a small memory footprint, and (3) not interfere with program correctness.

A number of the existing components of Osprey have been built with this in mind. To ensure that the speculative pass runs quickly, we adopt efficient trace collection techniques and satisfy page faults quickly within the content-oblivious memory region. The specialized memory allocator also enables the speculative pass to have a small memory footprint. Finally, because both the speculative and programmed pass interact with the same file system and interface with the network through the same ports, the implemented file I/O and networking overrides prevent incorrect behavior during program execution.

Chapter 8

Conclusion

This paper proposes Osprey, a design for a memory managed secure computation system with a focus on usability. Our system employs a lightweight speculative pass that generates page fault traces and hint-passing in the programmed pass to inform the OS of upcoming page faults. We detail the construction of a content-oblivious memory allocator to enable efficient speculation and file and networking overrides to enhance usability. Through integration case studies on three external cryptographic frameworks, we showcase the minimal code changes necessary to integrate Osprey. From evaluating our system on two distinct, data-intensive SC workloads, we show reductions in page faults and context switches compared to a baseline using classical OS paging, as well as favorable timing performance for one workload and opportunities for optimizations from the other.

Bibliography

- [1] Rachad Alao, Miranda Bogen, Jingang Miao, Ilya Mironov, and Jonathan Tannen. “How Meta is working to assess fairness in relation to race in the US across its products and systems”. In: *Meta Technical Report* (2021).
- [2] Christopher Branner-Augmon, Narek Galstyan, Sam Kumar, Emmanuel Amaro, Amy Ousterhout, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. “3PO: Programmed Far-Memory Prefetching for Oblivious Applications”. In: *arXiv preprint arXiv:2207.07688* (2022).
- [3] Ariel N Burton and Paul HJ Kelly. “Performance prediction of paging workloads using lightweight tracing”. In: *Future Generation Computer Systems* 22.7 (2006), pp. 784–793.
- [4] Ariel N Burton and Paul HJ Kelly. “Tracing and reexecuting operating system calls for reproducible performance experiments”. In: *Computers & Electrical Engineering* 26.3-4 (2000), pp. 261–278.
- [5] Fay Chang and Garth Gibson. “Automatic I/O hint generation through speculative execution”. In: (1999).
- [6] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. “Homomorphic encryption for arithmetic of approximate numbers”. In: *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer. 2017, pp. 409–437.
- [7] Jihoon Cho, Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. “Transciphering framework for approximate homomorphic encryption”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2021, pp. 640–669.
- [8] Jeff Desjardins. “How much data is generated each day”. In: *World Economic Forum*. Vol. 17. 2019.
- [9] David Evans, Vladimir Kolesnikov, and Mike Rosulek. “A pragmatic introduction to secure multi-party computation”. In: *Foundations and Trends® in Privacy and Security* 2.2-3 (2018), pp. 70–246.
- [10] Fireblocks. *Fireblocks’ Multi-layer Philosophy for Securing Digital Assets*. 2020.

- [11] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. “CBMC-GC: an ANSI C compiler for secure two-party computations”. In: *International Conference on Compiler Construction*. Springer. 2014, pp. 244–249.
- [12] Keir Fraser and Fay Chang. “Operating System I/O Speculation: How Two Invocations Are Faster Than One.” In: *USENIX Annual Technical Conference, General Track*. 2003, pp. 325–338.
- [13] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. “Faster secure Two-Party computation using garbled circuits”. In: *20th USENIX Security Symposium (USENIX Security 11)*. 2011.
- [14] Michael B Jones. “Interposition agents: Transparently interposing user code at the system interface”. In: *Proceedings of the fourteenth ACM symposium on Operating systems principles*. 1993, pp. 80–93.
- [15] Marcel Keller. “MP-SPDZ: A versatile framework for multi-party computation”. In: *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 2020, pp. 1575–1590.
- [16] Vladimir Kolesnikov and Thomas Schneider. “Improved garbled circuit: Free XOR gates and applications”. In: *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*. Springer. 2008, pp. 486–498.
- [17] Ben Kreuter, Abhi Shelat, Benjamin Mood, and Kevin Butler. “PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation”. In: *22nd USENIX Security Symposium (USENIX Security 13)*. 2013, pp. 321–336.
- [18] Sam Kumar, David E Culler, and Raluca Ada Popa. “MAGE: Nearly Zero-Cost Virtual Memory for Secure Computation”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 2021, pp. 367–385.
- [19] Chuanpeng Li, Kai Shen, and Athanasios E Papathanasiou. “Competitive prefetching for concurrent sequential I/O”. In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 2007, pp. 189–202.
- [20] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. “Oblivm: A programming framework for secure computation”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 359–376.
- [21] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. “Fairplay-Secure Two-Party Computation System.” In: *USENIX security symposium*. Vol. 4. San Diego, CA, USA. 2004, p. 9.
- [22] Moni Naor, Benny Pinkas, and Reuban Sumner. “Privacy preserving auctions and mechanism design”. In: *Proceedings of the 1st ACM Conference on Electronic Commerce*. 1999, pp. 129–139.

- [23] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M Hellerstein. “Senate: a Maliciously-Secure MPC platform for collaborative analytics”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2129–2146.
- [24] *Microsoft SEAL (release 4.1)*. <https://github.com/Microsoft/SEAL>. Microsoft Research, Redmond, WA. Jan. 2023.
- [25] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. “Tinygarble: Highly compressed and scalable sequential garbled circuits”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 411–428.
- [26] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. “Conclave: secure multi-party computation on big data”. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–18.
- [27] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. *EMP-toolkit: Efficient MultiParty computation toolkit*. 2016.
- [28] Andrew Chi-Chih Yao. “How to generate and exchange secrets”. In: *27th annual symposium on foundations of computer science (Sfcs 1986)*. IEEE. 1986, pp. 162–167.
- [29] Samee Zahur, Mike Rosulek, and David Evans. “Two halves make a whole: Reducing data transfer in garbled circuits using half gates”. In: *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II 34*. Springer. 2015, pp. 220–250.