Privacy-Preserving Communications from Privacy-Preserving Computations

by

Sajin Sasy

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Sajin Sasy 2024

Some rights reserved.



Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:

Srinivas Devadas
Professor
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Supervisor(s):

Ian Goldberg
Professor
Cheriton School of Computer Science

Internal Member:

Florian Kerschbaum
Professor
Cheriton School of Computer Science

Diogo Barradas
Assistant Professor
Cheriton School of Computer Science

Internal-External Member:

Anwar Hasan
Professor
Department of Electrical and Computer Engineering

Author's Declaration

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

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

Licence

This work is licensed under Creative Commons Attribution 4.0 International. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Statement of Contributions

This dissertation includes co-authored and peer-reviewed work that has appeared in journal and conference proceedings. For the purpose of the dissertation, I expand and revise upon material from the original publications.

In particular, Chapter 3 is based on a systematization of knowledge published in collaboration with Ian Goldberg [SG24]. Chapters 4, 5, and 6 is based upon two publications written jointly in collaboration with Aaron Johnson and Ian Goldberg [SJG22a, SJG23a]. Finally, Chapter 7 expands on work written jointly with Aaron Johnson and Ian Goldberg which is being prepared for submission.

The remainder of the dissertation consists of newly written material. All of the work appearing herein was conducted under the supervision of Ian Goldberg.

Abstract

Our society is now more than ever heavily dependent on digital data and electronic communications. Every day more of our daily activities move to the digital realm, further entrenching us in this dependency. Unfortunately, along with this transition, we are witnessing a steady increase in cyber attacks and data leakages that undermine the security and privacy of our digital data.

While the majority of online communications today use end-to-end encryption to protect data in transit, the *metadata* of who is conversing with whom, when, how much, and how often remains unprotected. Such metadata by itself can often reveal a lot about an individual to malicious network adversaries. In the context of digital services, while our personal data is protected in transit via end-to-end encryption, they are eventually stored in the clear in large data centers for processing and extracting utility. Such data centers have hence turned into coveted attack targets for malicious actors. Consequently there is a surge in research towards privacy-preserving computations over data, wherein the underlying data remains protected from malicious actors that could gain unauthorized access to a data center.

One approach towards such privacy-preserving computations that has gained practical traction is Trusted Execution Environments (or TEEs). However, TEEs introduce their own set of challenges. Research has demonstrated TEEs' susceptibility to side-channel attacks that undermine their purported security and privacy guarantees. In this dissertation, we start by establishing the notion of 'fully oblivious' algorithms. By definition such algorithms do not leak any information about the underlying private data they operate over via such side channels.

We then provide several fully oblivious algorithms for the fundamental and recurring data operations of compacting, shuffling, and sorting of data. Oblivious compaction, shuffling, and sorting are core building blocks underlying several privacy-preserving solutions today, and are often the expensive bottleneck of such solutions. Our new algorithms improve over the state-of-the-art approaches for these primitives in some combination of their asymptotics, underlying constants, and parallelizability, and with demonstrable improvements in concrete performance always, enabling efficient and secure deployments of privacy-preserving computations.

Finally, we propose a novel design for a metadata-protecting communication system (MPCS) TEEMS designed to facilitate metadata-protected messaging. While there has been tremendous research towards novel MPCS designs, there has been a clear dearth of MPCS constructions that can simultaneously support (i) low latency, (ii) high throughput, (iii) horizontal scalability, and (iv) asynchronicity. TEEMS presents an MPCS that

can simultaneously support all four desirable properties for messaging systems by leveraging our new efficient fully oblivious algorithms as building blocks.

Acknowledgements

This dissertation and each of the works presented herein would not have been possible without the support, encouragement, and help from many wonderful individuals that I was fortunate to have met over the course of my graduate studies.

First and foremost, Ian Goldberg, for his incredible supervision over the course of my Ph.D. Ian has been a fantastic mentor, intellectual oracle, and good friend, that I can bring anything and everything to over the last several years. His unwavering positive attitude towards research and science, meticulous attention to detail, and penchant for questions, has always been a source of inspiration and has strongly influenced my own growth. My few words here cannot possibly do justice for all the guidance I have enjoyed over the years. Thank you.

My many thanks to my incredible collaborators and coauthors — Edward Eaton, Xi He, Aaron Johnson, Gautam Kamath, Shubhankar Mahopatra, Olga Ohrimenko, Andrew Paverd, Om Thakkar, and Adithya Vadapalli, that I was very fortunate to work with over the course of my Ph.D. I have always enjoyed our discussions and collaborations and I am optimistic we will continue to have more in the future.

I would also like to particularly thank the members of my examining committee for my thesis — Srini Devadas, Florian Kerschbaum, Diogo Barradas, and Anwar Hasan, for graciously taking valuable time from their busy schedules to review my thesis and provide me with insightful feedback. Also my thanks to N. Asokan for graciously being a part of my Ph.D. proposal committee.

The CrySP lab is a truly special place filled with incredible individuals that I was fortunate to be a part of. My many thanks for the wonderful memories and vortices on topics completely beyond the realms of rationality. This work also benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

Beyond my academic gratitude and debts, I am also thankful for my life-long friendships forged over the course of graduate school, the *UWat Crew*: Micky Agarwal, Tom Bury, Nate Braniff, Atluan Deep, Joshua Dierolf, Savio D'Silva, Sabria Farheen, Christian Gorenflo, Egon Fernandez, Nathan Harms, Erik Hintz, Priyank Jaini, Ryan Kinnear, Steph Lee, Amit Levi, Nupur Maheshwari, Clive Mascarenhas, Karen Mascarenhas, Anil Pacaci, Raisa Sharmin, Gillian Shi, Ali Wytsma, and Omar Zarifi. My many thanks to all of you for the wonderful times and memories we cherish together.

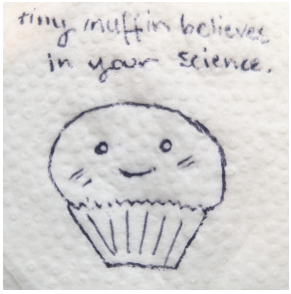
Last but not least, I thank Mr. Hobbes, the stalwart purr-reviewer of my research, for his hours of relentless dedication to academic rigour.

Dedication

To my parents, V.B. Sasy and Sunanda Sasy
for all their love and prayers.

To my brothers, Sarin Sasy and Sachin Sasy
for their infinite support and faith in my success.

To my wife, Camila María Pérez Gavilán Torres
for always being by my side and believing in me.



Thank you.

Table of Contents

Examining Committee Membership ii

Author's Declaration iii

Statement of Contributions iv

Abstract v

Acknowledgements vii

Dedication viii

List of Figures xv

List of Tables xvii

List of Abbreviations xvii

1 Introduction 1

1.1 Thesis Statement 4

1.2 Contributions 4

1.3 Organization 7

2 Trusted Execution Environments (TEEs) 8

2.1 Threat Model . . . 8

2.2 Side Channels . . . 9

2.3 TEE Applications . . . 11

3 Metadata-Protecting Communications Systems (MPCS) 14

3.1 Metadata-Protecting Communication Systems (MPCS) . . . 16

3.2 MPCS Properties . . . 18

3.2.1 Protections . . . 18

3.2.2 Usability . . . 19

3.2.3 Performance . . . 20

3.2.4 Dialing and Conversation Protocols . . . 21

3.3 Existing Work . . . 21

3.3.1 DC-net Based Systems . . . 22

3.3.2 Mixnet Based Systems . . . 23

3.3.3 Differential Privacy Based Systems . . . 26

3.3.4 Private-Read Based Systems . . . 29

3.3.5 Private-Write Based Systems . . . 31

3.3.6 Secure Multiparty Computation (SMC) Based Systems . . . 33

3.4 Discussion . . . 36

3.4.1 Tradeoffs Between Families . . . 38

3.4.2 Tradeoffs Within Categories . . . 39

3.4.3 Limitations of Security Assumptions . . . 41

3.4.4 Asynchronous Message Storage . . . 42

3.4.5 Anonymity Set Protections . . . 43

3.4.6 MPCS for Metadata-Protected Messaging . . . 44

3.5 Conclusion . . . 45

4 ORCompact: Oblivious Recursive Compaction **46**
4.1 Fully Oblivious Algorithms . . . 46
4.1.1 Fully Oblivious Assembly Verifier (FOAV) . . . 48
4.2 Oblivious Compaction . . . 48
4.3 ORCompact: Oblivious Recursive Compaction . . . 49
4.3.1 Algorithm . . . 50
4.3.2 Correctness, Obliviousness, and Efficiency . . . 52
4.3.3 Parallelization . . . 54
4.4 Related Work . . . 55
4.5 Implementation . . . 57
4.6 Formal Definitions and Proofs . . . 58
4.6.1 Obliviousness . . . 60
4.6.2 Correctness . . . 62
4.6.3 Efficiency . . . 64
4.7 Conclusion . . . 67

5 Oblivious Shuffling Algorithms in the Standard and Streaming Models **68**
5.1 ORShuffle: Oblivious Recursive Shuffle . . . 69
5.1.1 Algorithm . . . 69
5.1.2 Efficiency . . . 71
5.1.3 Parallelization . . . 72
5.2 Bucket Oblivious Random Permutation (BORP) . . . 73
5.2.1 BORPCompact . . . 74
5.2.2 BORPStream . . . 76
5.2.3 Correctness and Obliviousness . . . 78
5.2.4 Efficiency and Parallelization . . . 79
5.3 Implementation . . . 80
5.4 Related Work . . . 82

5.4.1 Oblivious Shuffling . . . 82
5.4.2 Systems Using TEEs and Oblivious Algorithms . . . 83
5.5 Proofs of Correctness and Efficiency . . . 85
5.5.1 Efficiency of ORShuffle . . . 85
5.5.2 Efficiency of BORPStream . . . 88
5.5.3 BORPStream Correctness and Optimizer . . . 88
5.6 Conclusion . . . 90

6 Oblivious Shuffling and Sorting Algorithms in the Offline/Online Model 91
6.1 Background . . . 93
6.2 Oblivious Algorithms for Waksman Networks . . . 94
6.2.1 Preliminaries . . . 94
6.2.2 Our Waksman Topology . . . 95
6.2.3 Setting Control Bits . . . 96
6.2.4 Worked Example of ControlBits . . . 101
6.2.5 Applying Permutations . . . 106
6.3 Oblivious Shuffling and Sorting . . . 106
6.3.1 Shuffling . . . 107
6.3.2 Sorting . . . 108
6.4 Analysis . . . 109
6.4.1 Correctness . . . 109
6.4.2 Obliviousness . . . 110
6.4.3 Efficiency . . . 111
6.5 Implementation . . . 113
6.5.1 Implementation Optimizations . . . 114
6.6 Experiments . . . 115
6.6.1 Comparing Shuffling Algorithms . . . 115
6.6.2 Comparing Sorting Algorithms . . . 118

6.6.3 Summarizing the State of Affairs . . . 119
6.7 Related Work . . . 120
6.7.1 Oblivious Sorting Algorithms . . . 120
6.7.2 Waksman Networks . . . 121
6.8 Proofs . . . 122
6.8.1 Correctness . . . 122
6.8.2 Obliviousness . . . 124
6.8.3 Efficiency . . . 128
6.9 Conclusion . . . 129

7 TEEMS **131**
7.1 Overview . . . 133
7.1.1 System Design and Goals . . . 135
7.1.2 Private and Public Channels . . . 137
7.1.3 Clients . . . 138
7.1.4 Servers . . . 138
7.2 Design Details . . . 139
7.2.1 User Identifiers . . . 140
7.2.2 Account Server . . . 140
7.2.3 Messages . . . 141
7.2.4 Clients . . . 141
7.2.5 Ingestion . . . 142
7.2.6 Routing . . . 143
7.2.7 Storage . . . 148
7.2.8 Friends . . . 149
7.3 Analytical Evaluation . . . 151
7.3.1 Efficiency . . . 151
7.3.2 Security . . . 153

7.4 Experimental Evaluation 154
7.4.1 Latency and throughput with fixed servers 156
7.4.2 Horizontal Scalability 157
7.5 Related Work 157
7.6 Conclusion 159

8 Conclusion **160**

References **164**

Appendices **181**

A Largest Experiment Reported for prior MPCS **182**

B Additional Experiments for WaksShuffle and WaksShuffle+QS **186**
B.1 Varying the Item Size 186
B.2 Varying the Number of Items 186

List of Figures

1.1 The standard, streaming, and offline/online models for algorithms 5

2.1 Illustration of remote execution of programs in a TEE-aided server. . . . 9
2.2 Framework for privacy-preserving applications through TEEs. 12

4.1 Tight compaction illustration 48
4.2 ORoffCompact algorithm 51
4.3 ORCompact algorithm 51
4.4 OROCPar algorithm 55
4.5 ORCPar algorithm 56
4.6 Number of OSwaps in compaction 57
4.7 Computation time for compaction (varying number of items) 57
4.8 Computation time for compaction (varying block size) 58

5.1 ORShuffle algorithm 70
5.2 MarkHalf algorithm 70
5.3 ORSPar algorithm 72
5.4 MarkPar algorithm 72
5.5 MSN: MergeSplitNode class 75
5.6 BORPStream algorithm 77
5.7 Comparison of computation time of shuffling algorithms 81

6.1 Standard Waksman network topology 93

6.2 Our new Waksman network topology . . . 96
6.3 ControlBits algorithm . . . 97
6.4 InBits algorithm . . . 100
6.5 CreateForwardLookup algorithm . . . 101
6.6 ForwardOrRand algorithm . . . 102
6.7 CreateReverseLookup algorithm . . . 102
6.8 CreateUnselectedCounts algorithm . . . 103
6.9 DecUnselectedCounts algorithm . . . 103
6.10 Worked example of a Waksman network . . . 104
6.11 ApplyPerm algorithm . . . 106
6.12 ApplyInvPerm algorithm . . . 107
6.13 WaksShuffle algorithm . . . 108
6.14 WaksSort algorithm . . . 108
6.15 Comparison of shuffling algorithms (varying block size) . . . 116
6.16 Comparing time taken for sorting algorithms (varying block sizes) . . . 118
6.17 Scatter plots comparing online and total time for shuffling algorithms . . . 119
6.18 Scatter plots comparing online and total time for sorting algorithms . . . 120

7.1 TEEMS architecture . . . 136
7.2 Client interaction procedure . . . 142
7.3 ColumnRoute algorithm . . . 144
7.4 Algorithm for the first round of ColumnRoute . . . 146
7.5 Algorithm for the second round of ColumnRoute . . . 146
7.6 Algorithm for the third round of ColumnRoute . . . 147
7.7 Evaluating TEEMS' latency . . . 155
7.8 Evaluating TEEMS' horizontal scalability . . . 157

B.1 Comparing time taken for shuffling/sorting with fewer clients (varying block size) . . . 187
B.2 Comparing time taken for shuffling/sorting with fixed block size (varying clients) . . . 187

List of Tables

3.1 A comparison of Metadata Protected Communications Systems (MPCS) 37

5.1 BORPStream's optimizer generated parameters 89

6.1 Analytical comparison of oblivious shuffling algorithms 113

7.1 The parameters in TEEMS 133
7.2 Notations used for TEEMS 134
7.3 Computation per epoch for a client or server 151
7.4 Communication per epoch for a client or server 152
7.5 Comparing TEEMS and existing MPCS for metadata-protected messaging 158

8.1 A comparison of Metadata Protected Communications Systems (MPCS) 163

A.1 Details of the largest experiments reported for prior MPCS 184

List of Abbreviations

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

AEX Asynchronous Enclave Exit

ART Anonymous Routing Tokens

AS Anonymity Set

BE Boomerang Encryption

BORP Bucket Oblivious Random Permutation

BRN Butterfly Routing Network

CPIR Computational Private Information Retrieval

CUS Communication Unobservable Systems

DP Differential Privacy

DPF Distributed Point Functions

ECB Electronic Code Book

ECDSA Elliptic Curve Digital Signature Algorithm

FOAV Fully Oblivious Assembly Verifier

ITPIR Information Theoretic Private Information Retrieval

MAC Message Authentication Code

MEE Memory Encryption Engine

MPCS Metadata Protected Communication Systems

MSN MergeSplit Node

NIZK Non-Interactive Zero Knowledge

ORCompact Oblivious Recursive Compaction

ORCPar Oblivious Recursive Parallel Compaction

ORShuffle Oblivious Recursive Shuffle

OSwap Oblivious Swap

PIR Private Information Retrieval

PRM Processor Reserved Memory

QS Quick Sort

RE Resource Exhaustion

RPIR Reverse Private Information Retrieval

RUMS Receiver Unlinkable Messaging Systems

RUS Relationship Unobservable Systems

SGX Software Guard eXtension

SMC Secure Multiparty Computation

SNIP Secret-shared Non-Interactive Proofs

SUBS Sender Unlinkable Broadcast Systems

SUMS Sender Unlinkable Messaging Systems

TEE Trusted Execution Environments

TME Total Memory Encryption

Chapter 1

Introduction

Over the last few decades communication infrastructures have made significant strides, making ideas that seemed impossible a reality. Individuals from opposite ends of the world communicate seamlessly over the Internet using several choices of popular messengers like Signal, Whatsapp, or Telegram. We innately expect such conversations we have over the Internet to have the same privacy as us conversing with our correspondent in person in a private space. The Snowden revelations from 2013 [Gua13], however, revealed that this expectation was fallacious, resulting in a widespread call to arms within the security and privacy community to proactively resist mass surveillance [USO14, Rog15]. In practice, this eventually culminated with the adoption of end-to-end encryption as a standard for the vast majority of communications over the Internet [Sig13, WIR14, IET18].

End-to-end encryption protects the contents of communications between two parties from network adversaries. Nonetheless, by virtue of how these communications tools operate, they inherently leak the *existence* of a conversation and the *metadata* of *who conversed with whom, when, and how much they conversed*. Such metadata or *'non-content'* components of our digital communications suffer from weak to non-existent legal protections, and consequently it can be gathered and analyzed en masse by network adversaries that simply eavesdrop on our online activities.

Crucially, 'non-content' does not mean 'non-sensitive'. Even if the contents of communications between two parties remain protected via end-to-end encryption, oftentimes even simply knowing the existence of a communication can be privacy invasive. As an example, the knowledge of simply the existence of (possibly frequent) conversations of an individual with a recruiter, local alcoholics anonymous chapter, divorce attorney, or a cancer treatment center, all fundamentally reveal a lot more about the individual

through metadata alone. Things only get worse as copious amounts of metadata from different channels are available to malicious entities, that can then silently profile and surveil individuals, oftentimes even reconstructing the contents protected by encryption via just metadata. For instance, research has shown that adversaries simply observing the number of packets, their sizes, and inter-packet timings can infer the ailments of a user interacting with their online health portal, without ever having access to the encrypted contents of their interactions [CWWZ10].

In addition to making our online communications private, protecting metadata of digital communications also brings forth broader social benefits. Free and democratic societies depend on an informed public, which depends on whistleblowers shedding light on misdeeds and corruption [Gua13, Buz20]. Imagine the plight of a whistleblower, faced with the daunting task of publishing confidential documents that would expose acts of incompetence, fraud, or inhumane practices carried out by their government or employer that thrives on deceiving the public. Any action in such a direction often results in incarceration or risks to their lives [Was18, BBC21]. The same is true of individuals in oppressive regimes whose lives are endangered by their mere sexual orientation or political stance [Reu18, Deu19]. In the era of global surveillance that we live in, ultimately, metadata might be sufficient for terrible consequences [NYR14, ACL14]. Hence, protecting individuals' communications online today requires communications systems with strong security and privacy guarantees for both data and metadata.

Research in this direction towards communication systems that protect metadata predate the Snowden revelations [Cha81, Cha88, SCM05, CGF10, WCGFJ12a, WCGFJ12b, CGWF13], and have received a renewed attention since [CGBM15, VDHLZZ15, KLDF16, AS16, TGL+17, KCGDF17, PHE+17, LGZ18, LYK+19, APY20, ECGZB21, NSSD22, VSH22, LK22, JWC+23]. However, these designs have varying privacy and functionality goals, making it hard to understand, compare, and identify shortcomings of designs so far. To remedy this, we presented a Systematization of Knowledge (SoK) on Metadata-Protecting Communication Systems (MPCS) [SG24]. MPCS is an intentionally loosely defined term to broadly capture several different forms of metadata-protecting systems that we have seen so far. Our systematization provides two different taxonomies for existing MPCS constructions. The first taxonomy is based on the family of construction, i.e., the core underlying technique of the design. We identified six different families of constructions for MPCS, namely, (i) DC-nets, (ii) Mixnets, (iii) (Mixnets +) Differential Privacy, (iv) PIR, (v) Reverse-PIR (RPIR), and (vi) Secure Multiparty Computation (SMC) based systems. The second taxonomy we call the type of MPCS, which assigns precise labels for such systems based on the combination of their privacy and functionality goals. We will discuss both these taxonomies at depth later in Chapter 3. MPCS typically support one

of the following two functionality goals: (i) broadcast systems, which aim to provide metadata protections to an individual that wishes to broadcast a message to all other participants of the system, or (ii) end-to-end systems, that provide metadata protections to pairs of users corresponding through the system.

In this dissertation, we focus on the latter problem of end-to-end metadata-protected communications. In particular, this dissertation focuses on metadata-protected messaging systems that can support end-to-end metadata-protected conversations between two users. By messaging systems, we mean systems like Signal and Whatsapp that have become one of the dominant forms of online communications today. Users of such popular messaging systems today enjoy very low latency (about 100 ms) conversations, and in fact low latency has become a baseline expectation from any messaging system today. Metadata-protected messaging systems are a type of end-to-end MPCS, but with additional constraints on their properties that can make them candidates for adoption by popular messaging systems. We detail below the relevant properties of existing MPCS that preclude them from being viable metadata protected messaging systems.

Despite such a vast body of literature in this direction, we are yet to see any practical widespread deployments of such metadata-protecting communication systems, let alone a metadata-protected messaging system. There are several reasons why this is the case. First, metadata-protected communication systems designs typically induce significant overheads over their traditional counterparts as such systems often use expensive cryptographic primitives like verifiable shuffles, private information retrieval protocols, or oblivious shuffle protocols. These primitives induce **impractical latencies** on the order of several minutes to deliver messages with as few as a million users [CGF10, VDHLZZ15, TGL+17, KCGDF17, AKTZ17, KLD20, EB22], with the latencies getting worse as more users join the system. Second, while some designs can support low latencies, they do so at the expense of **poor throughput** [ACLS18, ECGZB21, VSH22]. Moreover, such schemes get significantly worse latencies and throughput as the number of clients in the system increase, as such systems have **poor horizontal scalability**; i.e, the ability of a system to scale as more clients join the system by adding more servers into the system. Finally, the vast majority of existing MPCS **lack support for asynchronous clients** [Cha88, VDHLZZ15, AS16, AKTZ17, TGL+17, KLD20, EB22, JWC+23]. Such systems often make the impractical assumption that all clients will remain online and connected to the system at all times, participating in each logical round of the system sending and receiving (possibly dummy) messages.

Succinctly, as we highlight later in Chapter 3, *the core problem is the lack of MPCS designs that can support four fundamental properties desirable of any communications system, namely, (i) low latency, (ii) high throughput, (iii) horizontal scalability, and (iv) support*

for asynchronous clients.

1.1 Thesis Statement

In this dissertation, we establish the following:

> We can design an end-to-end metadata-protecting communication system that can simultaneously achieve low latency, high throughput, horizontal scalability, and asynchronous client support.

1.2 Contributions

The central focus of this thesis is in the design and analysis of a novel metadata-protecting communication system designed for messaging. The design presented herein, TEEMS, Trusted Execution Environments based Metadata-protected Messaging System, as the name suggests is based on Trusted Execution Environments or TEEs. In recent years, TEEs have gained significant academic and industry attention and is beginning to see large-scale real world deployments as well. However, TEEs also have some pitfalls, the most concerning one being their vulnerability to side-channel attacks that can undermine the security guarantees purported by this computation model. Hence in this thesis, we start with first clearly establishing the notion of 'fully oblivious' algorithms. By definition fully oblivious algorithms are immune to known software side-channel attacks, and consequently can be run within a TEE to ensure that security guarantees of data processed in such TEEs are upheld. We present more details on TEEs and vulnerabilities in Chapter 2.

Under the constraint of full obliviousness, prior approaches for even fundamental data operations like shuffling and sorting of data incurred significant overheads. However, data operations like fully oblivious shuffling and sorting of data are a cornerstone of several different data processing pipelines and have numerous applications. For instance, they happen to be the core expensive component and bottleneck in solutions for privacy-preserving telemetry [BEM+17], genomic data analysis [CWJ+16], central and federated machine learning [SO19, KCY23], contact discovery [TGS23], and database

Standard: All data to operate upon is available when the algorithm begins.

Streaming: Data for the algorithm slowly trickles in over a long period of time. Once the last item has arrived, such algorithms enter their *online* phase.

Offline/Online: Data-independent precomputations can be performed in advance during an *offline* phase, to speed up the data-dependent *online* phase of the algorithm.

Figure 1.1: The standard, streaming, and offline/online models for algorithms

operations [EZ19]. Naturally, such overheads also undermine the ability of TEEs to instantiate efficient metadata-protected messaging systems. Consequently, in this thesis, we present several new fully oblivious algorithms for fundamental data primitives that form the basic building blocks of our system. These new fully oblivious algorithms can be of independent interest and have myriad applications by themselves, both within the TEE model of computations and even in other computation models as well, and we will point out such avenues later in Chapters 4, 5, and 6 as we present our algorithms. Our fully oblivious algorithms span a few different algorithmic models, namely *standard*, *streaming*, and *offline/online*; Figure 1.1 provides a succinct description of these three models.

Below we highlight the most notable contributions of this thesis:

1. **A systematization of metadata-protecting communication systems (MPCS).** We survey and present two taxonomies for classifying prior metadata-protecting communications systems, and identify the shortcomings of designs so far.

2. **Introduce 'fully oblivious' algorithms.** We detail the different levels of obliviousness for TEE programs, and establish the notion of fully oblivious algorithms that are safe to run within TEEs.

3. **A collection of new efficient fully oblivious algorithms.** This thesis introduces seven new efficient fully oblivious algorithms that improve over the prior state-of-the-art approaches in some combinations of asymptotics, underlying constants, and parallelizability; with improvements in concrete performance always.

(a) **A new fully oblivious tight compaction algorithm ORCompact.** ORCompact has an $O(n \log n)$ asymptotic complexity matching prior state-of-the-art, but halves the underlying constant, and is efficiently parallelizable.

(b) **A new fully oblivious control bit setting algorithm for Waksman networks.** Waksman permutation networks are a configuration of binary switches that can permute data to any permutation. However such networks were limited to operating over number of items that are powers of two. We present control bit setting algorithms for Waksman networks that are fully oblivious; in addition they admit general numbers of items, not just powers of two. We also present an improved topology for Waksman networks that has better memory locality and consequently yields better performance.

(c) **Three new fully oblivious shuffling algorithms**:

i. **ORShuffle** which has the same $O(n \log^2 n)$ asymptotic complexity as prior approaches but improves the underlying constants.

ii. **BORPStream** with $O(n \log^2 n)$ complexity but presents even more improvements in constants for the streaming data model.

iii. **WaksShuffle** which uses Waksman networks for a fully oblivious *offline/online shuffle*. WaksShuffle presents a fully oblivious shuffle with an *online* complexity of $O(n \log n)$, saving an $O(\log n)$ asymptotic factor over prior fully oblivious shuffle approaches in its online time, at the expense of incurring a data-independent *offline* precomputation.

(d) **Two new oblivious sorting algorithms WaksSort and WaksShuffle+QS.** WaksSort uses our oblivious control bit setting algorithm to generate control bits to sort the data, and hence cannot leverage the offline-online partition. WaksShuffle+QS uses the 'Scramble-then-Compute' paradigm [DDCO17] to first obliviously shuffle the data with WaksShuffle, followed by a non-oblivious sort quicksort (QS). This results in a fully oblivious sort $O(n \log n)$ matching the asymptotics of non-oblivious sorting, and saving a $O(\log n)$ asymptotic factor over the standard fully oblivious approach of using a bitonic sorting network.

4. **TEEMS**, a new MPCS design for metadata-protected messaging that leverages the above efficient fully oblivious algorithms to present a system that supports low latency, high throughput, horizontal scalability, and asynchronous client support simultaneously.

1.3 Organization

The remainder of this thesis is organized as follows:

We start by presenting relevant background on Trusted Execution Environments (TEEs) in Chapter 2. We enumerate the different side-channel attacks that TEEs are vulnerable to, and list several real-world applications and research directions that leverage TEEs and consequently require oblivious algorithms for secure deployments.

Chapter 3 covers the relevant background and terminology to discuss metadata-protecting communications systems (MPCS). Here we provide a high-level overview of the many different families and types of MPCS constructions from the literature towards metadata-protecting communications systems. We highlight the lack of MPCS that can facilitate metadata-protected messaging systems with simultaneous support for low latency, high throughput, horizontal scalability, and asynchronous clients, which serves as the foundation for the thesis statement in this dissertation.

We first establish the notion of 'fully oblivious' algorithms in Chapter 4, and then present our first fully oblivious algorithm ORCompact: Oblivious Recursive Compaction, an order-preserving tight compaction algorithm.

In Chapter 5, we present two new fully oblivious shuffling algorithms ORShuffle and BORPStream. ORShuffle and BORPStream present oblivious shuffling in the standard model where all the data to shuffle is available when the algorithm begins, but additionally BORPStream presents improvements in the streaming data model when data to shuffle trickles in over a longer time period, by partially shuffling items as they arrive.

We present new fully oblivious shuffling (WaksShuffle) and sorting (WaksSort, WaksShuffle+QS) algorithms in Chapter 6. WaksShuffle and WaksShuffle+QS are tailored for the offline-online model. In the offline-online model, servers gain significant speedups in online performance when an oblivious shuffle or a sort is requested by performing some offline data-independent precomputation a priori. In this chapter we also present WaksSort. Although WaksSort cannot leverage the offline-online model, we demonstrate that all three of these algorithms perform better than prior approaches in their total time (including the offline phase) for shuffling or sorting moderately large items, since their offline precomputation is independent of the size of items.

We present our design for a metadata protected messaging system, TEEMS, in Chapter 7. TEEMS internally uses the building blocks we presented over the course of Chapters 4, 5, and 6, and presents the first MPCS that can simultaneously achieve all the four desirable properties of a messaging system.

Finally, we conclude this dissertation in Chapter 8.

Chapter 2

Trusted Execution Environments (TEEs)

Since their inception with Intel TXT [Int12] more than a decade ago, TEEs have undergone several iterations of refinements. Recent advances in hardware-aided TEEs such as Intel SGX [AGJS13], AMD SEV [KPW16], and their open-source sibling Keystone [LKS+19][1] can provably execute programs securely in hardware-protected containers called *enclaves* that provide strong *confidentiality* and *integrity* guarantees, even if its host is malicious. Effectively they provide a small Trusted Computing Base (TCB) on otherwise malicious servers. For ease of exposition, we limit the discussion of how TEEs provide these guarantees by just detailing how it is achieved in Intel SGX. In this chapter we only provide an overview and cover details relevant to TEEs for our work. For the interested reader, an in-depth dive of Intel SGX is provided by Costan and Devadas [CD16]. Intel SGX is the most mature TEE among the choices available today, and we use it to implement and benchmark our algorithms. However, the techniques used by other TEEs are similar to those of SGX, making our work applicable to them as well.

2.1 Threat Model

In Intel SGX, the enclaves provide trusted execution environments by bootstrapping security from cryptographic keys fused into the processor at manufacture time [AGJS13]. At boot time such processors set aside a portion of their available DRAM as Processor

[1] Keystone is based atop RISC-V, the open-source hardware instruction-set architecture to facilitate secure and trustworthy open-source trusted execution environments.

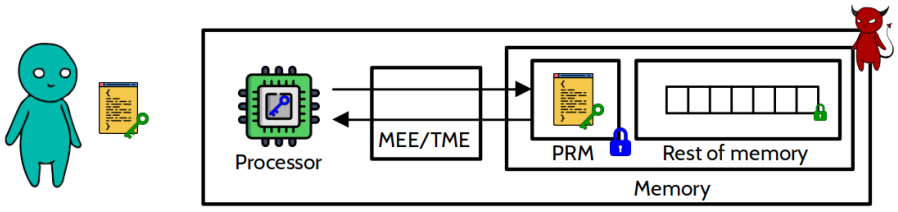


Figure 2.1: Illustration of remote execution of programs in a TEE-aided server.

Reserved Memory (PRM). In SGX version 1, all accesses to memory pages in the PRM occur through Intel's Memory Encryption Engine (MEE). MEE uses the processor-fused keys to encrypt these pages every time they leave the system cache and return to the PRM. Similarly MEE decrypts and integrity checks all pages loaded from the PRM before they are moved to the cache for use by the SGX-supported processor; thus the contents of these pages are always protected.[2] Figure 2.1 provides a high-level visual of how programs are executed in a TEE. To ensure the integrity of programs running within such enclaves, Intel supports a remote attestation protocol [AGJS13]. The attestation protocol provides an ECDSA certificate that assures the program is indeed running inside an Intel SGX enclave with details of the version of the SGX trusted computing base on that device.

The SGX threat model allows an adversary to control the software stack of an SGX-supported server, including in particular the OS. The adversary may have physical access to server but is assumed to be incapable of physically tampering with the processor chip, extracting the processor-fused keys, or snooping the state of the processor registers, which would void the purported security guarantees. Nonetheless, the adversary is formidable and can observe all the shared resources and peripherals.

2.2 Side Channels

Over the last few years, researchers have presented several side-channel attacks on SGX [XCP15, SCNS16, BMD+17, VWK+17, MIE17, LSG+17, LJF+20, MVH+20]. Broadly speaking such attacks can be grouped into two different categories: ones that are based

[2] MEE has been replaced by Total Memory Encryption (TME) in SGX version 2. TME, however, no longer supports the integrity guarantees that MEE offered. MEE supported a limited amount of enclave memory (≈100 MB), TME on the other hand allows for arbitrarily large enclave memory but integrity verification is offloaded to the enclave application.

on software side channels and the other based on microarchitectural side channels. Software side channel attacks break confidentiality by exploiting side channels that arise from secret-dependent memory accesses or control-flow branching.

Memory-access side channels have three different subgranularities of side channel leakages and consequently attacks:

1. **Page-granular attacks**: Xu et al. [XCP15] presented controlled-channel attacks, which allow a malicious OS to infer sensitive data when the victim program has input-dependent control or data transfer. Similarly, Shinde et al. [SCNS16] demonstrate that this page fault side channel can be used to extract encryption keys from implementations of cryptographic routines. This led to defences that suppress page faults during enclave execution [SCNS16, SLKP17]. However, a malicious OS observing side effects of the address translation process can still extract this leakage without relying on page faults explicitly [VWK+17].

2. **Cacheline-granular attacks**: PRIME+PROBE [OST06, LYG+15] and other cache attacks have been demonstrated outside of the SGX setting for more than a decade now. Several works [BMD+17, MIE17, GESM17] have demonstrated that such attacks in fact translate well to the SGX setting. However, these type of attacks are typically quite noisy and lossy, from cache activities of other processes on the same system and the probing process having non-negligible probability of missing the victim's access patterns. Hence they require several measurements of the victim program executions to reliably extract cache access patterns. Additionally, they require forcing interruptions of enclave executions to precise the cache access patterns. This led to defense mechanisms such as T-SGX [SLKP17] and Déjà vu [CZRZ17] that detect frequent interrupts as an indicator of the enclave program being under attack. More recently, Lee et al. demonstrated Membuster [LJF+20], an off-chip physical attack that snoops the memory addresses accessed on the memory bus, which in conjunction with novel cache manipulation tricks that force cache misses allows them to extract cache usage patterns from the victim process with minimal overheads and without inducing interrupts.

3. **Subcacheline-granular attacks**: Finally, a recent attack TEEJAM [SZB+24] has demonstrated a subcacheline attack that can precise the exact four-byte word loaded by a victim program. These attacks exploit '4k-aliasing', which is used by Intel processors' memory ordering model to ensure correctness of memory load and store operations. At a high level, 4k-aliasing results in a small delay when the victim and adversary attempt to access the same or overlapping memory, which forms the basis of these attacks. Currently, such attacks require very fine-grained

control, i.e., single-stepping [VBPS17] over the enclave's execution. Such single step based attacks have since been mitigated for SGX through Instruction Set Architecture (ISA) extensions [CBC+23], but subcacheline-granular attacks are still in their nascent stages and we are likely to see more of them.

Control-flow attacks: While several cache attacks can be used to attack the control flow of programs executing in an enclave, Lee et al. [LSG+17] demonstrated a precise technique of tracing control-flow execution by leveraging the branch prediction history, which was not cleared while switching out of enclave mode. Moghimi et al. [MVH+20] demonstrated a control-flow attack, which precisely counts the number of instructions executed within an enclave execution, which when combined with coarse-grained page-level leakages can reconstruct the exact control flow taken by the enclave program.

Another class of side-channel attacks are based on microarchitectural side channels. In particular side channels like speculative execution [VMW+18, CCX+19] or the voltage scaling interface [MOG+20]. Attacks such as these violate the SGX, and indeed the processor, security model. Consequently, they have been mitigated with microcode patches from Intel [Int18, Int19]. Importantly, the kinds of memory-access and control-flow side channels described above are *not* considered to violate the SGX security model—SGX makes no claim to protect the addresses of accessed memory. Therefore, these side channels must be addressed in software, using the kinds of *fully oblivious algorithms* we propose later in Chapters 4, 5, and 6. Such side-channel attacks can be thwarted by designing the enclave program to be fully oblivious, as it eliminates these vulnerabilities at their introduction point. As more powerful attacks that amplify a TEE adversary's ability to violate the privacy guarantees of programs running in an enclave become efficient and practical, it becomes paramount to ensure that security and privacy critical components of such programs be implemented in a fully oblivious fashion.

2.3 TEE Applications

There have been several interesting academic and industry research directions and applications that leverage TEEs. One key benefit of using TEEs is that it facilitates collaborative privacy-preserving computations on sensitive data. Since programs running with these TEEs can have their own cryptographic keys, mutually distrusting parties can encrypt their data under such an enclave program's key. These parties can submit their data for only a priori agreed upon computation within this enclave, such that the data is never accessible to anyone but that program's computation, and no other computation other than the agreed upon one can be performed on it.

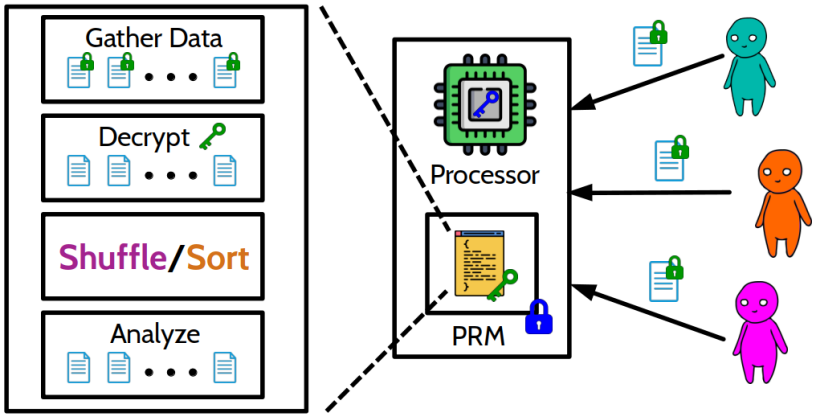


Figure 2.2: A common framework for privacy-preserving applications that leverage TEEs.

Such TEE applications spread over a wide range of domains. For instance, a few real-world applications include Prochlo [BEM+17], Google's privacy-preserving telemetry system, Opaque's privacy-preserving data analytics service [ZDB+17], or Signal's contact discovery protocol [Mox17]. Several other academic research directions have also proposed using TEEs for applications like privacy-preserving machine learning [SO19, KCY23], genomic data analysis [CWJ+16], private databases [KKS20], and oblivious file systems [DFD+21]. Notably, several such applications follow a very similar template like the one illustrated in Figure 2.2. For instance, in the context of privacy-preserving telemetry, Prochlo waits to gather sufficient user contributions, before shuffling all the contributions together to provide users an anonymity set of all the contributions for that round. The shuffled data, which no longer preserves the mapping of the users and their contributions, are then used for telemetry analysis.

A privacy critical component of such pipelines is in fact this shuffle (or depending on the applications sometimes a sort) step, which needs to be done obliviously within these TEEs, as otherwise the infrastructure host can undermine privacy through memory and control-flow side channels. Consequently oblivious shuffles and sorts form a very fundamental primitive for privacy-preserving computations via TEEs. Prior works often incorrectly assumed that either the accesses to the PRM do not leak memory access patterns [BEM+17, SO19], or that such access pattern leakages cannot be more granular than cacheline [GWC+23], or use bitonic sorting networks for such shuffling and sorting of data which incurs significant overheads and is often the bottleneck for such systems. In Chapter 4, we establish the notion of fully oblivious algorithms, and present new

oblivious shuffling and sorting algorithms in Chapters 5 and 6 that are resilient to all such software side-channels and are significantly more efficient than the prior fully oblivious state of the art.

Chapter 3

Metadata-Protecting Communications Systems (MPCS)

In order to tackle the problem of communications' metadata leakage, various forms of anonymous communication networks (ACNs) with varying degrees of metadata protections and design goals have been proposed. In practice users are limited to a few choices like Tor [DMS04] or I2P [ZH11]. These systems are designed to enable their users to protect their communications from a local eavesdropping network adversary like the user's Internet Service Provider (ISP). These systems have low latencies, but they are innately susceptible to traffic analysis attacks by a global passive network adversary. Moreover, recent attacks have surfaced that make a Tor user's Internet usage susceptible to privacy breaches by even local network adversaries [WCN+14, KAL+15, PLP+16]. Cwtch [Lew18] internally leverages Tor's v3 onion services protocol to facilitate peer-to-peer communications, and consequently inherits the privacy limitations of the Tor network, and similarly I2P's threat model too is limited to local adversaries.

Academically, peer-to-peer (P2P) based anonymous communication networks with varying privacy properties have been designed and studied [RP02, NW06, PRR09, MB09]. These systems do not provide the desired metadata-protecting guarantees since they too are limited in their privacy guarantees in the face of a strong global adversary. Furthermore, several attacks have been demonstrated against such P2P designs [TB06, SDH+10, WMB10, MB12]. Most of such early literature focuses on either designing P2P-based anonymous communication networks that circumvent some of Tor's shortcomings (such as its ability to scale, and the central points introduced in its design), or on improving DC-net and mixnet based constructions to be more efficient. Earlier surveys on anonymous communication [DDS09, SSA+18] well cover these aforementioned approaches;

these surveys focus on the routing-level challenges of anonymous communication networks. However over the last few years there has been a paradigm shift. Several recent metadata-protecting communication schemes use radically different approaches than those covered in prior surveys, to the extent that prior routing hurdles encountered do not apply anymore.

To understand the landscape better, we systematized 32 systems that aim to facilitate metadata-protecting communications even in the presence of global network adversaries. To contrast existing works and extract meaningful insights of the status quo, we provide two broad classifications of these works. The contributions of our systematiziation are:

1. We classify existing works into four different categories by the combination of type of metadata privacy guarantee and functionality they offer, and into six different families based on the core techniques that underpin these works. We then leverage these classifications to identify tradeoffs induced on such systems by their privacy guarantees, as well as the inherent limitations and benefits of the different families of constructions.

2. Existing works often target different goals. Consequently, they aspire to different properties and defend against different forms of attacks, resulting in a scattered view of the desirable properties and potential vulnerabilities of such systems. We contrast existing systems under a unified view of these different facets.

3. This unified view helps us identify potential vulnerabilities of existing works, as well as identify promising avenues for future research of metadata-protecting communication systems. Additionally, we also highlight subtle but impactful differences in the privacy implications of systems that leverage the same underlying assumption.

4. Deployable metadata-protecting messaging systems have been the aspirational goal of this line of research, and of this dissertation. We consider this application and highlight factors that hinder such systems today.

Methodology To gather all the works relevant to this survey, we started with Express [ECGZB21] as the seed paper. We then gathered all the works that were cited by Express, and those that cite Express using Google Scholar during November 2022. Among them, works that introduce a new system for metadata-protecting communications (as defined below) were selected while others where filtered out. This same process was

then repeated transitively for all selected papers, to finally arrive at the 32 works. Each of these works details a novel approach towards metadata-protecting communications.

3.1 Metadata-Protecting Communication Systems (MPCS)

In this work, we define a Metadata-Protecting Communication System (MPCS) to be any communication system designed with its threat model accounting for a global network adversary observing all communications on the network, and protecting (at least one form of) metadata of its users' communications. This definition of an MPCS is intentionally broad, and allows us to capture several different forms of MPCS designs. We focus on the global adversary threat model as that is the real-world adversary to combat today in the context of MPCS. We include systems that are secure even when a client's correspondent is adversarial and those that are not [ALT18]. We note that in this work, and in the MPCS literature, users are assumed to be able to connect to the MPCS. Users that are under censorship regimes that block access to MPCS will additionally need to leverage censorship circumvention tools, and that is an orthogonal research direction.

In systematizing existing MPCS, we identify two dimensions that help categorize existing MPCS, namely functionality and privacy goals. We first list the different types of functionality and privacy goals the systems we surveyed target.

Functionality Existing MPCS are designed to attain either *anonymous broadcast* or *end-to-end messaging (E2E)*. Anonymous broadcast protects the sender of a broadcasted message from network adversaries. End-to-end MPCS protect the conversation relationships of who is communicating with whom on the network, and possibly even the existence of a conversation.

Privacy Goals Existing works overload the terms 'private' or 'anonymous' with different meanings. Kuhn et al. [KBS+19] provide a taxonomy of the myriad privacy notions attainable in a communication network. We use their terminology to categorize privacy goals of existing systems into four, namely: (i) *Sender-Message Unlinkability* ($(SM)\bar{L}$: Sender-Message Un$\bar{L}$inkability) and (ii) *Receiver-Message Unlinkability* ($(RM)\bar{L}$) which aim to hide the metadata of who sent or received some message in a system, iii) *Relationship Unobservability* ($M\bar{O}[M\bar{L}]$ short for $M\bar{O}(SM\bar{L}, RM\bar{L})$: Message Un$\bar{O}$bservability with Sender-Message Un$\bar{L}$inkability and Receiver-Message Un$\bar{L}$inkability) aims to hide

the metadata of conversation relationships of users in the system, and iv) *Communication Unobservability* ($C\bar{O}$) hides the metadata of even the existence of a conversation.

Contrasting the 32 systems we surveyed fairly poses a challenge as they have different privacy and functionality goals. Hence we classify MPCS into four different categories based on their combined privacy and functionality goals,[1] such that systems within a category can be fairly compared. For example, comparing a broadcast scheme against an end-to-end scheme offers no insights. Similarly systems that align in functionality but differ in their privacy goals cannot be fairly compared either. The four categories are:

1. **Sender Unlinkable Messaging Systems (SUMS)** [$(SM)\bar{L}$, E2E]: These systems protect the sender and the message they submit. Typically, clients have a dedicated 'mailbox', and the sender's message is delivered to its destination mailbox without leaking which mailbox it was.

2. **Sender Unlinkable Broadcast Systems (SUBS)** [$(SM)\bar{L}$, Broadcast]: These are systems designed for metadata-protected broadcast, and typically operate over epochs. Messages are gathered during the course of an epoch. At the end of an epoch, the messages are published to a public bulletin board while hiding the mapping between senders and messages.

3. **Relationship Unobservable Systems (RUS)** [$M\bar{O}[M\bar{L}]$, E2E]: In a RUS, the global adversary can infer the set of senders and recipients, as well as the number of ongoing conversations, but they cannot identify any sender-receiver pair from this pool of active correspondents. We identify three broad designs for a RUS that start with a system with a weaker privacy goal, and then boost it to a RUS by enforcing communication epochs where first each client in the system sends a message followed by all clients retrieving a message.

    (a) **Mix Systems (Mix)** [$(SM)\bar{L}, (RM)\bar{L}$, E2E]: In these systems all incoming messages are mixed to break any sender-message correlations, before they arrive at their destination. From an adversary's perspective, the senders and receivers that participate in a round of communication are observable, but the sender-receiver pairs remain hidden.

[1] While there are eight permutations of functionality and privacy goals, only the four pairs we detail (plus one more that only appears in the RUS subdesigns) are logically sound. For instance, broadcast does not pair with relationship unobservability as there are no relationships to be observed for a broadcast.

(b) **Sender Unlinkable Messaging Systems (SUMS)** [$(SM)\bar{L}$, E2E]: SUMS (discussed earlier) can be converted into a RUS with the above generic transformation.

(c) **Receiver Unlinkable Messaging Systems (RUMS)** [$(RM)\bar{L}$, E2E]: In these systems, the sender and the (encrypted) message they submit in a round are linkable. However recipients retrieve messages without revealing which messages in this round were addressed to them.

4. **Communication Unobservable Systems (CUS)** [$C\bar{O}$]: These systems correspond to the highest level of metadata privacy attainable by Kuhn et al.'s taxonomy, as the global adversary is unaware of even the existence of a conversation in such systems. Typically CUS designs attain this high level of privacy by enforcing rigid requirements on clients. Existing CUS designs require all clients participating in such a system to be online at all times and perform predetermined conversation actions (like sending and receiving dummy messages) even if they are not currently in a conversation.

3.2 MPCS Properties

Informed by the works covered in our systematization, we present several system-level properties for MPCS, and later we use these properties to contrast existing schemes. We partition these properties into three categories, namely *Protections*, *Usability*, and *Performance*.

3.2.1 Protections

1. **(Server) Robust**: In this work we consider the property of server robustness as the ability of an MPCS to function correctly while tolerating server churn.[2]

2. **Anonymity Set (AS) Protection**: The anonymity set of honest clients participating in the system must not be reducible by malicious servers. This property ensures that an adversarial server cannot perform deanonymization attacks by controlling the anonymity set. Being able to manipulate the inclusion or exclusion of clients using an MPCS can place adversarial servers at a strong vantage point.

[2]Within our definition of robustness, we do not include fairness or guaranteed output notions of robustness from the secure multi-party computation (SMC) literature.

For instance, in the extreme case an adversarial server could perform an $n-1$ attack [SDS02] to deanonymize a user and their message in a sender-anonymous broadcast system. This ability to manipulate participation can also be used by servers for targeted blocking of clients, as captured by the notion of censorship resistance [APY20], or the 'audit attack' [NSSD22].

3. **DoS Resistance**: There are broadly two types of DoS attacks that MPCS aim to protect against, namely:

   (a) **Resource Exhaustion (RE)**: Like traditional DoS attacks, clients can attempt to overwhelm servers through Sybil messages, precluding honest clients from communicating.

   (b) **Disruption**: In some synchronous systems a client can potentially submit malformed messages that disrupt a communication round for all users. Alternatively in asynchronous systems, clients can craft malformed messages that disrupt the logical database that stores all client messages.

4. **Disconnection Impact**: MPCS often impose liveness assumptions on their users; i.e., they require the set of clients be the same throughout the operation of the system (until a reset at which point new clients can be enrolled), and that these clients never go offline. Alternatively, some constructions do not impose a liveness assumption, but participants incur privacy penalties when users go offline during a round of communication. We identify four possible outcomes for user disconnections (and present the symbols we later use to denote them): (i) system fails altogether (🚫), (ii) correspondent leaked (♉), (iii) anonymity set reduction (▼), and (iv) *temporary* anonymity set preservation (≈), i.e., even when a client disconnects for a small window of time they can still appear to send and receive messages by offloading this work to an untrusted server on their behalf.

3.2.2 Usability

1. **Setup**: The MPCS we cover do not require clients to have any IT knowledge or host servers to support metadata-protected communications, however some systems do have other prerequisites of clients. Namely, some systems require (i) clients exchange information (public keys, pseudorandom addresses, etc.) out of band (🤝), or (ii) out-of-band exchange, plus clients must participate in a dialing protocol (see Section 3.2.4) before they can send actual conversation data (📞).

2. **Parallel Conversations**: The ability of an E2E MPCS to support multiple simultaneous conversations for a user.

3. **Asynchronous**: Users in an asynchronous system can send messages to others even if the recipients are not currently online; such messages can then be retrieved the next time the recipients connect to the system.

4. **Low Latency**: To use an MPCS for messaging, it needs to have an *acceptable messaging latency*; i.e., when a user sends a message it should be delivered to their correspondent (or published) within a reasonable time frame. We set this acceptable latency to $10\,\text{s}$ for a system serving more than 10^5 clients, about two orders of magnitude greater than messaging latencies enjoyed by users today over standard non-metadata-protecting messengers (supporting billions of users). It typically takes few seconds for a user to type a message, so the idea of a $10\,\text{s}$ acceptable latency is that sending a message should not take too much longer than creating one for a seamless messaging experience.

5. **High Throughput**: In addition to having low latency, MPCS must be able to support high message throughput. We use messages per second as the unit for throughput, and consider a throughput of at least 100,000 messages per second as a system with high throughput. A practical MPCS should in fact be able to support a much higher throughput. The low choice of this threshold for high throughput was to fairly include more designs that show promise in terms of throughput as high-throughput systems.

3.2.3 Performance

1. **Horizontal Scalability**: The ability of a system to scale as more users join by adding more servers to the system. In the context of MPCS, the system needs to be able to scale while maintaining metadata protections without partitioning the anonymity set. Later, we will quantify the multiplicative factor of servers required to scale horizontally such that the work done per server remains the same when the system's user base grows by a factor of k.

2. **Client Overhead**: The asymptotic computational or communication overheads induced by the MPCS on its clients.

3. **Messages and Message/Compute Complexity**: The number of messages the MPCS can process in a logical round, and the corresponding message (or compute) complexity induced on its servers.

3.2.4 Dialing and Conversation Protocols

Several existing designs for E2E MPCS divide communication into two subprotocols, namely dialing and conversation protocols. Dialing protocols enable an online user Alice to express their desire to communicate with another user Bob. If Bob is online, and reciprocates this interest, such protocols result in Alice and Bob establishing a shared secret that they can leverage to facilitate their metadata-protected conversation. The dialing protocol itself must also be designed to hide metadata.

Once the shared secret is established by a dialing protocol, Alice and Bob converse using a conversation protocol. Conversation protocols typically require the correspondents to have a pre-established shared secret. Different systems from the literature suggest different frequencies of dialing to conversation rounds in the network; this ratio is typically system- and application-dependent.

The requirement of a dialing protocol is an artifact of designs that assume a synchronous communication model between users. In such works a pair of users can enjoy metadata-protected conversations only when they are both online and actively conversing, but users cannot send messages to an offline user for later retrieval. In several cases [VDHLZZ15, TGL+17, LGZ18, KLD20, EB22] it even hurts the privacy guarantees of a user if their correspondent goes offline during a conversation round, as we detail later in Section 3.4.5. If a metadata-protecting communication network is asynchronous by design then it removes the requirement of a dialing protocol to check if their participant is currently online, as we see later in Section 3.3.4.

3.3 Existing Work

Now we provide an overview of the 32 different MPCS that we surveyed. We broadly categorize these metadata-protecting communications systems into six different families based on the core cryptographic technique that underpins their security and privacy properties, namely DC-nets, mixnets, Private Information Retrieval (PIR), Reverse PIR, differential privacy (DP), and secure multi-party computation (SMC). For ease of exposition we discuss the various system designs by family here, later in Table 3.1 we present these systems partitioned by their category based on functionality and privacy guarantees as detailed in Section 3.1.

3.3.1 DC-net Based Systems

The *dining cryptographers problem* introduced by David Chaum [Cha88] poses the question of designing communication networks ingrained with sender and recipient unobservability, and introduced **DC-nets** (1988) as a solution. This original DC-net solution envisions a network with all clients connected to each other without any server involvement in the protocol; we hence call this an all-client DC-net. Participants communicate lockstep in a series of rounds, with each round split into as many slots as there are participants. Only one participant sends a real message in a given slot of a round, while the rest of the participants provide "cover" ciphertexts carefully crafted such that when any participant XORs all the ciphertexts received in a given slot, they end up with the message of that slot. While DC-nets guarantee perfect sender and recipient unobservability, they induce significant communication overheads to do so, as well as have several constraints that make them impractical to deploy.

For instance, DC-nets are prone to Denial-of-Service or disruptions, when more than one participant attempts to send a real message in a round. Disruptions are only feasible in certain designs of metadata-protecting schemes. In particular DC-net based and reverse PIR based schemes (that we discuss later in Section 3.3.5) are susceptible to such attacks, as these underlying techniques enable clients to influence the contents of other messages in the system. **Dissent** [CGF10] (2010) addresses the disruption problem by scheduling slots for its participants while preserving the anonymity of senders. They do so by introducing an accountable group shuffle protocol, the output of the shuffle determining the slot ordering of the round.[3]

Another fundamental problem with DC-nets is their inability to scale as they require an all-to-all communication network among all participating clients. To address this, Wolinsky et al. [WCGFJ12a] leverage the *anytrust* model in **D3** (2012), which converts the classical DC-net from a fully peer-to-peer network to a client-server model with n clients and m servers. In the anytrust model the clients need only trust that at least one server from the set of m servers behaves honestly.[4] The privacy guarantees are upheld even if the honest server is not the client's immediate upstream server. The anytrust model reduces the overheads imposed on a client, as the client outsources the bulk of its cover ciphertext generation to its upstream server. **Dissent in Numbers** [WCGFJ12b] (2012) (Dissent v2) leverages the anytrust assumption from D3 to enable the original Dissent protocol to support more clients.

[3] In Dissent, the disruption resistance only applies to the shuffle protocol, but not the communication protocol itself; i.e., clients misbehaving in the shuffle protocol can be detected and removed but Dissent cannot protect against clients disrupting the following communication rounds.

[4] This is the same trust model as Chaum's mix networks [Cha81].

While these systems that leverage the anytrust assumption are efficient, they are still susceptible to disruptions, where malicious clients deviate from the standard protocol and transmit arbitrary ciphertexts, which prevents revealing the plaintext for that round. Hence, D3 and Dissent v2 additionally introduce a retroactive blame protocol to detect disruptors in a group. However, they cannot prevent a disruption from happening in the first place. **Verdict** [CGWF13] (2013) attempts to prevent disruptions by introducing verifiable DC-nets. Clients in Verdict submit a proof of correctness to prove their contribution is well formed. This defence incurs significant overheads of verifying correctness proofs of all client messages.

3.3.2 Mixnet Based Systems

Riffle [KLDF16] (2016) uses the anytrust assumption as well, but instead of the servers facilitating a DC-net, they act as a mixnet. The servers form an ordered chain, and each server performs a verifiable shuffle on the set of messages it receives in a given round and passes it on to the next server; hence as long as one of the servers in this mixnet is honest no information about the ordering of messages is revealed to any malicious nodes after the honest node in the chain. At the end of a round the last server publishes all the shuffled messages thus providing sender anonymity, hence making Riffle a SUBS.

cMix [CDJ+17] (2017) proposes a fixed cascade of mix servers like Riffle with the same trust assumption. The key innovation in cMix is the separation of all the (expensive) public key operations into a precomputation phase, which allows for lighter computation during the online phase of the protocol. cMix can serve as a SUBS or a RUS as these shuffled messages can then be published, or forwarded to their intended recipient if they have a recipient identifier.

Leibowitz et al. introduced the Anonymous Post-Office Protocol, **AnonPOP** [GHL18] (2018), a synchronous mix-cascade network with Post Office (PO) and mix servers. In AnonPOP, each client has a mailbox that is maintained by one of the PO servers. Messages in AnonPOP are onion routed with authenticated encryption through a set of random mix servers. To retrieve messages, clients send a pull request (which contains a proof of ownership of the mailbox) to their mailbox PO through a mixchain, and the PO responds back with the first message in the client's mailbox. Their main contributions are techniques for bad-server isolation, and the notions of request pool and per-epoch mailboxes that enable limited disconnection of a client, without complete loss of their sender and recipient anonymity. However, the latter two mechanisms, while efficient defences, provide only heuristic security.[5]

[5]These defences are fragile. They assume honesty of a particular server (as opposed to any one server

Kwon et al.'s **Atom** [KCGDF17] (2017) is the first system in this line of work that has the explicit design goal of horizontal scalability. Atom is an anonymous broadcast protocol designed for 'short latency-tolerant' messages, enabling anonymous microblogging. The high-level idea of Atom is to have its servers arranged as a random permutation network and shuffle clients' messages as they pass through the network. Atom uses the square network by Håstad [Hås06], which can permute m elements using $\sqrt{m}$ nodes with each shuffling $\sqrt{m}$ ciphertexts and connects to $\sqrt{m}$ nodes in the subsequent layer.[6]

In Atom each logical server in this network is realized by a group of physical servers such that the anytrust assumption holds over the group. The number of servers in each group is tuned to ensure that given the total number of servers, and knowledge that up to a bounded fraction of them can be malicious, at least one server in each group should be honest with very high probability. To send a message, a user picks an entry group and encrypts their message to the public keys of all servers in that group, and then sends it to all servers in that entry group. Once enough ciphertexts are available to an entry group, each server in a group shuffles the ciphertexts and passes them on to the next server. The last server after shuffling divides the permuted ciphertexts into $\sqrt{m}$ batches, and passes all the batches back to the first server. Each server then leverages the out-of-order reencryption property of El Gamal to partially decrypt their portion from each batch and reencrypt the ciphertexts to the next group that this batch is destined for. The last group in the network simply makes the plaintexts available, completing the anonymous broadcast functionality.

In order to protect against malicious servers that could tamper with client messages in Atom, Kwon et al. propose two techniques i) using non-interactive zero knowledge (NIZK) proofs and ii) using client-inserted trap messages (which turns out to be significantly faster than using NIZK proofs in practice, but at the expense of doubling the number of messages in the system). Finally, in order to prevent the system from being blocked due to server churn they also propose a fault-tolerant version of their protocol by leveraging 'manytrust' groups, which are anytrust groups with at least h honest servers in each group, such that each group can tolerate up to $h - 1$ failures by replacing the group's public key with the key of a threshold cryptosystem.

XRD [KLD20] (2020) (short for Crossroads) further reduced the overheads of shuffling in such mixnet based protocols. XRD aims to support relationship-unobservable communications and hence is by design a RUS, unlike the other mixnets schemes dis-

of a pool like anytrust), and the defence fails if that server is adversarial. This leads us to use the term heuristic security.

[6] This network only requires constant iterations of mixing to produce a near-uniform random permutation, and in their work they use 10 iterations of mixing.

cussed so far that are SUBS that can be extended to a RUS. Each client in XRD has a unique mailbox associated with them. Servers in the system are organized into mix chains, and in each round each user sends a message along ℓ chains. The system is designed such that all pairs of users have at least one chain in common, and the choice of chains themselves are publicly computable; in order for this to operate correctly given q chains, clients have to choose $\ell \approx \sqrt{q}$ chains to ensure that they have an intersecting chain with every other user. The high-level idea is that users not communicating send ℓ 'loopback messages' in a round which return to their mailbox, while if two users Alice and Bob are communicating in a given round they replace their common chain with messages for each other instead of the loopback message. This results in an adversary always observing each active user's mailbox receiving ℓ messages in each round, and they cannot determine whether any two individuals are actively communicating or not.

The two immediate downsides that arise from this network design based on chain intersections is that given n users in the system and m servers, clients effectively have to send $O(\sqrt{m})$ messages. Each server also consequently has to process $O(n/\sqrt{m})$ messages as opposed to $O(n/m)$ by other horizontally scalable designs. Additionally, XRD does not provide (or recommend) a dialing protocol for their system, and none of the currently existing dialing protocols are appropriate for XRD. However, XRD introduces a novel technique 'aggregate hybrid shuffle' (AHS) to circumvent having to perform expensive verifiable shuffles, and hence has lower latencies for each round of communication.

Loopix [PHE+17] (2017) takes a significantly different approach. Other than mix servers and users, Loopix includes special service provider nodes. The service provider nodes act as entry and exit points of the network (bookending the mix servers), and clients interface with the Loopix system through these nodes. The network itself consists of mix nodes that are organized in a stratified network topology with shallow depth, with each node connected to all the other nodes from its next and previous layer.[7] The high-level idea is that users send each other messages by routing their messages through this network with source-picked delays from a Poisson distribution at each of the mix nodes. Mixes and users also generate cover traffic using a Poisson distribution to thwart passive and active attacks; consequently the entire system can be viewed as a large Poisson mixing process, providing it with formal guarantees on its message delivery and privacy. The service provider plays a semi-trusted role and maintains clients' mailboxes. If a client is offline, their service provider stores the messages destined for that client, enabling clients to retrieve messages even after they go offline, making it the only mixnet design with support for asynchrony.

Unfortunately, Loopix relies on trusting its clients' service provider nodes. A mali-

[7] In its experiments, Loopix uses a network depth of three.

cious service provider node can violate receiver anonymity, as they can observe receivers' interactions with their mailbox, and can infer metadata about the messages received. Finally, while Loopix implicitly leverages the manytrust assumption, its design has concerning privacy implications when contrasted with other systems that leverage the same assumption, as we detail later in Section 3.4.3.

Trellis [LSSD22] (2022) introduces an anonymous broadcast system that is robust to server churn. Their work, however, assumes a synchronous communication network and user liveness (i.e., all users are online for all rounds of communication). Similar in flavour to cMix, the high-level idea of Trellis is to lift all the heavyweight public key cryptography operations into a one-time path establishment phase, followed by lightweight message transmission rounds.

In Trellis, Langowski et al. introduce two novel cryptographic building blocks, namely Anonymous Routing Tokens (ART) and Boomerang Encryption (BE). Similar to Atom, Trellis leverages a permutation network to instantiate the mix network. ART enables a user to generate a publicly verifiable random next server for routing their message through the permutation network. Given a chain of servers to onion encrypt a message for (dictated by the ART at each layer), BE onion encrypts a message in the reverse path (of the onion encryption chain), and then onion encrypts this in the forward direction. The resulting ciphertext serves as a proof of message delivery in onion routing schemes, when the message in the reverse direction is set to be a random nonce, as this guarantees that every server along the path decrypted onion layers correctly and forwarded it to the next server. The interplay of ART and BE enables honest servers to blame deviant servers if messages for an ART it is responsible for is missing or faulty, during both path establishment as well as message broadcast rounds. In order to recover from such faults, similar to Atom, Trellis leverages proactive secret sharing to distribute the secret key of each server in the permutation network across an anytrust group, thus enabling a substitute server to recover the state of a faulty (or malicious) server, and resume operations in the event of a blame protocol.

Finally, **Boomerang** [JWC+23] (2023) is a recent system that fits in this family; however, we defer its discussion to the next section because it is closely built on another system that will be described therein.

3.3.3 Differential Privacy Based Systems

These systems start with a mixnet design and further enhance them by leveraging differential privacy (DP) [DMNS06]. However, using DP implies their privacy guarantees

are significantly different from those of mixnets (as we detail later in Section 3.4.1), so we treat them as their own family.

Vuvuzela [VDHLZZ15] (2015) introduced this approach of leveraging DP to noise the adversary-observable variables of a communication network. This allows them to claim via DP that any set of observations made given a user's real action is almost as likely as what would be observed given any other plausible cover story. They apply this DP principle to build a metadata-protected messaging system using two protocols: i) a point-to-point conversation protocol, and ii) a dialing protocol to initiate the conversation.

Vuvuzela consists of a single chain of servers under an anytrust assumption, which clients communicate through. In Vuvuzela, users communicate through virtual locations called *dead drops*. Two conversing users send their messages to the same dead drop. Conversing clients are assumed to know each other's long-term public key a priori, and the dead drop location is computed for a given round via a shared secret derived from their public keys and the current round number. Clients send onion-encrypted messages through the server chain, to have their message delivered to their desired dead drop at the last server in the chain. In order to achieve privacy, each server in the chain after peeling its layer of encryption shuffles all the messages it receives and forwards it to the next. After all the messages have arrived at the dead drop (if two arrive at the same dead drop, they get exchanged), the messages make their way back through the system in reverse order with each server reversing the shuffle they performed.

In order to hide the true number of conversations happening in the system, all users irrespective of if they are actually conversing or not send a message. Users not participating in a conversation send messages to a random dead drop. Additionally each server in the chain also injects two forms of cover traffic: single-message and pair-message dead drop cover traffic, which hides the true count of conversing and non-conversing users in the system.

Since Vuvuzela is throttled in scalability by the number of messages each server has to process as the number of users in the system grow, **Stadium** [TGL+17] (2017) was designed with the goal of horizontal scalability. All servers add noise messages collaboratively at the start of a round into the system, and throughout the duration of the round they get processed along with users' messages by verifiable processing techniques. In Stadium, the system comprises two types of mix chains, input chains and output chains, with the same anytrust assumption held true for each individual chain. Two users conversing with each other choose a random input chain, but the same output chain as their messages are destined for the same dead drop. In order to reduce the overheads of the verifiable shuffle they extend the hybrid verifiable shuffling techniques from Rif-

fle [KLDF16], and separate verifiable shuffling of large messages onto shuffle proofs over smaller AEAD keys.

The deviation from Vuvuzela's design then arises from Stadium's phase of distributing messages from input to output chains. In order to prevent malicious servers from tampering with users' messages, servers in an input chain collaborate to compute a hash of the messages intended for each output chain and sends the hash to every server of that output chain. If every server in the input chain acts honestly, these hashes will be equivalent, and the distribution phase would have been correctly executed.

Karaoke [LGZ18] (2018) further refines the horizontal scalability by introducing two key techniques, namely efficient noise verification and optimistic indistinguishability. In Karaoke, Lazar et al. point out that instead of performing verifiable operations, it is sufficient to verify that all noise messages inserted into the system still exist in the system at the end of each round. Towards this end they propose using Bloom filters [Blo70]; each server at each layer in Karaoke's topology computes a Bloom filter of all the messages it has received and sends this to all other servers. The other servers then verify whether the noise messages they generated appear in the Bloom filter. If any server indicates that their noise has been lost, the round is stopped. This circumvents the overheads of verifiable shuffle in exchange for efficient hash operations.

Instead of each user sending and receiving one message like in Vuvuzela and Stadium, Lazar et al. observe that if users send and receive two messages in a round they can appear to be in a conversation. In this setting, when the adversary is passive or there are no network outages, the number of dead drop accesses reveals no metadata about the communication of any pair of users, since for a pair of users either idling or chatting, there will be two dead drops both of which are accessed twice. If messages are lost, however, the adversary may observe a dead drop with a single access, which reveals some information. Karaoke addresses this through the addition of noise messages, as in the prior systems. Interestingly, message loss is detectable by the user in Karaoke since they can see the messages received back from the server. Hence, in Karaoke, Lazar et al. make the distinction of "leakage-free" rounds for rounds where no message loss is incurred. Karaoke's leakage-free rounds allow them to improve performance by tuning the amount of noise messages injected depending on whether the system is currently experiencing message loss or not.

Barman et al. introduced **Groove** [BKL+22] (2022) which further refines DP-based systems with three main user flexibility options. The first is oblivious delegations, which enables a user to leverage an untrusted service provider to participate in the typically rigid mechanisms of a CUS; i.e., users can preemptively provide servers with their messages for some number of future rounds of communications, which enables these service

providers to receive messages from the users' correspondents while they go offline. Upon reconnection, users can retrieve just the messages of interest by a shuffle-and-select protocol that hides which of the messages were retrieved. Second, Groove enables users to converse with multiple correspondents (a system constant of 50) in parallel, and finally it enables multi-device usage for clients so they can use Groove over multiple devices, without any privacy leakages that may arise from lack of synchronization between these devices.

Finally, **Boomerang** [JWC+23] (2023) presents a new mixnet style of CUS that leverages TEEs, but as we mentioned earlier in Section 3.3.2 we introduce it here since it is easier to explain in the context of Stadium. Where traditional mixnets used different nodes hosted by different entities and leveraged the anytrust assumption, in Boomerang all the nodes of the system are individual TEE-aided servers that can be hosted by the same entity. Moreover, it does away with differential privacy altogether making it a mixnet design based on cryptographic security under the TEE assumption.

Boomerang is most similar to Stadium; by leveraging TEEs the system avoids multiple servers on each of the input and output chains that have to shuffle and inject DP noise messages, and replaces them with an entry node and a B-node respectively. The entry nodes create equal-sized batches of messages for each of the B-nodes obliviously. The batch size is upper bounded to ensure minimal padding overheads while maintaining a low overflow failure probability. Since the batch sizes are uniform across all the B-nodes for each entry server, the system no longer has to inject DP noise messages. Once the B-nodes receive all the messages (including padding) for a round, they obliviously sort all the messages by their labels and perform a linear scan swapping messages with the same label. Finally, the B-nodes sort the messages by the entry node ID they came from, and send all the packets back in the reverse direction. Boomerang additionally introduces oblivious algorithms to detect any irregular label patterns, i.e., a single message with a unique private label, or more than two messages with the same private label, at both the entry and B-nodes, and returns such irregular messages as is to the sender without disrupting other messages in the system.

3.3.4 Private-Read Based Systems

We classify PIR-based systems into private-read and private-write based systems. Private-read systems leverage PIR in the standard fashion. Private-write systems use 'PIR in reverse' [CGBM15] instead, and hence we call this technique Reverse PIR (RPIR). The high-level operation of a PIR-based MPCS is that arriving messages are placed into a

data structure, and receivers use PIR to privately read that data structure without revealing which messages were read.

Pynchon Gate [SCM05] (2005) is one of the earliest works that used PIR to design private communications; their work proposes a practical pseudonymous mail retrieval system. Their system consists of three types of nodes: a nym server, a collator, and distributors. The nym server is the public-facing end of Pynchon Gate that sends and receives pseudonymous email. On its own the nym server does not provide sender anonymity but clients can leverage mix networks [Cha81] for sender anonymity. The nym server collects mails for all the pseudonyms and at the end of every 'cycle' passes these messages to the collator. The collator organizes the mails into a structure that allows clients to query for their mails. The collator then distributes this structure across the distributor nodes, and the clients use Chor's ℓ-server PIR protocol [CGKS95] to retrieve their mail.

Pung [AS16] (2016) proposed the first PIR-based MPCS and indeed the first MPCS at all since the original (disruption-prone) all-clients DC-net that does not rely on an assumption about non-colluding servers. They treat the problem as a metadata-protected key-value store, and enable clients to privately retrieve messages from this key-value store by leveraging the computational PIR (CPIR) scheme, XPIR [AMBFK16]. In each round, clients send and retrieve exactly one message. The key for each message is a random label generated by a shared secret (established via sharing public keys through an out-of-band channel), and the value is the encrypted message. At the end of the round, the Pung server organizes the received messages into a binary search tree, and clients perform an oblivious search by labels to retrieve messages of interest to them.

Angel et al. [ACLS18] improved the efficiency of CPIR schemes with **SealPIR** (2018), by i) reducing the network bandwidth overheads of CPIR schemes by a novel query compression technique which reduces the query size from $O(n)$ ciphertexts to $O(1)$ for an n-record PIR database and ii) a data-encoding scheme for multi-query PIR schemes that allows it to amortize the cost of processing a batch of requests from a single client. They port these techniques to Pung to further reduce the network costs and throughput of Pung.

Talek [CSM+20] (2020) introduces a private *group messaging* system that aims to provide privacy for its users via access sequence indistinguishability. Talek's core abstraction is a private log, that enables a single writer to share messages with many readers. In order to convert this to a group messaging protocol, each member subscribes to the logs of all other group members.

The log itself is hosted by ℓ non-colluding ITPIR servers, and leverages a blocked cuckoo hash table [PR04] to improve efficiency. In Talek, clients issue writes to pseudo-

random locations on each server, where the locations are determined by applying a PRF to a secret log handle that is shared between the log's writer and readers. The cuckoo hash table is parameterized to have b buckets, each with d messages in them, which dictates the maximum capacity of messages n in the system to be $b \times d$. When new messages enter the system after this maximum capacity, the oldest entries are evicted from the bucket they get assigned to. Hence, unlike other metadata-protecting communication systems Talek has to be parameterized first to set this value of n, b, and d to trade off the time-to-live (TTL) for each message in the system with the latency of each PIR request.

Their other core contribution is a private notification system via Bloom filters [Blo70], which allow users to check which of their subscribed logs have updates with significantly lower overheads than performing a PIR request. This allows users to poll their subscribed logs periodically and perform PIR requests efficiently.

3.3.5 Private-Write Based Systems

In contrast to systems from the previous section, instead of leveraging PIR to retrieve a message from a collection of messages privately, RPIR systems use PIR to *insert* a message into a collection privately. Corrigan-Gibbs et al. introduced this idea with **Riposte** [CGBM15] (2015), a novel design for anonymous broadcast, allowing clients to anonymously post messages to a shared 'bulletin board', maintained by a small set of non-colluding servers. Riposte effectively implements a write-private and disruption-resistant database scheme.

In Riposte's efficient variant, two of the three servers act as ITPIR servers that the client issues requests to, while the third is an 'audit' server that participates in an MPC protocol to ensure clients submit well-formed requests. The system operates in epochs, and at the end of every epoch the Riposte servers publish all the write requests they received during an epoch, providing each honest client an anonymity set of honest clients that participated in that epoch. The two servers together hold a logical database, and each client aims to anonymously drop messages into a slot in this database.

Since clients drop messages into a random slot via private writes, collisions could render messages unreadable. Corrigan-Gibbs et al. address this issue in two ways: i) tuning the size of the database table to be large enough to accommodate the expected number of write requests for a given success rate, and ii) introducing a novel technique to recover from collisions at the expense of storage. Given the expected number of messages in an epoch, these two techniques can be tuned to set a database size that imposes the

least overheads for the PIR scheme. Finally, in order to improve the bandwidth efficiency of the PIR scheme, they propose using Distributed Point Functions [GI14] (DPFs) to reduce the asymptotic overhead of client requests from $O(L)$ to $O(\sqrt{L})$, where L is the number of entries in the database (set by the tuning described earlier).

Express [ECGZB21] (2021) details a metadata-protecting messaging system that only incurs constant-factor overheads on the client side regardless of the number of users. Express is a natural extension of Riposte, and similar to Riposte leverages a non-collusion assumption among the two servers of the system. Unlike Riposte, in Express all participating recipients have their own individual mailbox. Hence Express does not operate in epochs, since messages can only be read by their recipient, making it the first asynchronous end-to-end metadata-protecting communication system that leverages PIR.

Express leverages a more efficient DPF scheme [BGI16], and uses Secret-shared Non-Interactive Proofs (SNIP) [CGB17, BBC+19] instead of the more expensive zero knowledge proof that was used in Riposte for auditing the client-submitted DPFs. The former reduces their client-side overhead to logarithmic in the number of mailboxes, instead of square root. However, in addition they propose using a virtual address space of 2^λ for mailboxes, and senders need to know their recipient's virtual and physical mailbox address in order to send messages, making the client overhead $O(\lambda)$. This virtual address trick enables Express to prevent targeted disruption attacks.

More recently, Vadapalli et al.'s **Sabre** [VSH22] (2022) illustrated how both Riposte and Express are susceptible to resource exhaustion (RE) attacks. Although they both defend against ill-formed DPFs, an RE attacker can still submit well-formed DPFs that write to a non-existent mailbox resulting in the server having to perform expensive $O(n)$ operations for an effective no operation, yielding a DoS attack with an exponential amplification factor as the client work is $O(\lambda \cdot \log n)$ lightweight operations. In the process of defending against this form of resource exhaustion DoS attack, Sabre also succeeds at fundamentally speeding up the message delivery protocol.

The improvement arises from Vadapalli et al.'s observation that mailbox address verification (i.e., verifying that a client is writing a message to a valid mailbox in the system) can be performed separately from message delivery. This allows, for n users, the use of smaller $O(\log n)$ instead of $O(\lambda)$ sized DPF keys that Express uses for message delivery (this is better since $\log n \ll \lambda$). However, one still needs to perform mailbox address verification, and to this end Vadapalli et al. provide two novel protocols, the more efficient one of which is an $O(\lambda)$ protocol. Only if that check passes, the audit protocol then ensures the well-formedness of the DPF, for which Sabre has another innovation enabling servers to only perform $O(\lambda \cdot \log n)$ work, thus removing the DoS amplification factor,

unlike Express where the server has to perform $O(\lambda \cdot n)$ work. In concrete performance their experiments show that their optimizations result in a significant multiplicative factor improvement under "good" write conditions (i.e., no malicious DoS attacker writes), with this performance gap widening up to one to two orders of magnitude as the ratio of bad to good writes increases.

Newman et al.'s **Spectrum** [NSSD22] (2022) solves a closely related yet slightly different problem of high-bandwidth anonymous broadcast. Their setting considers a small number of broadcasters that wish to share documents or large files with many subscribers by leveraging two or more non-colluding servers. Unlike the previous systems discussed so far, by limiting the number of broadcasters to L out of n participants (broadcasters + subscribers) they succeed at reducing the server-side computations to $O(L)$ per message write, while still providing these L broadcasters an anonymity set of all n participants. The trick lies in allocating one 'channel' (synonymous to mailboxes from Riposte and Express) to each of these L broadcasters, with the broadcaster having a credential that allows them to publish to this channel. The messages to the channels leverage the same DPF-based technique discussed for Riposte and Express, with the only difference being the share of one's credential being sent along with the DPF key share to each server. The subscribers too can submit message writes which are intended for any of the channels, however these writes are strictly zero entries.

Additionally, Newman et al. also point out that Riposte, Express, (and Sabre) are susceptible to malicious servers selectively dropping valid client messages to perform selective deanonymization attacks. To address this they design BlameGame, an audit protocol that helps the honest servers detect if any of the other servers are deviating from their protocol, by having clients send their message shares encrypted under a verifiable encryption scheme to each server and having all servers commit to the message share they used, allowing the servers to implicate a malicious client or server.

3.3.6 Secure Multiparty Computation (SMC) Based Systems

MCMix [AKTZ17] (2017) recasts the problem of metadata-protected communication as a SMC consisting of dialing and conversation functionalities. The system operates in rounds; at the end of each conversation round conversing users have the messages they sent into the system swapped. To do so, users interested in conversing establish their intent via the dialing protocol, which allows them to establish a dead drop location for the next (possibly several) communication rounds. This dead drop location is a 64-bit address that can be generated by both participants by a hash function via their shared secret and the current conversation round. Once all messages for a communication round

have been gathered, the servers perform an SMC protocol that obliviously sorts all the messages by the dead drop addresses and swaps adjacent messages that are destined for the same address. This oblivious sort is in fact the crux and bottleneck of their protocol, and they leverage the "shuffle before sort" paradigm from Hamada et al. [HKI+13], which results in an overarching $O(n \log n)$ complexity for both their conversation and dialing protocols when there are n participants.

Clarion [EB22] (2022) designs an improved three-party malicious secure shuffle protocol, and shows how it can serve as an anonymous broadcast channel by having the servers publish the secret-shared shuffled messages at the end of the shuffle. Clarion details a simple mechanism to use such an anonymous broadcast channel as a drop-in replacement for the conversation protocol in MCMix. Instead of having conversation participants send messages to the same dead drop address, they now send messages addressed with two unique dead drop locations derived from the same shared secret, one for each direction of the conversation. At the end of the broadcast, each client retrieves the location their correspondent would have written to, and clients that are not in an active conversation retrieve the same random location they wrote to. Since every client still always sends and retrieves a single message identified by a random address tag, the behaviour of clients having a conversation remains indistinguishable from the other clients. Clarion's improvement stems from the fact that the new secret-shared shuffle protocols are $O(n)$ protocols, saving a $\log n$ factor over the oblivious sort underlying MCMix.

Lu et al. present **Asynchromix** and **PowerMix** [LYK+19] (2019). They propose techniques to provide robustness guarantees to SMC protocols, without incurring the additional overheads of malicious-secure SMC protocols. They aim to support fairness and guaranteed output, in addition to the notion of robustness we consider in Section 3.2. The high-level idea for both designs is to have each server in the system obtain a share of each client's message, mix the message shares, and reconstruct the messages once the mixing has disassociated the clients and their messages. To receive valid message shares from clients without the overheads induced by Verifiable Secret Sharing [RBO89], Lu et al. propose clients blind their message before broadcasting it to the servers. The blinding value is provided by the servers; each server provides a share of a random blinding value to the client, and clients reconstruct and blind their messages with it. Once the client broadcasts their blinded message, the servers reproduce shares of the original message by removing their share of the blind from it. Asynchromix [LYK+19] has $O(\log^2 n)$ round complexity as it leverages $O(\log n)$ rounds of iterated butterfly networks (which each have depth $O(\log n)$) towards mixing the messages. This results in a protocol with $O(mn \log^2 n)$ communication and computation costs. Alternatively, Pow-

erMix [LYK+19] only has 2 rounds of communication, but incurs significantly more computation. PowerMix leverages Newton's sums instead of a shuffle network. In order to mix a batch of n messages, the servers compute the powers of each share of the message from 1 to n and then compute the sums of each power. This can be computed with a single round of communication by using precomputed powers of random secret-shared values. The servers then reconstruct the sums of each power publicly, and then solve for the set of messages.

Blinder [APY20] (2020) leverages SMC towards enabling robust anonymous broadcast. Blinder aims to limit the ability of malicious servers to block honest clients from submitting their messages, and refers to this property as censorship resistance, effectively preventing the system from reducing the anonymity set offered by honest clients. Blinder is based on Shamir secret sharing [Sha79], and leverages the recent observation of an efficient 'sum of products' gate [CGH+18]. Blinder reuses the server-side storage layout and collision handling techniques from Riposte. Clients compress their queries in a fashion similar to recursive PIR [CGKS95] to reduce the communication overheads, and the servers perform a format verification on the compressed query. Censorship resistance is achieved via a robust input sharing protocol that prevents malicious servers from discarding (more than a small number of) honest client messages.

RPM [LK22] (2022) proposes three schemes for anonymous broadcast. Their idea is to generate a permutation matrix to shuffle the client messages in a round. Each server picks its own permutation matrix, and secret shares it with the rest. All servers multiply these shared matrices to get the final permutation matrix. The first scheme requires an inner product SMC computation in the online phase, and the second removes this by shifting it into an offline phase. Both variants require constructing and multiplying n-by-n matrices for n client messages in the offline phase, which is prohibitive for large n. The third variant repurposes the first two schemes as building blocks, by using them as the nodes of Håstad's permutation network [Hås06], resulting in offline creation and multiplication of a constant number of permutation matrices of size $\sqrt{n}$-by-$\sqrt{n}$. The first two variants are limited in scalability as they incur complexity cubic in the number of clients, while the third variant brings down the online computation complexity to $O(n^{1.5})$.

3.4 Discussion

We succinctly summarize all the works we detailed in Section 3.3 into Table 3.1, partitioned by the type of metadata privacy guarantee these systems achieve, and highlighting the properties from Section 3.2. The table also contrasts each scheme's asymptotic communication and computation complexities, and any dialing or offline computation overheads induced as well. We refrain from comparing concrete performance numbers here as each proposed system uses their own experimental setup for evaluating their prototype implementations. Appendix A provides details on the performance numbers reported by these works for a better sense on practicality of these schemes, which is not immediate from the underlying complexities.

System	Year	Type	Family	Assumption	Robust	AS Protection	RE Resistant	Disruption Resistant	Disconnection Impact	Setup	Parallel Conversations	Asynchronous	Low Latency	High Throughput	Horizontally Scalable	Audit Complexity	Client Overhead	Message Compute [if different] Complexity	Messages
					Protections					Usability						Performance			
Vuvuzela [VDHLZZ15]	2015	CUS (Mix)	DP	$(1:m)$-S: DP	○	○	●	●	▼	📞	○	○	○	○	○	×	m	$n \cdot m + n \cdot m$	n
Pung [AS16]	2016	CUS (RUMS)	PIR	RLWE	○	○	●	●	▼	📞	●	○	●	○	k^2	×	n	$n^2 + n$	1
Stadium [TGL+17]	2017	CUS (Mix)	DP	f-S: DP	○	○	●	●	▼	📞	○	○	○	●	k	×	ℓ	$(n + m_d) \cdot m_d + n \cdot \ell + m^3$	n
MCMix [AKTZ17]	2017	CUS	SMC	$(1:3)$-S	○	○	●	●	▼	📞	○	○	●	○	○	×	1	$n \cdot \log n + n \cdot \log n$	n
AnonPOP [GHL18]	2018	CUS (Mix)	M	f-S: DP	○	○	●	●	≈	🤫	●	○	○	○	k	×	ℓ	$n \cdot \ell$	n
SealPIR [ACLS18]	2018	CUS (RUMS)	PIR	RLWE	○	○	●	●	▼	📞	●	○	●	○	k^2	×	n	$n^2 + 1$ $[n+n]$	1
Karaoke [LGZ18]	2018	CUS (Mix)	DP	f-S: DP	○	○	●	●	▼	📞	○	○	●	●	k	×	ℓ	$(n + m_d) \cdot m_d + n \cdot \ell + m^3$	n
XRD [KLD20]	2020	CUS	M	f-S	○	○	●	●	▼	📞	◐	○	○	○	k^2	×	$\sqrt{m}$	$n^2 + n \cdot \sqrt{m}$	1
Clarion [EB22]	2022	CUS	SMC	$(2:3)$-MA	○	○	●	●	♉	📞	○	○	○	○	○	×	1	$n \cdot \log n + n$	n
Groove [BKL+22]	2022	CUS (Mix)	DP	f-S: DP	○	○	●	●	≈	📞	●	○	●	●	k	×	ℓ	$(n + m_d) \cdot m_d + n \cdot \ell + m^3$	n
Boomerang [JWC+23]	2023	CUS	M	TEE	○	○	●	●	▼	📞	○	○	●	●	k	×	1	$(n + m_d) \cdot m_d + n$ $[n \cdot \log^2(n/m)]$	n
DC-nets [Cha88]	1988	CUS/SUBS	DC	None	○	●	●	○	🚫	🤫	○	○	○	○	○	×	n	$n^2[n]$	1
Dissent [CGF10]	2010	CUS/SUBS	DC	None	○	○	●	○	🚫	🤫	○	○	○	○	○	×	n	$n^2 + n^2$ $[n^2 + n]$	1
D3 [WCGFJ12a]	2012	CUS/SUBS	DC	$(1:m)$-S	○	○	●	◐	▼	🤫	○	○	○	○	○	×	$n^2 \cdot m$	m	$n \cdot m + n \cdot m$	1
Dissentv2 [WCGFJ12b]	2012	CUS/SUBS	DC	$(1:m)$-S	○	○	●	◐	▼	🤫	○	○	○	○	○	×	$n^2 \cdot m$	m	$n \cdot m + n \cdot m$	1
Verdict [CGWF13]	2013	CUS/SUBS	DC	$(1:m)$-S	○	○	●	●	▼	🤫	○	○	○	○	○	×	$n^2 \cdot m$	m	$n \cdot m + n \cdot m$	1
Riposte [CGBM15]	2015	SUBS	RPIR	$(2:3)$-S	○	○	○	●	▼	×	🚫	●	●	○	k^2	×	n	$\sqrt{n}$	$\sqrt{n}$ $[n]$	1
Riffle [KLDF16]	2016	SUBS	M	$(1:m)$-S	○	○	●	●	▼	×	🚫	●	○	○	○	×	m	$n \cdot m$	$n \cdot m$	n
cMix [CDJ+17]	2017	SUBS	M	$(1:m)$-S	○	○	●	●	▼	×	🚫	○	○	○	○	×	m	$n \cdot m$	$n \cdot m$	n
Atom [KCGDF17]	2017	SUBS	M	f-S	●	◐	●	●	▼	×	🚫	●	○	○	k	×	1	$n \cdot m$	$n \cdot m$	n
AsynchroMix [LYK+19]	2019	SUBS	SMC	$(2n/3:n)$-M	◐	●	●	●	▼	×	🚫	●	○	○	○	×	m	$m \cdot n^2 \cdot \log^2 n + m \cdot n \cdot \log^2 n$	$m \cdot n^2 \cdot \log^2 n + m \cdot n \cdot \log^2 n$	n
PowerMix [LYK+19]	2019	SUBS	SMC	$(2n/3:n)$-M	◐	●	●	●	▼	×	🚫	●	○	○	○	×	m	$n^2 + m \cdot n$ $[n^2 + n^3 + m]$	$n^2 + m \cdot n$ $[n^2 + n^3 + m]$	n
Blinder [APY20]	2020	SUBS	SMC	$(3n/4:n)$-M	●	●	●	●	▼	×	🚫	●	○	○	○	$m \cdot \sqrt{n}m$	$\cdot \sqrt{n}m \cdot n \cdot \log n + m \cdot n$ $[m \cdot n \cdot \log n + n^2]$	$m \cdot n \cdot \log n + m \cdot n$ $[m \cdot n \cdot \log n + n^2]$	n
Spectrum [NSSD22]	2022	SUBS	RPIR	$(1:m)$-S	○	●	●	●	▼	×	🚫	●	○	○	○	1	m	m $[m \cdot b]$	m $[m \cdot b]$	1
Trellis [LSSD22]	2022	SUBS	M	f-M	●	●	●	●	🚫	×	🚫	○	○	○	k	×	$\log m$	$n \cdot \log^2 m + n \cdot \log m$ $[n \cdot m]$	$n \cdot \log^2 m + n \cdot \log m$ $[n \cdot m]$	n
RPM [LK22]	2022	SUBS	SMC	$(3n/4:n)$-M	●	●	●	●	▼	×	🚫	●	○	○	○	×	m	$m \cdot n^{1.5} + n$ $[m \cdot n^2 + n^{1.5}]$	$m \cdot n^{1.5} + n$ $[m \cdot n^2 + n^{1.5}]$	n
Express [ECGZB21]	2021	SUMS	RPIR	$(1:2)$-S	○	○	○	●	▼	🤫	●	●	●	○	k^2	n	λ	1 $[n]$	1 $[n]$	1
Sabre [VSH22]	2022	SUMS/SUBS	RPIR	$(1:2)$-S	○	○	●	●	▼	🤫	●	●	●	○	k^2	$\log n$	$\log n$	1 $[n]$	1 $[n]$	1
Mixnets [Cha81]	1981	RUS (Mix)	M	$(1:m)$-S	○	○	●	●	🚫	×	○	○	○	○	○	×	m	$n \cdot m$	$n \cdot m$	n
Pynchon Gate [SCM05]	2005	RUS (RUMS)	PIR	$(1:m)$-S: TTP	○	○	●	●	▼	×	●	●	○	○	○	×	$n \cdot m$	$n \cdot m$	$n \cdot m$	1
Loopix [PHE+17]	2017	RUS (Mix)	M	f-S: TTP	○	○	●	●	▼	🤫	●	◐	◐	●	k	×	1	ℓ	ℓ	1
Talek [CSM+20]	2020	RUS (RUMS)	PIR	$(1:m)$-S	○	○	●	●	▼	🤫	●	◐	○	○	○	×	m	$n \cdot m$	$n \cdot m$	1

Table 3.1: (I) *Notation:* (i) *Family*: DC = DC-nets, M = Mixnets, RPIR = Reverse-PIR, DP = Differential Privacy, SMC = Secure Multiparty Computation. (ii) *Variables*: n=#clients, m=#servers, m_d=#servers in dialing protocol, f=fraction of malicious servers, ℓ=#servers on an anytrust chain (ℓ=$g(f, m)$ where g depends on the system design to ensure that at least one server in every anytrust chain is honest), b=#broadcasters. (iii) *Assumptions*: TTP: Trusted Third Party. RLWE: Ring Learning With Errors. (x, y)-T: At least x of y servers should be honest, with servers of type T: S - Semihonest, M - Malicious, MA - Malicious but Available. f-T: At most fraction f of m servers can be malicious, with servers of type T. (iv) *Disconnection Impact*: 🚫: system fails, ♉: correspondent leaked, ▼: anonymity set reduction, ≈: temporary anonymity set preservation. (v) *Setup*: 🤫: clients exchange information out of band, 📞: 🤫 + dialing protocol. (vi) *Performance*: Dialing or shuffling (for slot scheduling) costs in blue. Preprocessing/offline costs in gray.

(II) *Meanings of* ◐*:* (i) Robust: non-robust offline phase. (ii) Disruption resistant: disruption resistant scheduling, but not conversations. (iii) Asynchronous: The RUS underlying Pung and SealPIR is asynchronous, but as a CUS it is not asynchronous. Groove allows users to disconnect for a small constant number of rounds. Loopix's asynchrony has a privacy expense (Section 3.3.2). Talek asynchronous messages can be lost by its design (Section 3.4.4).

(III) The blue-dashed box highlights the four crucial properties required for a metadata-protecting messaging systems.

3.4.1 Tradeoffs Between Families

DC-nets are the oldest line of work towards metadata-protecting communications, and can easily be adapted for use towards E2E or broadcast applications. Interestingly, DC-nets innately achieve a CUS by design. Unlike the other systems that attain a CUS by padding up with cover traffic as we discuss later in Section 3.4.2, the notion of cover traffic in these systems is implicit in the design. The primary weakness of DC-nets are its susceptibility to disruptions, and the works we detail in Section 3.3.1 aim to address that. Furthermore, DC-nets are inherently synchronous, and have poor scalability in spite of three decades of research scrutiny.

Meanwhile mixnets have shown incredible promise over the years. While earlier designs like Riffle and cMix lacked the ability to scale, more recent designs like Atom, Loopix, and Trellis provide horizontally scalable constructions. Moreover, some of the recent constructions like Atom and Trellis are even robust. The main shortcoming of this family is its inability to support asynchronous messaging. While Loopix supports asynchrony, it does so at the expense of a stronger trust assumption, namely trusting the service provider node that clients attach to.

The DP-based family can scale horizontally as well as support significantly lower latency than most prior metadata-protecting communication systems. However, this is achieved at the expense of shifting the underlying privacy guarantee; namely, these works provide differentially private metadata protections rather than cryptographic ones. This shift in privacy guarantee has two significant drawbacks. First, DP guarantees are based on the probability gap between two possible scenarios (the user sending a real message or a dummy message), and users gain their metadata protections by being able to provide a 'plausible cover story' to deny their actual action. However, this gap may be sufficient for certain adversaries to act upon. Second, the degree of privacy attained by a user is dependent on the number of rounds in which they participate. The more rounds a user participates in, the more their privacy degrades. Eventually with enough number of rounds they exhaust the allocated privacy budget with which the system was designed, at which point the privacy guarantees are unclear.

On the other hand, the PIR-based (private read) and RPIR-based (private write) families to their merit can support asynchrony, but have poor horizontal scalability and no support for robustness. Their design and experiments are limited to the 2 or 3 server settings, and as the number of users in these system increases they are computation bound by each of these servers. Designs from the SMC-based family lack the ability to scale horizontally. However, recent works from the SMC-based family [APY20, LK22, LYK+19] address anonymity set protection, a property that has received hardly any attention.

3.4.2 Tradeoffs Within Categories

SUBS

Existing SUBS constructions stem from all but DP and PIR families as observed in Table 3.1; the merits of DP and PIR simply do not align with the goal of SUBS and hence this is unsurprising. Amongst existing SUBS, RPIR based constructions (in particular Sabre) outperform other works by orders of magnitude in terms of concrete latency as detailed in Appendix A, but unfortunately do not support robustness, anonymity set protections, or horizontal scalability. Fortunately, recent SUBS constructions from the SMC family (like Blinder) have focused on robustness and anonymity set protection, and achieve these goals albeit at the cost of performance. Unfortunately, almost all existing SUBS as seen from Table 3.1 fall short of being able to scale horizontally, except the mixnet designs Atom and Trellis. However, both these designs achieve horizontal scalability at the expense of significant latency overheads in Atom,[8] and in Trellis at the expense of a rigid immutable set of users who are assumed to be online at all times.

CUS

As evidenced by the table, almost all CUS designs from the literature require either a shuffle (DC-net schemes) or a dialing protocol from its participants prior to any actual communication, which lower bounds the latencies they can achieve.[9] Furthermore, the only horizontally scalable constructions for CUS arise from the DP family, with the exception of XRD, a mixnet CUS design that is horizontally scalable. However, XRD has weaker scalability than that of the DP family as described earlier in Section 3.3.2. The other immediate observation from the table is the lack of asynchronous designs for CUS. However, this lack of asynchrony is in fact a conscious design choice for CUS for two reasons: i) it is easier to analyze metadata protections under a synchronous communication assumption, and ii) having all users synchronously communicate in all rounds of communication circumvents statistical inference and intersection attacks [Dan03, MD04] that prior ACNs fall prey to.

[8] For instance, with 2 million users transmitting a tweet-length message takes about 30 minutes.

[9] The exception is AnonPOP, for which each client has a designated mailbox as we detailed in Section 3.3.2.

RUS

As we mention in Section 3.1, there are three broad directions one can employ to achieve a relationship unobservable system, namely SUMS, RUMS, and mixnets. A RUS based on mixnets is synchronous and in fact provides the exact metadata privacy purported by the definition of a RUS, as messages from all senders during a round get mixed before they are distributed to the intended recipients; this allows an adversary to infer the exact set of senders and recipients in a round of communication. This is permissible by definition of a RUS; recall that the goal is to hide the pairwise sender-recipient relationships.

On the other hand, SUMS and RUMS constructions leverage PIR (in the form of PIR-writes and PIR-reads respectively), and hence innately support asynchrony. In SUMS, senders obliviously send their message into one mailbox amongst the pool of all possible registered mailboxes within the system. These systems do not have a notion of communication rounds; a recipient checks their mailbox periodically to retrieve any messages they may have received.[10] Hence this style of RUS innately benefits from a larger anonymity set (of all possible clients registered in the system) for the recipients, instead of just the true recipient set as in mixnets.

While RUMS has a notion of rounds, recipients in RUMS retrieve messages from the pool of all submitted messages within a round in a fashion that hides the message that they retrieved. Provided all clients in the system attempt to retrieve a message in every round (irrespective of whether they have real messages to retrieve or not), like SUMS, RUMS systems too can support a larger anonymity set for its recipients. This suggests that realizing a RUS via SUMS or RUMS in fact provides better metadata privacy guarantees than the definition of a RUS, since the recipient anonymity set is always the set of all possible clients (that have mailboxes) in the system, and is agnostic of the true recipient set for any batch of messages.

RUS + Cover Traffic = CUS

CUS provides the highest level of metadata protections feasible in a communication network. However, attaining this in practice is non-trivial, and as mentioned earlier typically requires that all participants of the system send and receive messages at all times so as to essentially provide cover traffic for any actual ongoing conversations. RUS on the other hand can be viewed as a relaxation of CUS without the stringent requirement of all users being online (and pretending to converse) at all times, at the

[10] The guidance provided by existing works consequently is that recipients do so in a predictable fashion, agnostic of the underlying real communication pattern.

expense of weaker metadata-privacy guarantees and susceptibility to intersection attacks. The gap between these two categories can be bridged by some trivial impositions on the clients of the system, and boils down to having sufficient cover traffic at all times. Any RUS design can generically be converted into a CUS by requiring that all clients in the system send and receive a (dummy, if they do not have a real) message within an allocated time frame. This is in fact the case with several existing works [KLD20, AS16, ACLS18, AKTZ17, EB22], for which we mention the underlying RUS type in Table 3.1. We note that the two partially asynchronous statuses in the table, for Pung and SealPIR, stem from the fact that the underlying RUS that is converted into a CUS does support asynchrony.

3.4.3 Limitations of Security Assumptions

Amongst the pool of assumptions that have been leveraged towards constructing MPCS, traditional all-clients DC-nets assume the least, as the protocol requires no servers (nor any server assumptions consequently); while clients could be malicious and could DoS the system, they cannot subvert metadata privacy. However, scaling DC-nets requires moving to the anytrust assumption. Assumptions that are based on splitting trust across multiple entities, either anytrust, or non-collusion amongst servers, are hard to realize in practice and moreover dangerous as these trust assumptions can be silently subverted, endangering the privacy of all users of such systems. This makes systems designed on top of CPIR (with assumptions like RLWE) significantly more desirable in practice from a privacy standpoint. Unfortunately such systems are computationally expensive and consequently incur prohibitively high latencies for message delivery; additionally, currently proposed schemes from these families lack horizontal scalability.

The Many Manytrusts

There is a silent but concerning difference in the metadata privacy guaranties attained by the different manytrust systems we detailed in Section 3.3. While on the surface they all require f out of m servers being honest, and seem equivalent in their trust and privacy implications, the way in which these servers are used dictates if metadata privacy guarantees are upheld or not. For instance, Atom's manytrust assumption hopes to distribute at least one of these f honest servers into each group that realizes a logical node in the permutation network used to route messages. Consequently to subvert even a single user's message, the adversary would have to subvert at least one entire path on the entire permutation network; since Atom reuses the nodes of the permutation

network, this means the adversary would have to effectively subvert all the anytrust groups in the system to deanonymize any traffic.[11] Hence, this is the strongest privacy guarantee one can extract under the manytrust model, as unless all manytrust groups are subverted, clients' metadata privacy guarantees are upheld.

A weaker variant of manytrust is achieved by designs like XRD, Stadium, and Karaoke, that leverage multiple anytrust groups in isolation for manytrust. Recall in XRD, the manytrust assumption arises from leveraging an anytrust assumption over each chain of servers. Hence subverting even one chain of servers that depends on anytrust for its metadata privacy results in the adversary being able to undermine metadata privacy of all communications over that particular chain. Similarly for Stadium and Karaoke, the system creates several input and output chains each of which leverage the anytrust assumption, resulting in the manytrust assumption. In this setup, any adversarially controlled input and output chain pair, leaks the metadata of all communications that use these two chains.

The manytrust assumption is at its weakest in Loopix due its stratified network structure, which implies messages get mixed over a few hops of mixing relays. Hence leveraging the same trust assumption of f servers out of all the n servers in the system be honest, results in an uneven privacy distribution as some subset of conversations will have all servers on their hop path be adversarially controlled, resulting in no metadata privacy guarantees.

3.4.4 Asynchronous Message Storage

In the RUS and CUS designs that do not support asynchrony, there is no notion of message storage for later retrieval. However in the asynchronous paradigm, this poses an interesting challenge that has not been well addressed yet. Systems like Pung and SealPIR that leverage CPIR in fact can handle this type of asynchronous storage in the best fashion possible, at the expense of server storage and computation. The server simply stores all the messages received in each round, and clients that reconnect to the system after several rounds can poll each of the rounds they missed to retrieve any messages they missed. This action leaks no additional information as both the CPIR server as well as any global adversary is already aware of when the client last interacted with the system.

Talek introduces a novel middle ground for asynchronous message storage. Recall from Section 3.3.5 that Talek is initiated with system parameters that dictate the TTL for any message in the system. Older messages eventually get evicted to make space

[11] Although if they accomplish this, they get to deanonymize all traffic in the system.

for newer messages. However interestingly their private notification system that uses a Bloom filter, retains the memory of existence of messages in a conversation. While the message itself may have been evicted, this Bloom filter system can notify a user that has gone offline for a long duration of time of the existence of messages they missed and can no longer retrieve. Such users can then request missed messages from their correspondents.

Loopix sidesteps this problem altogether as the design achieves asynchrony by trusting the service provider node, and reveals the number of real messages a client receives and the timing metadata of received messages to this trusted node.

3.4.5 Anonymity Set Protections

Most existing systems towards MPCS do not provide strong protections against anonymity set manipulation attacks as seen from Table 3.1. The ability to manipulate if a particular client participates in a communication system or not enables adversaries to perform several deanonymization attacks. Such an adversary, along with some Sybil clients, can easily subvert the victim's purported metadata privacy in a RUS or SUBS. The adversary simply manipulates the anonymity set or participants in a round to be composed of just the victim (padded up with Sybil client submissions) to deanonymize the recipient of their communication in the case of RUS, or the message they broadcasted in SUBS. While the definition of a CUS may seem to protect its clients from such attacks, that is not the case. An adversary with the ability to manipulate client participation can exploit the underlying scheme's structure to launch deanonymization attacks.

For instance, recall from Section 3.3.2 that in XRD clients send and receive ℓ messages in each round. Excluding a client currently in an active conversation from participating in the system can result in their correspondent receiving one fewer message back to their mailbox in that round. Similarly, in Clarion if an active client is excluded from a round, it will result in their correspondent attempting to collect a message from an index that does not have a submission, again revealing the communication relationship. Hence an important avenue for future research is designing MPCS with strong anonymity set protection guarantees for honest clients participating in the system even in the face of malicious servers, lest they open up to such deanonymization attacks. Techniques towards this end have surfaced in the SUBS literature for the SMC (Asynchromix, Blinder, RPM) and RPIR (Spectrum) families and these ideas need to percolate back into CUS and RUS designs.

3.4.6 MPCS for Metadata-Protected Messaging

The MPCS we cover in this chapter have been designed with two broad application goals, namely anonymous broadcast and metadata-protected messaging. While most of the existing CUS schemes have been proposed towards messaging, we observe that the vast majority of these schemes do not achieve low enough latencies to be a practical messaging system. In order to be usefully deployable, a messaging system requires at least four fundamental properties: (i) low latency, (ii) high throughput, (iii) horizontal scalability, and (iv) support for asynchrony. Table 3.1 highlights systems that have low latencies (even with a generous definition of 'low latency'), and we observe that only nine systems achieve this.[12]

Out of these nine systems, Pung, SealPIR, Express, Riposte, and Sabre (the RPIR and PIR based schemes) all have a fundamental problem that messages are processed sequentially; i,e, this low latency is only true if one user submits a message and others do not. If all users submit their messages at the same time, each message has to be processed sequentially and users would experience significant delays in delivering their messages. Furthermore, their horizontal scalability is quadratic in the amount of user growth ($k^2\times$ as seen in the table for an increase of $k\times$ users). So while these five systems support asynchrony, their low latency has the drawback of poor throughput, and subpar horizontal scalability. Loopix on the other hand has low latency and high throughput, but their asynchrony is at the expense of a trust assumption as detailed in Section 3.3.2, and additionally as we mentioned in Section 3.4.3 their design has an uneven privacy distribution. Finally, Karaoke, Groove, and Boomerang support messaging with low latency and high throughput, and horizontal scalability, but fall short of supporting asynchrony. Groove proposes novel directions for temporary disconnection of users (a form of partial asynchrony), permitting users to go offline for a small constant number of rounds after which they can no longer receive messages. We see from our systematization that there is a clear dearth of MPCS designs that can simultaneously achieve these four fundamental properties.

[12] Loopix has a partial marker for messaging latency, as their implementation only demonstrates a < 2 s latency with 500 clients, but the design of the system suggests that even as users grow the latency should not increase significantly more, as the route length remains same with more servers to distribute traffic over.

3.5 Conclusion

In this chapter, we systematize 32 recent systems proposed for metadata-protecting communications. Since each of these systems make different design choices and assumptions, they have their own advantages and disadvantages. In this work we group them into four categories based on the functionality and metadata-privacy guarantees they achieve, and into six families based on the core underlying technique that underpins the metadata protection. We examine the innate benefits and shortcomings of these categories and families of works towards this goal. We point out subtleties in the way some security assumptions are used to design MPCS that make certain systems provide a false sense of privacy, as well as identify promising directions for future research to further metadata-protected communications. Finally, we see from our systematization that there is a clear dearth of MPCS designs that can simultaneously achieve the four fundamental properties desirable of any messaging system, and this motivates the design of TEEMS that we will build towards in the remainder of this thesis.

Chapter 4

ORCompact: Oblivious Recursive Compaction

In this chapter, we start by establishing the notion of fully oblivious algorithms that are safe to run within TEEs. We then discuss oblivious compaction algorithms, and present our algorithm, ORCompact. Oblivious compaction is a generally useful tool for oblivious algorithms, with one common use being to filter out "dummy" items inserted earlier to mask conditional memory writes (e.g., in cryptographic primitives like oblivious RAMs (ORAMs) [AKL+20a] or an oblivious database query [KKS20]). In fact, ORCompact is a key component of the oblivious algorithms we present later in Chapters 5, 6, and 7. We then present our experimental results from implementing ORCompact, and compare it against the prior state-of-the-art practical oblivious compaction algorithm, Goodrich compaction [Goo11]. Finally, we present formal definitions for full obliviousness and provide proofs of correctness, efficiency, and obliviousness for ORCompact.

4.1 Fully Oblivious Algorithms

Informed by the attacks on TEEs that we discussed in Section 2.2, we identify three independent ways in which an enclave program can be oblivious to secret data.

1) **External-memory oblivious**: Memory accesses to data outside of the PRM are independent of any secret data.

2) **Protected-memory oblivious**: Memory accesses to data within the PRM are independent of any secret data. The literature has provided three finer granularities of attack surfaces on protected memory access patterns, namely:

i) **Page oblivious**: Several proposed attacks leverage the fact that the untrusted OS is still responsible for page table management of enclave programs. These attacks induce page faults on enclave programs to extract the memory locations accessed by the program. However, SGX masks the last 12 bits of a page fault address to the untrusted OS, thus limiting the immediate memory access pattern leakage to a malicious OS to page-level granularity (4 KiB) [SCNS16, VWK+17].

ii) **Cacheline oblivious**: Advanced attacks [BMD+17, MIE17, LJF+20] can extract the precise address of the (64 B) cacheline loaded during a page fault.

iii) **Subcacheline oblivious**: Recently we have witnessed the emergence of side-channel attacks at a finer granularity than cache level [SZB+24].[1] Such attacks can extract the exact (8 B) word loaded from a cacheline to a register in a memory fetch operation.

3) **Control-Flow oblivious**: Orthogonal to leakages from data accesses induced by the execution of the trusted program, another form of leakage arises with secret-dependent control flow branches [LSG+17, MVH+20]. These attacks can extract the exact branch chosen in a secret-dependent control flow branch, thus leaking information about the secret. We say a program is control-flow oblivious if it has no control branches in its execution dependent on secret data.

We say that a program is *fully oblivious* if it satisfies all the above definitions of obliviousness. We provide a formal definition for full obliviousness later in Section 4.6, along with the proofs for ORCompact's obliviousness, correctness, and efficiency. While previous work using TEEs [BEM+17, ZDB+17, MPC+18, PAS+20, KKS20] has satisfied a few of these definitions, none has targeted all of them. By providing full obliviousness, we defend against all attacks that depend on memory or control-flow side channels [XCP15, SCNS16, BMD+17, VWK+17, MIE17, LJF+20, LSG+17, MVH+20].

Without fully oblivious algorithms a TEE adversary, in the worst case, can distinguish the locations of real blocks in the array from those of dummies in the compaction setting, and can infer the final permutation of a shuffle or a sort. These algorithms are typically used as building blocks for applications like Prochlo to provide an anonymity set to users' data by shuffling them before analysis, or for providing strong differential privacy guarantees for data in privacy-preserving machine learning [KCY23, SO19]. Leakages in the underlying oblivious shuffle or sort hence either deanonymize users or provide significantly weaker privacy properties than claimed.

[1] At the time of publishing this work [SJG22a], such attacks had not been discovered yet. Nonetheless, we conceived that future research might uncover such attacks and hence we captured this is in the definition of fully oblivious back then.

4.1.1 Fully Oblivious Assembly Verifier (FOAV)

An obstacle to obtaining full obliviousness is that even if source code is control-flow oblivious, compilation may not preserve that obliviousness. Compilers are notorious for optimizing away programming logic that was intended to provide security guarantees like constant-time code, or side-channel resilient memory access patterns [SCA18]. To address this, we design FOAV, a tool that helps verify the control-flow obliviousness of the program *binary*. We use FOAV to ensure full obliviousness of all the algorithms we present in this dissertation.

FOAV ensures that the binary executed by the processor is control-flow oblivious. We instrument our C/C++ source files to insert assembly comments that state if a value in a register is "safe" before any (possibly) conditional branch inducing instructions, using `__asm__` macros. We then designed a tool that analyzes the output assembly produced by the compiler, to verify control-flow obliviousness. The tool tracks all the conditional jump instructions in the final assembly, and the corresponding *flag-manipulating instruction* (FMI) (such as `test`, `cmp`, `dec`, etc.) for each of these conditional jumps, and parses the lines above them to check if the operands of these FMIs have been marked as safe. Compiler optimizations make automating this process challenging since the optimizations may move operands to different registers before performing any FMI on them, which causes our registers marked safe by the comments to mismatch with the ones used by the FMI. To resolve this, our tool parses the assembly lines above each of the conditional jump, tracking both the register operands of the FMI, and the registers that were already marked explicitly safe from source comments, to identify other registers that may be deemed safe as well. This close analysis is performed up until it hits the start of a function or a jump instruction, and if it still cannot resolve a conditional branch as safe it marks it for closer human inspection.

4.2 Oblivious Compaction

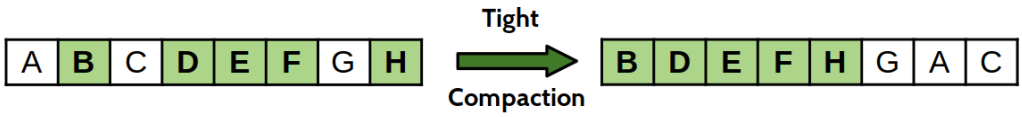


Figure 4.1: Tight compaction with marked items in **bold** and with green background.

Compaction is the problem of permuting items in an array such that a "marked" subset of them appears at the beginning. There are several flavors of compaction algorithms, such as *order-preserving* or non-order-preserving compaction, and *tight* or loose compaction. In order-preserving (or stable) compaction the relative ordering of the marked items is preserved in the output of the algorithm; note that there is no requirement on the relative ordering of the unmarked items in the definition of order-preserving compaction. *Tight compaction* requires all the marked compacted items to be brought to the head of the array without any unmarked items between them, while loose compaction permits some amount of slack for unmarked items to be interleaved with the marked items at the head of the array. In our work, we focus on **oblivious order-preserving tight compaction**. Figure 4.1 illustrates the outcome of an order-preserving tight compaction.

Several oblivious compaction algorithms were proposed recently [LSX19, AKL+20a, DO20, AKL+20b], but these algorithms are not order-preserving and have high constants; practical improvements have not been observed for order-preserving compaction since Goodrich [Goo11]. Our oblivious recursive compaction algorithm, ORCompact, improves performance in practice, which we demonstrate experimentally. Moreover, ORCompact is *order-preserving*; that is, it maintains the relative order among marked items. This property, also provided by Goodrich compaction, is useful when compaction is used as a subroutine and later steps require that marked items stay in the same order [KKS20].

It is also important that our algorithms be parallelizable because many TEE systems, including Prochlo and PSQF, are designed to process large amounts of data. We therefore give efficient parallel versions of our shuffling and compaction algorithms, and we implement and test them. We note that our parallel compaction algorithm is the first practical such algorithm to be explicitly described, and a straightforward parallelization of Goodrich compaction adds significant overhead.

4.3 ORCompact: Oblivious Recursive Compaction

We introduce the fully oblivious recursive compaction algorithm ORCompact. Compaction is a fundamental primitive for ORAMs and TEE systems, and we will also use it for oblivious shuffling (see §5.1). ORCompact takes as input an array D of n data items and a bit array M of length n. The positions of M with a 1 value indicate "marked" items, and the positions with a 0 value indicate "unmarked" items. ORCompact rearranges the items in D such that all marked items appear before all unmarked items, and it does so

fully obliviously. Furthermore, as we will show, ORCompact is order-preserving; that is, it maintains the relative order of the marked items. Note that the relative order of the unmarked items may change, however.

This algorithm has time complexity $O(n \log n)$, which is the same as the algorithm of Goodrich [Goo11] and which is higher asymptotic complexity than the linear-time optimum [AKL+20a, DO20]. However, unlike the asymptotically optimal algorithms, ORCompact is order-preserving and practically efficient for realistic values of n. It is faster than the Goodrich algorithm by a factor of at least 2. It is also more efficiently parallelizable. We present more detailed comparison against other algorithms in Section 4.4.

4.3.1 Algorithm

We use $D_{j..k}$ to indicate the subarray $(D_j, \ldots, D_k)$. We denote the oblivious computation of the comparison c as a boolean value with $[\![c]\!]$ (e.g., $[\![x < y]\!]$ has value 1 if $x < y$ and 0 otherwise). OSwap(D_i, D_j, b) refers to an oblivious swap subroutine that swaps the ith and jth values of D if $b = 1$ and leaves D unchanged if $b = 0$. It can be efficiently implemented obliviously as we will discuss in Section 4.3.2. Arithmetic operations must also be implemented in an oblivious way.

The core compaction logic is contained in the subroutine OROffCompact(D, M, z) (see Figure 4.2). OROffCompact compacts marked items to a region starting at an offset position z, with possible wraparound. However, it can only be called on an array D with a length that is a power of two because it divides D into halves to perform pairwise OSwaps. OROffCompact is fully oblivious to its input values (though not their sizes) including in particular the offset z. Because $|D|$ is a power of two, simple bitwise operators can obliviously implement the division and modulo operations.

The full ORCompact(D, M) algorithm is given in Figure 4.3. Intuitively, ORCompact divides D into a right side, which is the largest suffix with size n_1 a power of two, and a left side, which is the remainder of size $n_2 = n - n_1$. It then recursively compacts the n_2 items on the left and also computes the number of selected items m on that side. Finally, it compacts the right side with an offset that is $n_2 - m$ items from the end, and the final n_2 items on the right are conditionally swapped pairwise with the n_2 items on the left. OROffCompact works similarly, although it also takes into account that there may be an offset for the desired compaction. Because it can assume its input is a power of two, items on the left side and right side can be paired for potential swaps, which is why it can produce an offset in any position obliviously.

Figure 4.2: ORoffCompact(D, M, z): Compact items in D, as marked in M, to an offset z. D and M must have a power-of-two length.

1: $n \leftarrow |D|, m \leftarrow \sum_{i=0}^{\frac{n}{2}-1} M_i$
2: **if** $n = 2$
3: OSwap(D_0, D_1, $((1 - M_0)M_1) \oplus z$)
4: **else if** $n > 2$
5: ORoffCompact$\left(D_{0..\frac{n}{2}-1}, M_{0..\frac{n}{2}-1}, z \bmod \frac{n}{2}\right)$
6: ORoffCompact$\left(D_{\frac{n}{2}..n-1}, M_{\frac{n}{2}..n-1}, (z+m) \bmod \frac{n}{2}\right)$
7: $s \leftarrow \left[\!\left[\left((z \bmod \frac{n}{2}) + m\right) \geq \frac{n}{2}\right]\!\right] \oplus \left[\!\left[z \geq \frac{n}{2}\right]\!\right]$
8: **for** $i \leftarrow 0, \ldots, \frac{n}{2} - 1$
9: $b \leftarrow s \oplus \left[\!\left[i \geq \left((z+m) \bmod \frac{n}{2}\right)\right]\!\right]$
10: OSwap(D_i, $D_{i+\frac{n}{2}}$, b)

Figure 4.3: ORCompact(D, M): Compact items in D as marked in M. D and M need not have a power-of-two length.

1: **if** $|D| = 0$ **return**
2: $n \leftarrow |D|, n_1 \leftarrow 2^{\lfloor \log_2(n) \rfloor}, n_2 \leftarrow n - n_1, m \leftarrow \sum_{i=0}^{n_2-1} M_i$
3: ORCompact($D_{0..n_2-1}$, $M_{0..n_2-1}$)
4: ORoffCompact($D_{n_2..n-1}$, $M_{n_2..n-1}$, $(n_1 - n_2 + m) \bmod n_1$)
5: **for** $i \leftarrow 0, \ldots, n_2 - 1$ **do**
6: $b \leftarrow [\![i \geq m]\!]$
7: OSwap(D_i, D_{i+n_1}, b)

4.3.2 Correctness, Obliviousness, and Efficiency

Now we present our concrete claims on the correctness, obliviousness and efficiency of ORCompact. We provide informal arguments for our results, and their implications here in this subsection. The formal proofs for these theorems are provided later in Section 4.6.

Correctness

Theorem 1 shows that ORCompact compacts the marked items and maintains their relative order.

Theorem 1. *Given D and M, let $(D'_0, \ldots, D'_{w-1})$ be the subsequence of D consisting of all w items marked in M. ORCompact(D, M) rearranges the items in D such that $(D_0, \ldots, D_{w-1}) = (D'_0, \ldots, D'_{w-1})$.*

Obliviousness

We now explain how ORCompact is fully oblivious, assuming an implementation that follows Figure 4.3 in the straightforward way. The result is fully oblivious with respect to the values of its input arrays, although not to their size or location in memory. In fact, not only are all instructions and memory accesses independent of the input array values, as required for full obliviousness, they are *deterministic* given the size and location of those arrays. Here we present an informal explanatory argument for why ORCompact is fully oblivious, later in Section 4.6, we provide a formal statement and proof of this obliviousness.

Prior work [OSF+16, CC18] has shown how to efficiently implement comparisons (e.g., $[\![i \geq m]\!]$) and swaps (OSwap) fully obliviously. These implementations are deterministic and either use conditional instructions such as `SETB` and `CMOVZ` or Boolean arithmetic. The comparisons are oblivious to the compared values but not to the comparison operator. OSwap is oblivious to the input values and to the swap flag b. It is not oblivious to locations of the inputs.

We next argue that OROffCompact is fully oblivious. We can see that the only conditional branching (i.e., the `if`/`else` and `for`) is based on the length n of the input arrays and not on their contents or on the offset z. Thus, the instruction sequence is deterministic given the input size. We then observe that all accesses to the inputs are deterministic given n. As previously argued, the computation of comparisons is fully and deterministically oblivious to the values of its inputs. Similarly, the calls to OSwap are oblivious to

its last argument, and we observe that the other arguments are deterministic locations given n. We can recursively assume that the calls to ORoffCompact are fully and deterministically oblivious. Therefore, ORoffCompact is fully oblivious, and its instruction sequence and memory accesses are deterministic given the length of its input arrays D and M.

Finally, we argue that ORCompact (Figure 4.3) is fully oblivious. We observe that its only conditional execution is a `for` loop (Line 5) that executes a number of times that depends deterministically on n. We can also see that its variable accesses are all a deterministic function of n. By our previous arguments, the comparisons and ORoffCompact call are fully oblivious. Similarly, each OSwap call is fully oblivious to its last argument, and the other arguments are deterministic functions of n. Finally, we can recursively assume that the ORCompact call is fully oblivious. Therefore, ORCompact is fully oblivious, and its instruction sequence and memory accesses are deterministic given the input size.

Efficiency

We analyze the time and space efficiency of ORCompact as the number and size of data items grow. Theorem 2 gives the asymptotic runtime of ORCompact. As previously noted, this runtime is not the asymptotic optimal linear time, but, as our experiments results will show, the constants are low enough that it outperforms all other oblivious algorithms for realistic input sizes.

Theorem 2. *ORCompact runs in time $O(n \log n)$.*

We next consider the number of oblivious swaps (i.e., calls to OSwap) because the data items themselves are only accessed during swaps. Each swap takes time linear in the item size, and so, for large enough items, oblivious swaps will constitute an arbitrarily large fraction of the runtime. We count the number of oblivious swaps, which admits a concrete (rather than purely asymptotic) estimate, thereby enabling a direct comparison to other swap-based algorithms. Let n be the length of the input arrays to ORCompact. Theorem 3 provides upper and lower bounds on the exact number of OSwap calls in ORCompact.

Theorem 3. *Let $S_1(n)$ count the OSwap calls in ORCompact. Then*

$$\left(2^{\lfloor \log_2 n \rfloor}/2\right) \lfloor \log_2 n \rfloor \leq S_1(n) \leq \lfloor (n/2) \log_2 n \rfloor.$$

Theorem 3 shows that ORCompact performs approximately $(n/2)\log_2 n$ OSwaps. We can use this estimate to compare performance to the tight oblivious compaction algorithm of Goodrich [Goo11], which appears to date to be the fastest algorithm in practice [KKS20]. This algorithm can be adapted to perform compaction fully obliviously using OSwaps (the original algorithm description assumes some private memory and does not require unmarked items to be output). However, it uses more than $(\log_2 n - 2)n$ OSwaps, which suggests a runtime of about $2\times$ that of ORCompact when item sizes are large enough for swaps to constitute most of the runtime (for small item sizes, we see in Section 4.5 that ORCompact outperforms Goodrich by an even larger factor).

We note that ORCompact performs *in-place* compaction and thus need not maintain extra copies of possibly large data items. Moreover, the additional memory requirements are very small, with a small constant amount of memory needed at each of the $\lceil \log_2 n \rceil$ levels of recursion. We also observe that the memory-access patterns are highly regular (linear scans) and generally with high locality (especially at the deeper levels of recursion), which yields good practical performance due to caching and prefetching.

4.3.3 Parallelization

We describe how ORCompact can be parallelized by giving a version of it in the EREW PRAM model [MR95]. In this model, multiple processors proceed in synchronized steps, and they access shared memory with exclusive read and write (i.e., only one processor can read or write a given memory location in a given step). OROCPar, given in Figure 4.4, performs parallel oblivious compaction to an offset when n is a power of two, and ORCPar, given in Figure 4.5, performs parallel oblivious compaction for any n. We indicate parallel computations with a **do in parallel** block, in which each line can be computed in parallel, and a **parallel for** block, in which each loop iteration can be computed in parallel. This presentation omits explicit assignment to processors. For our algorithms, we consider the *work* (i.e., total computations across all processors) and the number of parallel *steps* (i.e., longest sequence of operations that any single processor might execute).

ORCPar uses the subroutine PrefixSumPar to compute the array S containing the sums of prefixes of the boolean array M. S is used by both ORCPar and OROCPar to compute offsets in constant time. Hillis and Steele [HSJ86] give an algorithm to compute the prefix sums obliviously in $\lceil \log_2 n \rceil$ steps and $O(n \log n)$ total work. Note that ORCPar takes S as an extra argument, which should initially be empty, and that S is produced with a length of $n + 1$, where each element is the sum of the preceding positions in M (with $S_0 = 0$). S can thereby be used recursively to compute subsequence

Figure 4.4: ORCPar(D, S, z): Parallel compaction of D to offset z. S is prefix sums of marked items. D must be a power-of-two length.

1: $n \leftarrow |D|, m \leftarrow S_{\frac{n}{2}} - S_0$
2: **if** $n = 2$ **then**
3: OSwap$(D_0, D_1, ((1 - (S_1 - S_0))(S_2 - S_1)) \oplus z)$
4: **else if** $n > 2$ **then**
5: **do in parallel**
6: ORCPar$\left(D_{0..\frac{n}{2}-1}, S_{0..\frac{n}{2}}, z \bmod \frac{n}{2}\right)$
7: ORCPar$\left(D_{\frac{n}{2}..n-1}, S_{\frac{n}{2}..n}, (z+m) \bmod \frac{n}{2}\right)$
8: $s \leftarrow \left[\!\left[\left((z \bmod \frac{n}{2}) + m\right) \geq \frac{n}{2}\right]\!\right] \oplus \left[\!\left[z \geq \frac{n}{2}\right]\!\right]$
9: **parallel for** $i \leftarrow 0, \ldots, \frac{n}{2} - 1$
10: $b \leftarrow s \oplus \left[\!\left[i \geq (z+m) \bmod \frac{n}{2}\right]\!\right]$
11: OSwap$(D_i, D_{i+\frac{n}{2}}, b)$

sums.

We observe that the parallelization does not significantly affect the work. As with ORCompact, the work in ORCPar is $O(n \log n)$. In addition, the total number of the OSwap calls remains the same as in ORCompact (quantified in Theorem 3). At the same time, parallelization significantly improves step complexity, with $O(\log n)$ parallel steps and $\lceil \log_2 n \rceil$ steps involving OSwap calls.

4.4 Related Work

Recent works present linear time compaction algorithms [AKL+20a, DO20, AKL+20b] but they are more theoretical in nature. These algorithms are non-order-preserving and have high underlying constants. In fact Lin et al. [LSX19] proves that any oblivious algorithm must incur $\omega(n \log n)$ work to tight compact an arbitrary input array of n elements while preserving order of marked items.

Goodrich [Goo11] defines compaction as outputting only the m marked elements among n total elements. Variants for *tight* and *loose* compaction are given where the output array is of size exactly m or of size $O(m)$. Goodrich gives an $O(n \log n)$ order-preserving tight compaction algorithm. It can be implemented in a way that outputs all n elements, starting with the m marked items in their original relative order (but possibly reordering the unmarked ones), using as few as $(\log_2 n - 2)n$ oblivious swaps. Note

Figure 4.5: ORCPar(D, M, S): Parallel compaction of D as marked in M. D and M need not be power-of-two lengths. Pass empty S in initial call.

1: **if** $|D| = 0$ **then return**
2: **if** $|S| = 0$ **then** $S \leftarrow$ PrefixSumPar(M)
3: $n \leftarrow |D|$, $n_1 \leftarrow 2^{\lfloor \log_2(n) \rfloor}$, $n_2 \leftarrow n - n_1$
4: **do in parallel**
5: ORCPar$(D_{0..n_2-1}, \emptyset, S_{0..n_2})$
6: OROCPar$(D_{n_2..n-1}, S_{n_2..n}, (n_1 - n_2 + S_{n_2}) \bmod n_1)$
7: **parallel for** $i \leftarrow 0, \ldots, n_2 - 1$
8: $b \leftarrow [\![i \geq S_{n_2}]\!]$
9: OSwap(D_i, D_{i+n_1}, b)

---

that this number is roughly a factor two more swaps than in ORCompact. Goodrich compaction can also be parallelized with $\log_2 n$ rounds at the costs of i) using a second array to move items into in parallel (otherwise items would be read and written in the same step), and ii) performing compaction on both the marked and unmarked items separately and then merging the result (otherwise the unmarked items would be overwritten). This parallel form requires approximately double the work and quadruple the memory cost of the sequential one, in contrast to ORCompact, for which the total parallel and sequential costs are about the same.

Asharov et al. [AKL+20a] give a linear-time tight compaction algorithm. Dittmer and Ostrovsky [DO20], however, show that the constant for this compaction algorithm is more than 2^{228}. They give a different $O(n)$ tight compaction algorithm that runs in time at most $9405.73n$, which, for realistic n, is much larger than the additional $\log n$ and constant factor in Goodrich compaction. Similarly, Mitchell and Zimmerman [MZ14] give an $O(n \log \log n)$ algorithm for compaction, but the analysis by Falk and Ostrovsky [FO21] indicates that the algorithm requires $4n \log \log n + 14n$ OSwaps (with a 2^{-40} failure probability), which is more than in ORCompact unless $n > 2^{78}$.

We also note that fully oblivious compaction is suitable for other (non-TEE) secure computing technologies, such as secure multiparty computation (MPC) and fully homomorphic encryption. For example, Blanton and Aguiar [BA16] use Goodrich compaction to perform MPC set operations. They could instead use ORCompact and reduce the expensive comparisons by a factor of two.

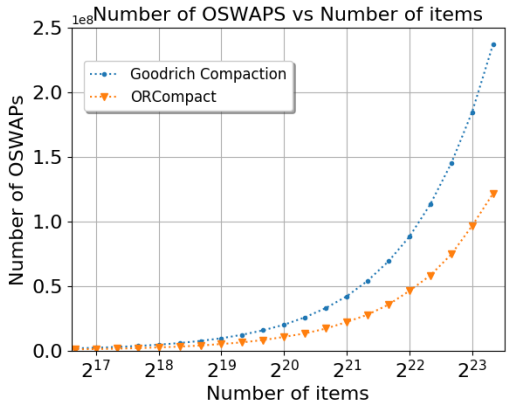


Figure 4.6: Number of OSwaps with b=8 B

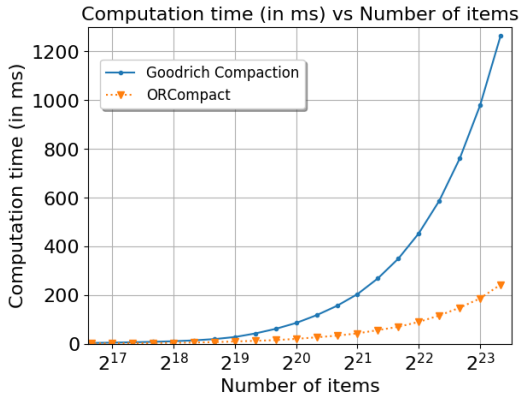


Figure 4.7: Computation time (in ms) with b=8 B

4.5 Implementation

We implement and benchmark our algorithms on a server-grade Intel Xeon E3-1270 running Ubuntu 20.04 with four physical cores, 64 GB of DDR4 RAM, and support for Intel SGX. Our implementation is open-source and publicly available [SJG22b]. The fundamental building block of our algorithms are oblivious swaps (OSwaps). Given two data elements and a `swap_flag`, OSwap swaps them if `swap_flag` is true, while not leaking any side-channel information about the contents or whether the swap was performed. Similar to prior work [RLT15, OSF+16, SGF18], we develop these building blocks by leveraging the x86 `CMOV` instruction. In our implementation we develop a library of templated inline assembly functions that perform OSwap on buffers. These functions are templated on the size of the buffer so that we can measure performance across varying problem sizes.

Although the amount of PRM is limited, Intel SGX does allow applications to use PRM beyond that limit via virtual memory. Using more memory than the PRM limit does however incur additional paging overheads of bringing the encrypted memory pages into the PRM, since the MEE performs an integrity and freshness verification on these pages via the Enclave Page Cache Map (EPCM) [CD16], which is a portion of the PRM that is inaccessible to user space and set aside to ensure SGX's isolation guarantees.

In Figure 4.6, we compare the performance of our ORCompact algorithm against Goodrich's compaction algorithm [Goo11] in terms of the number of OSwaps performed. As expected from Theorem 3, our ORCompact algorithm performs about half the number of OSwaps as Goodrich's compaction algorithm. Furthermore from Figure 4.7, we see

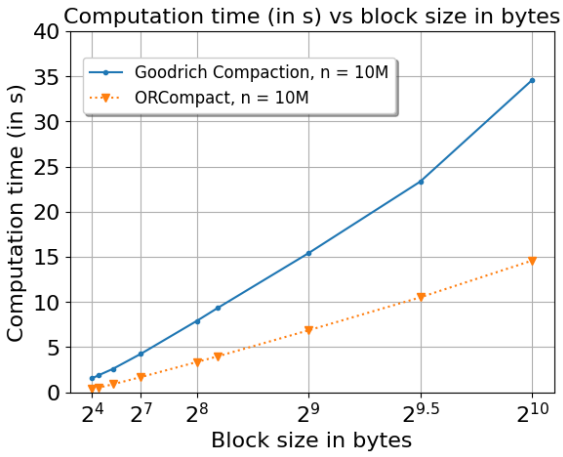


Figure 4.8: Computation time (in s) with $n = 10$ M. The elbow for Goodrich compaction corresponds to it incurring paging overheads from exceeding the PRM limit.

that the performance gap can grow much larger than $2\times$ for larger choices of n with a block size of 8 B. For instance, at the largest value of $n = 2^{23.33}$ that we see in Figure 4.7, ORCompact takes 243.1±0.3 ms, while Goodrich compaction takes 1263±0.3 ms; i.e., a $5.2\times$ speedup over the state of the art. The larger performance gap is because Goodrich compaction must update a distance array of size n in each of its $\log n$ rounds, while ORCompact performs a single linear-time computation to track the number of marked items in every subarray of the given problem.

This additional cost becomes less significant as the block size b increases, however. In Figure 4.8, we plot how the computation time varies for both of these compaction algorithms as b increases using a fixed $n = 10^7$. It shows that the timing gap persists even across larger block sizes. As the block size increases, however, the timing cost is dominated by OSwaps on those larger blocks, and so the performance gap becomes closer to $2\times$.

4.6 Formal Definitions and Proofs

In order to define full obliviousness, we first describe relevant aspects of our computational model. We consider a RAM model in which a set of instructions, a set of constant values, a set of registers, and a memory are available for computation. Each instruction takes a fixed number of registers to read from and write to. A subset of instructions reads

from or writes to memory, the locations of which are to be specified in registers, and a subset of instructions can change the next instruction to be executed on the basis of a register value. An algorithm A consists of a sequence of instructions with registers given as arguments, which are executed sequentially unless the flow is changed by an instruction. A terminates when the address of the next instruction to be executed exceeds the length of A. We write $A(x_1, \ldots, x_k)$ to indicate that A operates on inputs $(x_1, \ldots, x_k)$, where the inputs reside in memory starting at a fixed location (e.g., at the beginning). A also has access to a sequence of random bytes by reading from a special memory region.

An *execution* of A is the entire sequence of states during execution. It consists of the sequence of executed instructions, the register values, and the memory contents. Let $\mathcal{E}(A, x_1, \ldots, x_k)$ denote an execution of A on inputs $(x_1, \ldots, x_k)$. $\mathcal{E}$ is a random variable due to the possibly random behavior of A.

The *output* of an execution indicates the final value produced by the algorithm and returned to the caller. Given execution $E = \mathcal{E}(A, x_1, \ldots, x_k)$, $\mathcal{O}(E)$ denotes the output of E. $\mathcal{O}(E)$ is a random variable due to the randomness of E.

The *instruction trace* of an execution indicates the order in which the instructions of an algorithm were executed. Given execution $E = \mathcal{E}(A, x_1, \ldots, x_k)$, $\mathcal{I}(E)$ denotes the instruction trace of E, that is, the sequence of indices $(i_1, i_2, \ldots)$, $i_j \in \mathcal{N}^+$, pointing to each instruction in E. By convention, $i_1 = 1$, and, if the execution is of finite length ℓ, then $i_\ell > |A|$. Note that $\mathcal{I}(E)$ is a random variable due to the randomness of E.

We similarly define the *memory trace* of an execution to be the sequence of memory locations that are accessed during execution, along with a bit indicating if they are read or written. Let $E = \mathcal{E}(A, x_1, \ldots, x_k)$. Then $\mathcal{M}(E)$ denotes the memory trace of E, that is, the sequence of values (t, x, ℓ), where $t \in \mathcal{N}^+$ indicates that memory was accessed during the tth instruction execution, $x \in \{0, 1\}$ indicates if the memory access was a read or write, and $\ell \in \mathcal{N}^+$ indicates the memory location accessed. $\mathcal{M}(E)$ is a random variable due to the randomness of E.

We use a simulation-based definition of full obliviousness. If two random variables have identical distributions, we say that they are *perfectly indistinguishable*. Let $[k]$ denote the set $\{1, \ldots, k\}$. Let $I \subseteq [k]$ indicate the *secret* inputs, $\{x_i\}_{i \in I}$, with the remaining inputs considered *non-secret* inputs. The simulator will get the non-secret inputs but only the lengths of the secret inputs.

We define fully oblivious algorithms as follows:

Definition 1. Let $E = \mathcal{E}(A, x_1, \ldots, x_k)$ be the execution of algorithm A. For any $I \subseteq [k]$, A is *fully oblivious* with respect to $\{x_i\}_{i \in I}$ if there exists a simulator $\mathcal{S}$ such that

for all inputs $(x_1, \ldots, x_k)$, $\left(\mathcal{O}(E), \mathcal{S}\left((x_i)_{i\in[k]\setminus I}, (|x_i|)_{i\in I}\right)\right)$ is perfectly indistinguishable from $(\mathcal{O}(E), (\mathcal{M}(E), \mathcal{I}(E)))$.

Observe that Definition 1 requires that, for any possible output value $\mathcal{O}(E) = o$, the joint distribution of the instruction and memory traces, conditional on $\mathcal{O}(E) = o$, is the same for both $\mathcal{S}$ and A. Including the output in the definition ensures that the traces do not reveal any additional information about that output beyond what is implied by the inputs to $\mathcal{S}$. A weaker version of this definition, which we will use later in Chapter 5 for BORPStream, is *statistical* full obliviousness, which takes a parameter λ (also implictly given to the target algorithm A and the simulator $\mathcal{S}$) and replaces "perfectly indistinguishable from" with "within statistical distance $2^{-\lambda}$ of". An even weaker definition is to require only that the distributions are computationally indistinguishable with security parameter λ, which we will use later in Chapter 6 for setting the control bits of a Waksman network.

Note that this definition does not take into account the possibility that some instructions might be variable-time, depending on their input values, which could yield execution times that depend on secret data even thought the instruction and memory traces are identical. Our computational model abstracts away instruction-level timing, but actual implementations must take this into account and should only use constant-time instructions when operating on sensitive data. Our implementation does satisfy this additional requirement.

Finally, observe that the following primitive operations can be implemented with full obliviousness due to deterministic traces: arithmetic operations, OSwap, bitwise Boolean operators, and comparisons. We will prove the obliviousness of our algorithms assuming they use oblivious implementations of these primitives.

4.6.1 Obliviousness

To prove the obliviousness of ORCompact, we first prove in Lemma 1 that OROffCompact is fully oblivious. Moreover, the memory and instruction traces are deterministic functions of the input size.

Lemma 1. *OROffCompact(D, M, z) is fully oblivious with respect to D, M, and z, given that $n = |D| = |M| = 2^k$, $k \in \mathcal{N}$. In addition, its instruction and memory traces are deterministic given n.*

Proof. We describe a simulator $\mathcal{S}$ that outputs the same instructions and memory accesses as OROffCompact(D, M, z), knowing only $n = |D|$.

In Line 1, the instructions and memory accesses are deterministic and depend only on n, and $\mathcal{S}$ outputs them.

If $n < 2$, OROffCompact returns after Line 1.

If $n = 2$, then OSwap is called (note that executing this case depends only on n). The instructions and memory accesses to compute its swap bit are independent of the inputs and are deterministic. The OSwap instructions and memory accesses are input-independent and deterministic given the locations to swap, and those locations are input-independent and deterministic (D_0 and D_1). Therefore, if $n = 2$, $\mathcal{S}$ outputs those instructions and memory accesses.

Otherwise, Lines 5–10 are executed (again, this conditional execution depends only on n). OROffCompact is called in Lines 5 and 6 with subarrays of D and M that are deterministic functions of n, and it is given offset values that can be computed obliviously and deterministically (note that $n/2$ can be obliviously computed with a bit shift, and $(\mathrm{mod}\ n/2)$ with an additional bit mask, as n is a power of two). The traces of these calls are produced by $\mathcal{S}$ recursively. The intermediate swap flag s can be computed in Line 7 with a fixed sequence of instructions, given that comparisons are computed obliviously. Finally, the pairwise swaps in Lines 8–10 occur between fixed pairs in D (viz., D_i and $D_{i+n/2}$), the swap bit b can be computed using a fixed instruction sequence, and the final OSwap call is fully oblivious to the values of its arguments. $\mathcal{S}$ outputs this deterministic sequence of memory accesses and instructions if $n > 2$.

$\mathcal{S}$ thereby outputs memory and instruction traces identical to those of ORCompact, and they are deterministic given n. ORCompact is therefore fully oblivious to all of its inputs. □

Theorem 4 proves that ORCompact is fully oblivious and that its memory and instruction traces are deterministic functions of the input size.

Theorem 4. *ORCompact(D, M) is fully oblivious with respect to D and M. In addition, its instruction and memory traces are deterministic given $n = |D| = |M|$.*

Proof. We describe a simulator $\mathcal{S}$ that outputs the same instructions and memory accesses as ORCompact(D, M), given only n.

Line 1 returns only if $n = 0$, which $\mathcal{S}$ knows.

The values computed in Line 2 can be computed using deterministic memory accesses and instructions that are independent of the input values. $\mathcal{S}$ outputs these traces.

Lines 3 and 4 call ORCompact and OROffCompact using arguments that are subarrays of D and M depending only on n, as well as offsets that can be computed with a fixed instruction sequence and input-independent memory accesses. $\mathcal{S}$ outputs the memory and instruction traces used to compute these arguments, and then it recursively produces the instruction and memory traces of ORCompact and OROffCompact.

Finally, Lines 5–7 execute a loop that iterates n_2 times, which is a function of n. A swap bit b is computed with an oblivious comparison, for which $\mathcal{S}$ produces the memory and instruction sequences. Then an OSwap call is performed on a fixed pair of items in D using b, and so $\mathcal{S}$ produces its traces as well, which are deterministic and input-independent.

$\mathcal{S}$ thereby outputs memory and instruction traces identical to those of ORCompact, and they are deterministic. ORCompact is therefore fully oblivious to its inputs. □

4.6.2 Correctness

Theorem 1 (from Section 4.3.2) states the correctness of ORCompact (Figure 4.3). To prove it, we first prove Lemma 2, which states that OROffCompact (Figure 4.2) compacts marked items and maintains their relative order. Recall that OROffCompact only operates on inputs with power-of-two size.

Lemma 2. *Let $w = \sum_i M_i$, $n = |D|$, and $(D'_0, \ldots, D'_{w-1})$ be the subsequence of D consisting of all w items marked in M. When $n = 2^k$, $k \in \mathbb{N}$, OROffCompact(D, M, z) rearranges the items in D such that $(D_z, \ldots, D_{(z+w-1) \bmod n}) = (D'_0, \ldots, D'_{w-1})$.*

Proof. If $n < 2$, OROffCompact correctly does nothing to D. If $n = 2$, then the only possible rearrangement of D is to swap its two items. By a case analysis of the possible values of z, M_0, and M_1, we can see that OROffCompact performs a swap iff doing so would compact the marked items to offset z and maintain their relative order. (Line 3).

If $n > 2$, OROffCompact calls itself on the left and right halves of its inputs. Because n is a power of two, $n/2$ is also. We can therefore recursively assume that these calls correctly compact each half to the specified offsets while preserving order of marked items. We can show the rest of the algorithm (Lines 7–10) yields a correct output by a case analysis of z and $m = \sum_{i=0}^{n/2-1} M_i$:

1) $z < n/2$, $z + m < n/2$: In this case, $s = 0$ (Line 7). Swapping pairs does not begin until position $i = z + m$ (Line 9). Before then, the m marked items in the left half have already been compacted in order to the offset of z with the first call to OROffCompact

(Line 5). Starting then, the marked items get swapped from the right to the left side in order, up until $i = n/2 - 1$, because those marked items were compacted in order to the offset of $z + m$ with the second call to ORO$_{\text{FF}}$C$_{\text{OMPACT}}$ (Line 6). Any marked items remaining on the right side already wrapped to start at $i = n/2$, which completes the order-preserving compaction.

2) $z < n/2$, $z + m \geq n/2$: In this case, $s = 1$. Swapping pairs occurs from positions 0 to $(z + m) \bmod (n/2) - 1$ inclusive. (Nothing is swapped if $(z + m) \bmod (n/2) = 0$.) The marked items on the left side were recursively compacted in order to position z (Line 5). The swapping moves the $m+z-n/2$ marked items that wrapped around to position 0 to instead continue into position $n/2$, producing the correct ordering. These are followed by the marked items in the right side, which were already recursively compacted in order to start at position $z + m$ (Line 6). Any marked items that wrapped around during the right-side compaction, of which there were at most the offset value of $(z+m) \bmod (n/2)$, were moved to instead wrap around the entire array (i.e., starting at position 0) during the pairwise swaps of the first $(z + m) \bmod (n/2)$ items, which completes the in-order compaction.

3) $z \geq n/2$, $(z \bmod (n/2)) + m < n/2$: In this case, $s = 1$. Swapping pairs thus occurs from $i = 0$ to $i = (z + m) \bmod (n/2) - 1$. For the m positions starting at offset z, the swapping is active and swaps marked items on the left to the right, preserving their order, where the left items were marked due to the compaction to $z \bmod (n/2)$ (Line 5). For the subsequent positions from $z + m$ to $n - 1$, swapping does not act on them, and a prefix of in-order marked items may exist due to the recursive right-side compaction. For the next positions (wrapping around), 0 to $z \bmod (n/2) - 1$, swapping is active, and a prefix of in-order marked items from the right side is moved to the left only if the marked items on the right side continued up to $n - 1$. For the next positions $z \bmod (n/2)$ to $(z \bmod (n/2)) + m - 1$, swapping was applied, and similarly a prefix of in-order marked items from right side was moved to the left only if the compacted marked items on the right side wrapped around up to z. For the positions $z \bmod (n/2) + m$ to $n/2 - 1$, the positions are unmarked and were unswapped, and similarly the remaining positions $n/2$ to $z - 1$ are unmarked but were swapped from the left side.

4) $z \geq n/2$, $(z \bmod (n/2)) + m \geq n/2$: In this case, $s = 0$. Swapping pairs does not begin until position $i = (z+m) \bmod (n/2)$. For the positions from z to $n-1$, a sequence of in-order marked items is swapped in order from the left side due to its compaction to offset $z \bmod (n/2)$ (Line 5). For the subsequent (after wrapping around) positions 0 to $(z + m) \bmod (n/2) - 1$, they all contain in-orde marked items left over from the left side compaction, as the swapping did not act in that region. The following items from position $(z + m) \bmod (n/2)$ to $z \bmod (n/2) - 1$ (if any) may contain a in-order prefix

of marked items that were compacted on the right side and then swapped in order. The next items from position $z \bmod (n/2)$ to $n/2 - 1$ potentially continue this sequence of ino-rder marked items as this region was also swapped in order from the right side after it was compacted. The subsequent items from $n/2$ to $n/2 + ((z + m) \bmod (n/2)) - 1$ continue any preceding sequence of marked items if the right-side compacted region wrapped around, as this sequence was not swapped, which maintains the original order of the right-side marked items. Finally, the positions from $n/2 + ((z + m) \bmod (n/2))$ to $z - 1$ (if any) must be unmarked because they were swapped in from the left side and follow the sequence of m marked items on the left (wrapping around) due to left-side compaction to $z \bmod (n/2)$. □

Using Lemma 2, we now prove Theorem 1.

Theorem 1. Given D and M, let $(D'_0, \ldots, D'_{w-1})$ be the subsequence of D consisting of all w items marked in M. ORCompact(D, M) rearranges the items in D such that $(D_0, \ldots, D_{w-1}) = (D'_0, \ldots, D'_{w-1})$.

Proof. If $n = 1$ or $n = 2$, then ORCompact calls OROffCompact with an offset of 0, which by Lemma 2 yields the correct result.

If $n > 2$, then the input arrays are divided into a left side of length $n_2 = n - 2^{\lfloor \log_2(n) \rfloor}$ and a right side of length $n_1 = 2^{\lfloor \log_2(n) \rfloor}$, which is the longest power-of-two-sized suffix. OROffCompact is called on right side with an offset of $(n_1 - n_2 + m) \bmod n_1$, where $m \leftarrow \sum_{i=0}^{n_2-1} M_i$ (Line 4), and by Lemma 2 this call compacts the marked items in that region to that offset while maintaining their relative order. ORCompact is then called on the left side (Line 3), and we can recursively assume that it compacts the marked items in that region while mainting their order. Thus, after the pairwise swaps of the last $n_2 - m$ items of each side (Lines 5–7), all marked items on the right up to the $(n_2 - m)$th one have been swapped in order from into the positions following the m marked items compacted into the first m positions on the left. These marked items are then followed by a prefix of marked items in order among the next $n_1 - n_2 + m$ positions, due to the right-side compaction to offset $(n_1 - n_2 + m) \bmod n_1$, which are then followed by the unmarked items. This process results in a compaction to the beginning of D, it maintains the order within swapped subsequences, and it and places the right-side items to follow in order of the left-side items, proving the the theorem. □

4.6.3 Efficiency

Theorem 2 (from Section 4.3.2) shows the asymptotic runtime of ORCompact (Figure 4.3).

Theorem 2. ORCompact runs in time $O(n \log n)$.

Proof. First, assume that $n = 2^k$, $k \in \mathbb{N}$. Let the execution time of OROffCompact be $T_1(n)$. OROffCompact takes linear time to perform all operations except for the two recursive calls. Those recursive calls are each given half of the original input as input, and recursion ends at $n = 2$. Therefore there exists some c such that $T_1(n) \leq cn + 2T_1(n/2) = \sum_{i=1}^{\log_2 n} cn = cn \log_2 n$.

Next, consider arbitrary n. Let $T_2(n)$ be the execution time of ORCompact. ORCompact first performs at most $c'(n_2 + 1)$ operations for some c', next calls OROffCompact on the first n_1 items, and finally calls itself on the remaining n_2 items (if any). Therefore, for n a power of two, by the previous argument for T_1 and the fact that $n_2 < n$, we have that $T_2(n) \leq c'n + cn \log_2 n$. For n not a power of two, inductively assume that, for $n' < n$, $T_2(n') \leq c'n' + cn' \log_2 n'$. Then

$$\begin{aligned} T_2(n) &\leq c'(n_2 + 1) + T_1(n_1) + T_2(n_2) && (4.1)\\ &\leq c'(2n_2 + 1) + cn_1 \log_2 (n_1) + cn_2 \log_2 (n_2) && (4.2)\\ &\leq c'(2n_2 + 1) + cn \log_2 ((n_1/n)n_1 + (n_2/n)n_2) && (4.3)\\ &\leq c'n + cn \log_2 n, && (4.4) \end{aligned}$$

where Line 4.2 follows from the inductive assumption because n is not a power of two; Line 4.3 follows by applying Jensen's inequality and using the concavity of $\log_2$, noting that $n = n_1 + n_2$; and Line 4.4 holds because i) $2n_2 + 1 \leq n$, ii) $(n_1/n)n_1 + (n_2/n)n_2$ is a convex combination of n_1 and n_2, iii) $n_2 < n_1 \leq n$, and iv) $\log_2$ is an increasing function. □

Theorem 3 (from Section 4.3.2) bounds the number of OSwap calls made in ORCompact.

Theorem 3. Let $S_1(n)$ count the OSwap calls in ORCompact. Then

$$\left(2^{\lfloor \log_2 n \rfloor}/2\right) \lfloor \log_2 n \rfloor \leq S_1(n) \leq \lfloor (n/2) \log_2 n \rfloor.$$

Proof. Let $b(n) \in \{0, 1\}^*$ be the binary expansion of n: $n = \sum_{i=0}^{\lfloor \log_2(n) \rfloor} b(n)_i 2^i$. We first prove that

$$S_1(n) = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b(n)_i \left(2^{i-1} i + \left(n \bmod 2^i\right)\right). \tag{4.5}$$

This formula will lead to the desired lower bound, and also shows that the upper bound is tight when n is a power of two.

Let $S'_1(n)$ be the number of oblivious swaps in OROffCompact given an item array of length $n = 2^k, k \in \mathbb{N}$. There are $(n/2)$ OSwap calls (made in Line 3 or 10), and, for $n > 2$, two recursive calls on $(n/2)$ inputs each (Lines 5 and 6). Therefore, $S'_1(2) = 1$, and $S'_1(n) = (n/2) + 2S'_1(n/2)$. Solving this recurrence, we get that $S'_1(n) = (n/2)\log_2(n)$.

When n is a power of two, ORCompact simply calls OROffCompact on its entire input, and so $S_1(n) = S'_1(n)$. However, for arbitrary n, we need to take into account the fact that ORCompact repeatedly reduces n by the largest power of two not greater than n.

Let $n_1(n)$ be the largest power of two not greater than n, $n \in \mathbb{N}$: $n_1(n) = 2^{\lfloor \log_2(n) \rfloor}$, and let $n_2(n)$ be the remaining quantity of n: $n_2(n) = n - n_1(n)$. We denote by $b(n)$ the binary expansion of n. ORCompact calls OROffCompact on the first $n_1(n)$ items (Line 4), calls itself on the remaining $n_2(n)$ items (Line 3), and then calls OSwap $n_2(n)$ times (Line 7). Therefore,

$$
\begin{aligned}
S_1(n) &= n_2(n) + S'_1(n_1(n)) + S_1(n_2(n)) && (4.6)\\
&= n_2(n) + \frac{n_1(n)}{2}\log_2\left(n_1(n)\right) + S_1(n_2(n)) && (4.7)\\
&= \frac{n_1(n)}{2}\log_2\left(n_1(n)\right) + n_2(n) + && (4.8)\\
&\quad \frac{n_1(n_2(n))}{2}\log_2\left(n_1(n_2(n))\right) + n_2(n_2(n)) \ldots \\
&= \sum_{i=0}^{\lfloor \log_2(n) \rfloor} b(n)_i \left(2^{i-1} i + \left(n \bmod 2^i\right)\right). && (4.9)
\end{aligned}
$$

Equation 4.7 follows from substituting the expression we derived for $S'_1(n)$. Equation 4.8 simply repeats this expansion for $S_1(n)$ until n_2 returns 0, at which point S_1 also returns 0. Equation 4.9 follows from the facts that (1) the set of values returned by n_1 is equal to the set of values $\{b(n)_i 2^i\}_{i:b(n)_i=1}$, and (2) the set of values returned by n_2 is equal to the set of values $\{n \bmod 2^i\}_{i:b(n)_1=1}$. This derivation proves the formula for $S_1(n)$ in Equation 4.5.

This formula leads easily to the stated lower bound. We simply observe that $n_1(n) = 2^{\lfloor \log_2(n) \rfloor}$, and therefore the formula for $S_1(n)$ contains the term $(2^{\lfloor \log(n) \rfloor}/2)\lfloor \log_2(n) \rfloor$. All the other terms are non-negative, and so that term is a lower bound on $S_1(n)$.

For n a power of 2, the upper bound (and indeed equality) trivially holds because the sum has just a single term. Now suppose the upper bound holds for all $m < n$. For n not a power of 2, define n_1 and n_2 as above (and taking n as the implicit argument), so that n_1 is

a power of 2, $1 \leq n_2 < n_1$, and $n = n_1 + n_2$. Note that for $0 < x < 1$, $x < \log_2(1+x)$, and so since $0 < n_2/n_1 < 1$, $(\log_2 n_1) + n_2/n_1 < (\log_2 n_1) + \log_2(1 + n_2/n_1) = \log_2(n_1 + n_2)$. Also, $1 + \log_2 n_2 = \log_2(2n_2) < \log_2(n_1 + n_2)$.

Then

$$
\begin{aligned}
S_1(n) &= \frac{n_1}{2}\log_2 n_1 + n_2 + S_1(n_2) && (4.10)\\
&\leq \frac{n_1}{2}\log_2 n_1 + \frac{n_2}{2} + \frac{n_2}{2} + \frac{n_2}{2}\log_2 n_2 && (4.11)\\
&= \frac{n_1}{2}\left((\log_2 n_1) + \frac{n_2}{n_1}\right) + \frac{n_2}{2}(1 + \log_2 n_2) && (4.12)\\
&< \frac{n_1}{2}\log_2(n_1 + n_2) + \frac{n_2}{2}\log_2(n_1 + n_2) && (4.13)\\
&= \frac{n}{2}\log_2 n, && (4.14)
\end{aligned}
$$

where Line 4.10 comes from the definition of ORCompact, Line 4.11 follows from the inductive assumption, and Line 4.13 is obtained from the preceding arguments. Finally, $S_1(n)$ is integral, and so if $S_1(n) \leq (n/2)\log_2 n$ then it must also be that $S_1(n) \leq \lfloor (n/2)\log_2 n \rfloor$.

□

4.7 Conclusion

TEE systems are gaining traction as a candidate to enable large-scale applications and data analytics that maintain user privacy. In parallel, however, we observe ever-growing side-channel attacks against these systems. In this chapter we establish the notion of fully oblivious algorithm designs and implementations for the fundamental primitive of order-preserving tight compaction. Order-preserving tight compaction serves as a building block for several applications that have been proposed in the TEE setting. Our algoriths outperform the state of the art in concrete timing benchmarks as illustrated by our experimental results. We additionally verify that our implementations indeed achieve the most elusive degree of obliviousness, control-flow obliviousness, by instrumenting an assembly checker tool to this end. The more recent side-channel attacks that precisely extract cache granular memory access patterns or fine-grained control flow information of programs executing in a TEE illustrate how much more powerful such attack vectors are than previously understood. It is therefore critical that we design applications that run in TEEs to be fully oblivious lest they be susceptible to these side channels.

Chapter 5

Oblivious Shuffling Algorithms in the Standard and Streaming Models

In this chapter we present our fully oblivious shuffling algorithms ORShuffle and BORPStream. Both these algorithms can be used in the *standard model*, where all the data to shuffle is available up front when a shuffle is requested. BORPStream is also well suited for the *streaming model*, where data items to shuffle trickle into the systems over a larger time period. BORPStream has two variants (V1 and V2), where V1 is optimized for the standard model, and V2 is optimized for the streaming model. Essentially by partially shuffling items as they arrive, BORPStream presents improvements in the *online* time to obliviously shuffle data once the last item to shuffle has arrived.

BORPStream stands for Bucket Oblivious Random Permutation (BORP) and is inspired by the the BORP subprotocol of bucket oblivious sort [ACN+20]. The original BORP protocol is in the client-server model, where the client performs all computations in their local unobservable memory, and uses the server to store freshly encrypted (rerandomized) data during the protocol execution. We first present BORPCompact, a naive adaptation of the BORP protocol to present a fully oblivious shuffle protocol in the TEE setting, but BORPCompact turns out to be quite inefficient. Consequently, we design BORPStream that is much better tailored to the TEE model. While it shares similarity in its name, BORPStream, however, is significantly different from the original BORP protocol, and uses significantly different techniques to achieve its goal as well as for proving its security and correctness.

5.1 ORShuffle: Oblivious Recursive Shuffle

We first present the fast fully oblivious recursive shuffling algorithm ORShuffle, which uses ORCompact as a key subroutine. It takes as input an array D of n items and rearranges the items of D so that they appear in a uniformly random order.

This algorithm has time complexity $O(n \log^2 n)$. This complexity is the same as some efficient sorting networks [Bat68], which can themselves be used for shuffling by attaching random labels to items before sorting based on those labels. However, ORShuffle represents an efficiency improvement because it does not need to attach labels and move them around with the data. The complexity is also higher than the $O(n \log n)$ complexity achieved by other approaches [AKL+20a]. The constants are low, though, and we will show that the performance of ORShuffle exceeds that of other approaches.

5.1.1 Algorithm

ORShuffle follows the high-level approach of the PerfectORP algorithm of Asharov et al. [AKL+18, Algorithm 6.8]. In that algorithm, the array of items to shuffle is split into two halves, each is recursively shuffled, and then the items in each half are randomly interleaved with an Intersperse algorithm. Intersperse effectively performs a "reverse" compaction, sometimes called "expansion" [Goo11]. We can therefore use ORCompact directly by reversing these steps (alternatively, running ORCompact in reverse itself would yield an expansion algorithm). That is, we randomly separate the items into two halves using ORCompact and then recursively shuffle each half. The result is the fully oblivious shuffle algorithm ORShuffle. Unlike PerfectORP, ORShuffle is a practically efficient algorithm because of its use of ORCompact in place of Intersperse which is theoretic in nature and has high underlying constants.

The ORShuffle algorithm is given in Figure 5.1. It uses the subroutine MarkHalf(n) (Figure 5.2) to mark a random half of the n positions, which is also essentially given by Asharov et al. [AKL+18, Algorithm 6.1]. ORShuffle rearranges the items in D into a uniformly random order. It is fully oblivious to the values in D, and it is also fully oblivious to its random bits and therefore to the permutation it applies. ORShuffle is not oblivious to the size or location of D. Its correctness and obliviousness follow from the correctness and obliviousness of ORCompact, using arguments similar to those of Asharov et al. [AKL+20a] regarding PerfectORP.

Figure 5.1: ORShuffle(D): Put items in D in uniformly random order.

1: $n \leftarrow |D|$
2: **if** $n = 2$ **then**
3: Generate $b \in \{0, 1\}$ uniformly at random.
4: OSwap(D_0, D_1, b)
5: **else if** $n > 2$ **then**
6: $M \leftarrow$ MarkHalf(n)
7: ORCompact(D, M)
8: ORShuffle($D_{0..\lceil n/2 \rceil - 1}$)
9: ORShuffle($D_{\lceil n/2 \rceil..n-1}$)

Figure 5.2: MarkHalf(n): Mark random half of boolean array of length n.

1: Create boolean array M of length n.
2: $\ell \leftarrow \lceil n/2 \rceil$
3: **for** $i \leftarrow 0, \ldots, n-1$ **do**
4: Generate $r \in [0, 1)$ uniformly at random.
5: $M_i \leftarrow [\![r < \ell/(n-i)]\!]$
6: $\ell \leftarrow \ell - M_i$
7: **return** M

5.1.2 Efficiency

The asymptotic runtime of ORShuffle is given in Theorem 5. The proof for it appears later in Section 5.5.

Theorem 5. *ORShuffle runs in time* $O(n \log^2 n)$.

While this runtime is not asymptotically optimal (oblivious shuffling algorithms exist with runtime $O(n \log n)$), the constants make it practically efficient. Theorem 6 gives an exact count of the number of OSwap calls. These oblivious swaps are relatively expensive operations, and, as with ORCompact, the data items themselves are only accessed via these swaps, and so the swaps dominate the runtime for large enough items.

Theorem 6. *Let* $S_2(n)$ *count the number of OSwap calls performed by ORShuffle*(D)*, with* $|D| = n$*. For* $n = 2^k, k \in \mathbb{N}$*,* $S_2(n) = (n/4)(\log_2 n + 1)\log_2 n$*. For all* n*,* $S_2(n) < (n/4)(\log_2 n + 1)\log_2 n + n/(6 \ln 2)$*, and* $S_2(n) \geq \left(2^{\lfloor \log_2 n \rfloor}/4\right)(\lfloor \log_2 n \rfloor + 1)\lfloor \log_2 n \rfloor$.

This shows that ORShuffle performs roughly $(n/4)((\log_2 n) + 1)\log_2 n$ oblivious swaps. We can compare this to the oblivious swaps performed using a sorting network. A sorting network performs pairwise compare-and-swap operations in a fixed sequence and thus can be used for oblivious shuffling by attaching random labels to items. Using the bitonic sorting network [Bat68] is the common choice for existing TEE systems that require oblivious sorting [KKS20, ZDB+17, EZ19, ASC21]. The odd-even mergesort network [Bat68] is slightly smaller but experimentally performs worse due to poor memory locality. The bitonic sorting network also requires $(n/4)((\log_2 n) + 1)\log_2 n$ oblivious swaps. Therefore, for large data items, we can expect that ORShuffle will perform similarly to Bitonic Shuffle. However, for smaller items, ORShuffle can yield an efficiency improvement because random labels need not be added and included in the oblivious swaps. Indeed, for word-sized items, random labels can approximately double the cost of the swaps.

The memory used by ORShuffle is little more than the memory already occupied by the data items. The data items are swapped in place. At most an additional n bits are created to mark items for compaction, and a small constant amount of memory is needed for each of the $\lceil \log_2 n \rceil$ levels of recursion. We also note that the algorithm can benefit significantly from memory caching and prefetching due to its regular memory-access patterns and good locality.

Figure 5.3: ORSPar(D): In parallel, put items in D in uniformly random order.

```
1: n ← |D|
2: if n = 2 then
3:    Generate b ∈ {0, 1} uniformly at random.
4:    OSwap(D_0, D_1, b)
5: else if n > 2 then
6:    M ← MarkPar(⌈n/2⌉, n)
7:    ORCPar(D, M, ∅)
8:    do in parallel
9:       ORSPar(D_{0..⌈n/2⌉−1})
10:      ORSPar(D_{⌈n/2⌉..n−1})
```

Figure 5.4: MarkPar(m, n): In parallel, mark a random m of n positions.

```
1: if n = 1 then
2:    return m
3: else if n > 1
4:    n_1 ← ⌈n/2⌉, n_2 ← n − n_1
5:    k ← HyperGeomPar(n, n_1, m)
6:    do in parallel
7:       M_1 ← MarkPar(k, n_1)
8:       M_2 ← MarkPar(m − k, n_2)
9:    return M_1||M_2
```

5.1.3 Parallelization

We give a parallel version of ORShuffle in Figure 5.3, again in the EREW PRAM model. This algorithm, ORSPar, is nearly identical to the sequential version except that it calls parallel versions of the subroutines, namely MarkPar and ORCPar, and makes its recursive calls in parallel.

MarkPar, given in Figure 5.4, marks a random m of n locations. This algorithm is a variant of the algorithm given by Sanders et al. [SLHS+18, Algorithm P]. The differences from the original algorithm are that it is fully specified (i.e., no unspecified "local sampler") and outputs a full boolean array of length n (i.e., not only the marked index values). These changes are needed to guarantee full obliviousness with respect to m and to the random bits.

MarkPar uses HyperGeomPar for hypergeometric sampling. HyperGeomPar(n, d, m)

samples the number of successes among d random draws without replacement from a set of n items that contain m total successes. Following Sanders et al. [SLHS+18], we perform hypergeometric sampling with an algorithm of Stadlober [Sta90]. We use an oblivious implementation that uses the HRUE^t variant, computes $\ln(x!)$ with a Stirling approximation, omits the fast-acceptance optimization, and repeats the rejection sampling a constant number of times to obtain a given failure probability.

The work of ORSPar is similar to that of ORShuffle. The only difference in the operations executed is from calling ORCPar instead of ORCompact and MarkPar instead of MarkHalf. As discussed in Section 4.3, ORCPar has the same $O(n \log n)$ complexity as ORCompact and the same number of OSwap calls. MarkPar has the same $O(n)$ complexity as MarkHalf given a fixed failure probability. Therefore, ORSPar has the same $O(n \log^2 n)$ asymptotic runtime as ORShuffle. Moreover, the number of OSwap calls is the same in both.

Parallelization does improve the step complexity of shuffling. ORSPar takes $O(\log^2 n)$ parallel steps, and it has $(\lceil \log_2 n \rceil + 1)\ \lceil \log_2(n) \rceil / 2$ parallel steps that include an OSwap call. These results show that ORSPar is appropriate for large-scale private data analysis, where many processors might be available for fast processing.

5.2 Bucket Oblivious Random Permutation (BORP)

We next introduce the BORPStream shuffling algorithm, which is inspired by the Bucket Oblivious Random Permutation (BORP) of Asharov et al. [ACN+20]; BORPStream leverages entirely different machinery than BORP to efficiently shuffle while attaining full obliviousness, and reuses just the underlying butterfly routing network (BRN) used by BORP. BORP is a promising alternative to ORShuffle because of its small constants, and indeed its authors present it as a faster alternative to Bitonic Shuffle. However, it is given in a client-server setting where the client has private memory. Oblivious sorting or compaction are suggested as ways to avoid needing private memory, but using them increases the complexity in n, and, as our experiments will show, this approach yields poor shuffling performance (despite using the new and faster ORCompact).

At a high level, Asharov et al. [ACN+20] randomly permute items by passing them through a well-parameterised BRN. The routing network is comprised of buckets, and each of the buckets in the initial layer of the network contain exactly half real data elements, and half dummy elements. First the algorithm partitions the n input items across $\mathsf{B} = \frac{2n}{\mathsf{Z}}$ buckets, with each bucket containing Z elements. The constant Z dictates

the correctness guarantee of BORP, and the authors recommend Z=512. These initial-layer buckets contain exactly half real items and half dummy items. Real items get assigned a random destination from among the B output buckets, while the dummies are assigned $\perp$ as their destination. The butterfly routing network of depth $\log_2 \mathsf{B}$ is then used to route all the real items to their destination buckets obliviously. In order to do so, the algorithm performs a *MergeSplit* operation on the $\mathsf{B}/2$ pairs of buckets that are 2^i apart in layer i. The MergeSplit operation distributes the real items of both its input buckets into two output buckets based on the ith bit of their destination label, and pads each output bucket with enough dummy items to make exactly Z in each output bucket. These output buckets are then locally shuffled by the client to randomize the order of items before returning them to the server. Once the routing is completed, the dummy items from each output bucket are discarded to obtain the randomly permuted real items.

Executing BORP naively within a TEE would break its security guarantees. During the MergeSplit operation or the final discard of dummies, an adversary could use observations of the memory access patterns to distinguish real and dummy items. The straightforward solution is to replace the private-memory computations with oblivious variants (largely by changing MergeSplit to use ORCompact). We call this algorithm BORPCompact, which we present in Section 5.2.1. It is unfortunately not competitive with the state of the art, as we will see in Section 5.3. We therefore design BORPStream, which operates in a very different way: by pushing items through the BORP BRN one by one. Not only does this yield much better efficiency overall, it allows much of the work to happen while items are being input in a *streaming* manner.

5.2.1 BORPCompact

BORPCompact is almost identical to the original BORP. The difference is that the MergeSplit operation is instantiated with our ORCompact(D, M). To produce the correct bit array M used by ORCompact, the algorithm makes two linear passes over D (the concatentated two input buckets going into the ORCompact operation, $|D| = 2\mathsf{Z}$). In the first linear pass it obliviously counts the number of real items going into each of the output buckets. In the second linear pass it marks real items correctly in M, while additionally also marking the right number of dummies by using the counts from the first pass, such that the total number of marked items will be exactly Z. Once all the items have been processed through the BRN as described above, the final output buckets are compacted to just the real items in each of them by leveraging ORCompact again. Furthermore, the real items in each bucket are then obliviously shuffled using ORShuffle, to break any

Figure 5.5: MSN: MergeSplitNode class

```
1:  Internal State Variables: c, buffer[], out_streams[]
2:  Initialize(f, s, b, streams):
3:      initializeBuffer(buffer, s, b)
4:      c ← 0 // c is the current eviction stream ∈ [0, f − 1]
5:      out_streams ← streams // MSNs this node connects to; or final buckets if this is
        a last layer MSN
6:  ProcessItem(p):
7:      for i ← 0, . . . , s − 1 do
8:          e ← extractStream(p) // Assigns ⊥ if p is ⊥
9:          z_1 ← [extractStream(buffer[i])=c]
10:         z_2 ← [[isReal(p)]] ∧ [[e ≠ c]]
11:         z ← z_1 ∨ z_2
12:         OSwap(p, buffer[i], z)
13:     if self.isLastLayerNode then
14:         Append(out_streams[c], p)
15:     else
16:         out_streams[c].ProcessItem(p)
17:     c ← (c + 1) mod f
```

correlations in relative ordering of items within an output bucket. Finally all these items are concatenated together to produce the required output array of randomly permuted items. The correctness argument for this variant is identical to that of BORP [ACN+20], but our variant is fully oblivious and can by design be safely run in a TEE.

5.2.2 BORPStream

BORPStream converts BORP into an algorithm processing "streaming" data that arrives incrementally. At a high level, instead of processing items layer by layer at a bucket-level granularity, we stream each input item as it arrives through the entire butterfly routing network. Each of the discrete MergeSplit operations are replaced with a stateful processing node we call a *MergeSplitNode (MSN)*, described in Figure 5.5. Every MSN in a layer of the network is connected directly to the appropriate MSNs in the next layer that their logical output bucket would have been connected to, effectively building a butterfly routing network of purely MSNs, followed by a layer of B output buckets. To uphold obliviousness in this streamed model, items have to be processed in a deterministic fashion; i.e., routing decisions cannot be based on the destination of the item being routed. Therefore, all MSNs follow a strict round robin schedule for routing data across all of its f output streams.

The schedule is clearly at odds with the requirement of routing items to their correct destination. To ensure correctness, each MSN is imbued with a local buffer of capacity s items. When an MSN receives an item to process it scans across its local buffer and swaps the incoming item with one destined for the current eviction stream using oblivious swaps (Figure 5.5, lines 7–12). Figure 5.6 details BORPStream in its entirety. BORPStream can be split into two phases: i) Phase 1 processes input data packets as they arrive, assigning each element a random destination label $\in [0, B-1]$ and then routing it through the BRN (Lines 9–15); ii) Phase 2 starts once all input data have been received and processed through the BRN.

In scenarios where data items are acquired over time, this two-phase division takes advantage of time otherwise spent waiting for all items to arrive. Several real world use-cases provide such scenarios, including application telemetry, software profiling, or error reporting. Typically such data accumulate over time as they are generated by user activity, and shuffling is performed only after enough users have provided inputs to yield a sufficiently large anonymity set. Later in Section 5.3 we will illustrate the concrete performance improvements obtained by this division by measuring the time taken by Phase 2, which reflects the turnaround time for the shuffle once all data items have arrived.

Figure 5.6: BORPStream(n, b, λ) where n is the number of elements to shuffle, b is the size of each element, and λ is the failure probability parameter.

```
1:  Initialize(n, b, λ):
2:      f, d, s ← BORP_Optimizer(n, b, λ)
3:      B ← f^d // Number of buckets at start/end of BRN
4:      m ← f^(d−1) // Number of MSNs in each layer of BRN
5:      c ← 0 // Number of data elements processed so far
6:      D′ ← [] // Output array to be populated
7:      BRN ← InitializeBRN(f, d, s, b) // InitializeBRN sets up the BRN of MSNs
        defined by the parameters f, d, s, and b, returning a 2D array of MSN nodes,
        BRN: MSN[d][f^(d−1)]
8:
9:  BORPStream_Phase1(p): // Call for each arriving packet p
10:     entry_node ← ⌊mc/n⌋
11:     Generate label ∈ [0, B − 1] uniformly at random
12:     p′ ← ⟨label, p⟩
13:     BRN[0][entry_node].ProcessItem(p′)
14:     BRN[0][entry_node].ProcessItem(⊥) // Send dummy item
15:     c ← c + 1
16:     if c = n then
17:         return BORPStream_Phase2(BRN)
18:
19: BORPStream_Phase2(BRN):
20:     FlushBuffers(BRN)
21:     for i ∈ [0, B − 1] do
22:         M,r ← MarkReal(buckets[i]) // M is a bit array; r ← ∑ M_i; buckets is the
        array of B buckets at the end of the BRN.
23:         ORCompact(buckets[i], M)
24:         ORShuffle(buckets[i]_{0..r−1})
25:         Append(D′, buckets[i]_{0..r−1})
26:     return D′
```

Given the number of items n to shuffle, the size of each item b, and the desired correctness parameter λ, we design and leverage an optimizer that returns the optimal choice of parameters (f: the fan of the network, d: the depth of the network, and s: the internal buffer size of each MSN) for the underlying BRN. The failure probability of the returned set of parameters will be smaller than $2^{-\lambda}$. In our experiments we set $\lambda = 80$. Once the optimal parameters are known, the enclave initializes and sets up the BRN by initializing a grid of MSNs and connecting them appropriately. For every real item, we also propagate a dummy item through the network. The dummy elements are required in order to provide sufficient slack in the internal buffers of the MSN, lest they overflow and trigger a failure. Once all the real and dummy items have been processed through the network, there may still be real items in the MSN local buffers. We therefore additionally flush all the local buffers, one layer at a time. To maintain obliviousness, during the flush operation, each MSN performs $s \cdot f$ evictions, and there are f^{d-1} MSNs in each of the d layers of the BRN, thus resulting in $s \cdot d$ additional items in each of the $B = f^d$ final buckets. We then use ORCompact on the buckets to reduce the buckets down to just the real items. We then shuffle just the real items of each bucket using ORShuffle to reorder them, as real items would otherwise arrive in the final bucket in the order that they entered the network. Finally, we concatenate all the real items in each bucket to arrive at the desired output.

5.2.3 Correctness and Obliviousness

The use of MSNs for BORPStream changes its correctness argument significantly from the original BORP. We are no longer concerned about the possibility of individual bucket overflows through the network, but instead our correctness argument rests on the union bound of the probability of buffer overflows of each of the MSNs' internal buffers, P. We empirically evaluate P via a failure probability calculator that performs a Markov chain analysis.

The original BORP algorithm is fully oblivious from the server's perspective, since the algorithm retrieves and stores buckets in a deterministic fashion, but it requires unobservable client memory. For BORPStream, obliviousness rests on the ProcessItem algorithm (Figure 5.5) of MSN being oblivious. For every incoming item, MSNs perform the same set of operations (Figure 5.5, Lines 7–17). An adversary can only view the memory access patterns for these OSwap operations, which are deterministic and secret independent. The final operations of Phase 2 of BORPStream are ORCompact and ORShuffle, and their obliviousness has already been established. Note that revealing the number of real items in a bucket after the ORCompact step does not violate

obliviousness [ACN+20]. The number of real items in these buckets is an artifact of the distribution of random labels assigned, but it does not reveal any information about which real item ends up in a given bucket. We detail our empirical failure probability calculator and a proof of correctness and obliviousness for BORPStream in depth later in Section 5.5.3.

5.2.4 Efficiency and Parallelization

The efficiency of BORPStream depends on some key parameter choices for the BRN: the fan f, the depth d, and the buffer size s. We use a numerical optimizer to set these as a function of the desired failure probability $2^{-\lambda}$. We can obtain an asymptotic upper bound on the runtime of $O(n \log^2 n)$.

Parallelization for BORPStream is not as immediate due to the two phases. However, both phases in isolation lend themselves to parallelization in a straightforward fashion. Phase 1 can be parallelized by viewing the entire network as a pipeline of OSwap operations. For Phase 1, the number of OSwaps without parallelization is $2 \cdot n \cdot \mathsf{s} \cdot \mathsf{d}$, since each of the $2n$ items (n real and n dummy) have to perform an OSwap comparison against each item of the s sized internal buffer of the d MSNs it passes through. Pipelining this brings down the number of OSWAP steps to $2 \cdot n + \mathsf{s} \cdot \mathsf{d}$. While the speedup from parallelization here is just a small factor $\mathsf{s} \cdot \mathsf{d}$, we additionally note that since Phase 1 only handles incoming data items, and since we envision such packets to arrive haphazardly, the notion of parallelism only arises in the case that items arrive in rapid succession (or even simultaneously), in which case we get the aforementioned step complexity reduction.

Phase 2 is innately more parallelizable; the FlushBuffers component of Phase 2 requires $\mathsf{s} \cdot \mathsf{f}^{\mathsf{d}} \cdot (1 + \mathsf{s} \cdot \frac{\mathsf{d} \cdot (\mathsf{d}-1)}{2})$ OSwaps, which are trivially parallelizable across the $\mathsf{f}^{\mathsf{d}-1}$ MSNs at each layer of the BRN, and can be pipelined along the depth of the network resulting in a parallel step complexity of $\mathsf{d} \cdot \mathsf{f} \cdot \mathsf{s} + \mathsf{s}(\mathsf{d}-1)$. For the remainder of Phase 2 all the $\mathsf{B} = \mathsf{f}^{\mathsf{d}}$ buckets can be individually computed upon in parallel. Each ORCompact takes $\frac{V}{2} \log V$ OSwaps where $V = \frac{2n}{\mathsf{B}} + \mathsf{s} \cdot \mathsf{d}$ is the number of items in each bucket at the end of Phase 1, and each ORShuffle takes about $\frac{\mathsf{r}}{4} \log^2 \mathsf{r}$ OSwaps, where $\mathsf{r} = \frac{n}{\mathsf{B}}$ is the expected number of real items in each bucket.

In addition, ORCPar can be leveraged to parallelize the compaction within each bucket, and the final counts of real items in each bucket can be established after the PrefixSumPar function is invoked within ORCPar each to ensure that writes are done to the correct positions in the output array in parallel. This can be done by executing

PrefixSumPar on an array of B size, populated with the sum of the outputs of PrefixSumPar of each individual bucket; i.e., the number of real items in each bucket. Additionally, the recursive shuffles of real items in each of the B buckets are also completely parallelizable, and even parallelizable within a bucket by leveraging ORSPar. The final parallel step complexity of the compaction and shuffle then boils down to $O(\log V + \max(\log \mathsf{B}, \log^2 \mathsf{r}))$.

The $\max$ term arises from the fact that depending on the underlying BRN configuration, the dominant term could either be $\log \mathsf{B}$ or $\log^2 \mathsf{r}$, from running (in parallel) PrefixSumPar on the number of real items in each bucket and ORSPar within each bucket. It is safe to execute these operations in parallel because the locations of where to write the real items in each bucket need to be known only by the end of the shuffle itself.

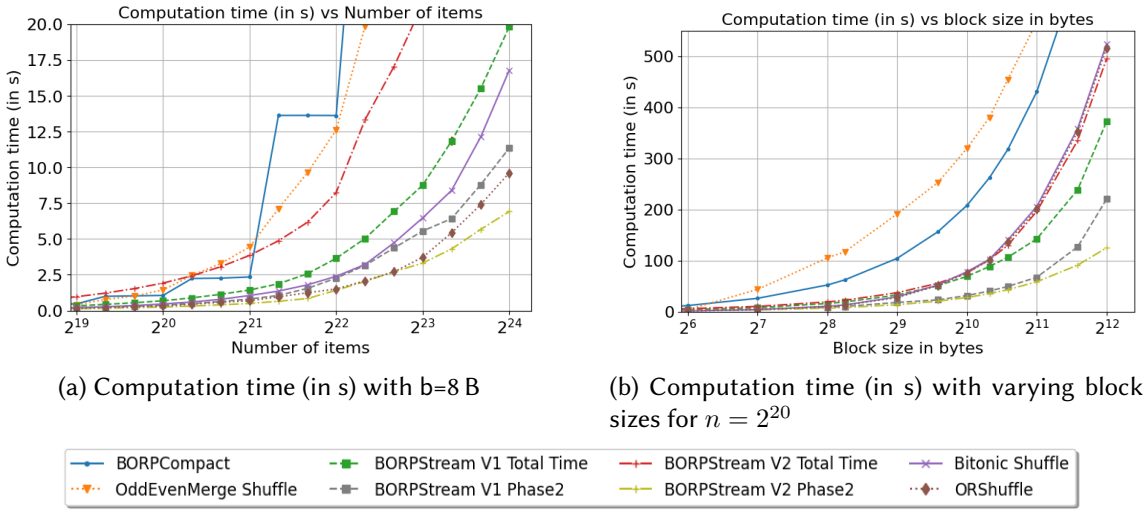
5.3 Implementation

We implement and benchmark our algorithms on a server-grade Intel Xeon E3-1270 running Ubuntu 20.04 with four physical cores, 64 GB of DDR4 RAM, and support for Intel SGX; the same setup as that of our experiments in Section 4.5. Our implementation is open-source and publicly available [SJG22b].

We implement and evaluate the performance of the three fully oblivious shuffling algorithms presented in previous sections: ORShuffle, BORPCompact, and BORPStream. We also implement and evaluate fully oblivious versions of shuffling via OddEvenMerge sort and Bitonic sort [Bat68]. Figure 5.7a shows the performance results for various problem sizes n, with a fixed block size of $\mathsf{b} = 8$.

As mentioned in Section 5.1, shuffles based on sorting networks, such as Bitonic Shuffle, require attaching additional random tags to the data elements. In Figure 5.7a, we see the concrete overheads of these approaches when compared to our ORShuffle algorithm. For instance, with $n = 2^{24}$ from Figure 5.7a we see that ORShuffle takes 9.556 ± 0.005 s, while Bitonic Shuffle takes 16.78 ± 0.03 s, the performance gap widening with increasing problem size. We note that the stddevs for all our algorithm implementations are small, by virtue of their oblivious design.

Figure 5.7a has four lines for BORPStream. The "V1" variants correspond to the timings (total and just for Phase 2) when the BORPStream optimizer is tuned to reduce the total time taken. However, as we alluded in Section 5.2.2, in the context where the data to be shuffled arrives intermittently, one could tune the optimizer to reduce the



(a) Computation time (in s) with b=8 B

(b) Computation time (in s) with varying block sizes for $n = 2^{20}$

Figure 5.7: Comparison of shuffling algorithms in computation time (in s) over varying problem sizes. Error bars are plotted in the graph, but are too small to be seen.

Phase 2 time instead ("V2"), and this is in fact the setting where BORPStream shines. For instance, with $n = 2^{24}$ when tuned to minimize total time (optimizer parameters f=4, d=1, s=48), we see that it takes a total time of 19.37±0.03 s, of which 11.14 ±0.03s is spent on Phase 2, implying a per-packet cost of approximately 0.49 μs as packets arrive. When tuned to minimize the time taken in Phase 2 (optimizer parameters f=4, d=5, s=49), BORPStream takes a total time of 42.42 ±0.05s, however only requires 6.784±0.002 s for Phase 2, implying a per packet cost of approximately 2.1 μs. While this is larger than that from the V1 variant, we note that this cost is still quite small in practice. In this example, if the server is likely to have 2.1 μs of otherwise idle time on average after each user submits their data, it would behoove it to use the V2-optimized BORPStream algorithm, so that once the final data arrives, the shuffle can be completed more quickly in order to reduce the latency of the overall protocol.

Finally, Figure 5.7b illustrates the overheads of the various oblivious shuffling algorithms with increasing block sizes, while maintaining a fixed problem size of $n = 2^{20}$. The advantage of BORPStream overall is clear here; for instance, for $n = 2^{20}$ 4 KiB blocks, ORShuffle and Bitonic Shuffle cost 515.2 ±0.3s and 523.9±0.3 s respectively, while BORPStream V1 (optimizer parameters f=4, d=3, s=47) and V2 (optimizer parameters f=4, d=4, s=47) cost 371.8 ±0.8 s and 494.7 ±0.8 s respectively, a 1.4× improvement over the state of the art. Furthermore, BORPStream V2's Phase 2 cost is 124.7±0.1 s

which is a 4.2× improvement over the state of the art and ORShuffle. We observe performance benefits for BORPStream V1 even though it requires about 3× more OSwaps than Bitonic Shuffle; this is because BORPStream has significantly better memory locality than Bitonic Shuffle, as all the memory required for the MSNs in the first phase of BORPStream even at such large problem sizes can fit within the 90 MB PRM limit, avoiding expensive PRM paging overheads for the majority of the algorithm. The second half of BORPStream is also locality efficient as it handles items one bucket at a time. BORPStream V2 has the same memory advantages as mentioned earlier, although it incurs more than 4× the number of OSwaps in total than Bitonic Shuffle. However in Phase 2 alone, BORPStream V2 only incurs about half the number of OSwaps as Bitonic Shuffle does in total, and these OSwaps again have better locality than that of Bitonic Shuffle as the OSwaps are all contained within a single bucket, while in Bitonic Shuffle they are distributed over all the items in the entire problem. Additionally, in our evaluated range of block sizes the performance gap clearly widens as the block sizes get larger. While comparing just the Phase 2 cost of BORPStream with that of the entire ORShuffle/Bitonic Shuffle may seem unfair, note that the latter shuffles can only be performed once all the data elements are available. Therefore, ignoring the per-item timing costs of BORPStream during Phase 1 may be a fair comparison for scenarios in which data items trickle in over a relatively long time period.

To summarize, our results show that ORShuffle consistently outperforms the fastest existing fully oblivious shuffle, Bitonic Shuffle, for all problem sizes. For small block sizes, ORShuffle provides a 1.8× improvement, and BORPStream's two-phase division provides a 2.5× improvement over Bitonic Shuffle once all the packets have arrived. As block sizes get larger, ORShuffle gives only minor performance improvements over Bitonic Shuffle, but BORPStream provides a 1.4× speedup, and with the two-phase division it provides a 4.2× improvement once all the packets have arrived. Despite recent work on practical oblivious shuffling, which has targeted varying levels of partial obliviousness [OGTU14, BEM+17, ACN+20], this work demonstrates the first practical improvements in *fully* oblivious shuffling over the classic Bitonic Shuffle.

5.4 Related Work

5.4.1 Oblivious Shuffling

Melbourne Shuffle [OGTU14] is a data-oblivious shuffle designed for cloud storage with minimal client storage overheads. Its optimized (recursive) variant requires $O(\sqrt[c]{n})$ client

or private memory to store and permute items, and this private memory needs to be obscure to the server. To instantiate Melbourne Shuffle in a TEE, one would have to convert the computation that happens within this client memory to be fully oblivious. This computation however is non-trivial to convert into a fully oblivious variant; it requires obliviously distributing $\sqrt[c]{n}$ items across $\sqrt[c]{n}$ logical buckets (that reside on the server), as well as padding them up to $p \cdot \sqrt[c]{n}$ items for each of those buckets, where the constant p dictates the failure probability of the algorithm. The algorithm fails if the shuffle permutation requires moving more than $p \cdot \sqrt[c]{n}$ items to any of the destination buckets. The algorithm induces $O((c-1)(n^{\frac{c-1}{c}}))$ such client-side operations over $n^{1/c}$ items. One could convert this into a fully oblivious variant in quadratic time over the $n^{1/c}$ items it operates on, which leads to an algorithm of $O((c-1)(n^{\frac{c+1}{c}}))$ complexity, asymptotically worse than the ones we proposed. Additionally, there are constants missing here that further blow up the cost of implementing this in practice within a TEE.

Stash Shuffle, the shuffling algorithm in Prochlo [BEM+17], was designed to be efficient for the TEE setting. The shuffle is similar to a Melbourne shuffle without the recursion, but it also uses a stash of size $O(\sqrt{n})$ to hold items that cannot be distributed in a given round to their intended output buckets. Stash Shuffle is designed to reduce the number of times a single item has to be processed by the shuffler, which in classical oblivious shuffling algorithms is $\log^2 n$; their work reduces this down to a constant factor. Their construction succeeds in their desired efficiency goal only in the setting where access patterns within PRM are safe from side channels, but as we detail in Section 2.2, this is no longer a realistic assumption.

Sorting networks with random labels are a natural approach for oblivious shuffling. They are a common tool for oblivious sorting inside TEE systems [KKS20, ZDB+17, EZ19, ASC21]. The bitonic network [Bat68] is preferred because it performs better in practice [Ebb15], even though the odd-even mergesort network [Bat68] is slightly smaller. Probabilistic sorting networks [LMS97] can achieve smaller size at the cost of failing to sort correctly with certain probability.

5.4.2 Systems Using TEEs and Oblivious Algorithms

Several privacy-preserving systems have been proposed that make use of TEEs and oblivious algorithms.

The Oblix system [MPC+18] provides secure search of encrypted data using a TEE. Oblix uses "doubly oblivious" algorithms, in which memory accesses are oblivious to both protected memory and external memory. The definition of double obliviousness

does not include the control flow, in contrast to full obliviousness. The authors note that the Oblix implementation does not guarantee obliviousness after compilation, an issue we address with FOAV.

Opaque [ZDB+17] is a TEE-based platform for secure data analytics. It performs SQL-type queries on tables, such as filter and join. Opaque assumes the existence of some memory where accesses are not observable to the adversary, and it does not provide control-flow obliviousness. Opaque performs compaction to filter unwanted records and to remove dummy records created during oblivious processing, and it could benefit from our oblivious compaction algorithm ORCompact.

Prochlo [BEM+17] provides privacy-preserving telemetry, for example to collect information on how a web browser is used. The Prochlo architecture includes shuffling as a major component to help anonymize user data. Prochlo introduces the aforementioned Stash Shuffle algorithm, which is external-memory oblivious. To obtain security against memory and timing side channels, Prochlo could instead use our ORShuffle algorithm. Alternatively, Prochlo could replace the algorithms it uses to process data in protected memory with oblivious versions. These algorithms accomplish shuffling and compaction tasks, and so our algorithms could be used.

Krastnikov et al. [KKS20] give oblivious algorithms for database joins. A key operation uses a reverse compaction operation (called "distribution"), for which they need order preservation. They use Goodrich compaction in reverse, and ORCompact in reverse could be used instead to improve performance.

Ohrimenko et al. [OSF+16] propose to use TEEs for privacy-preserving machine learning. They give several machine-learning algorithms in which all memory accesses are data-oblivious. Several of their algorithms use shuffling as a key component, including to randomize the training samples for SVMs and neural networks and to randomize initial clusters for k-means clustering. These algorithms could use ORShuffle or BORPStream to improve performance.

Scramble-then-Compute (StC) [DDCO17] makes certain non-oblivious algorithms oblivious by preceding them with an oblivious shuffle. StC can eliminate memory side channels within TEEs for otherwise non-oblivious algorithms for problems including compaction, sorting, and selection. StC is described and evaluated using the Melbourne shuffle [OGTU14], which only provides external-memory obliviousness. ORShuffle or BORPStream could be used instead in StC to provide fully oblivious algorithms for the same problems.

5.5 Proofs of Correctness and Efficiency

5.5.1 Efficiency of ORShuffle

Theorem 5 (from Section 5.1.2) gives the asymptotic runtime of ORShuffle (Figure 5.1).

Theorem 5. ORShuffle runs in time $O(n \log^2 n)$.

Proof. Let $T_3(n)$ be the runtime of ORShuffle. MarkHalf takes linear time, and by Theorem 2 ORCompact takes time $O(n \log n)$. Therefore, $T_3(n) \leq c(n \log n) + 2T_3(n/2)$ for some $c > 0$. By expanding this recurrence and simplifying the arithmetic sum, we can see that $T_3(n) \leq cn \log_2 n((\log_2 n) + 1)/2 = O(n \log^2 n)$. □

Theorem 6 (from Section 5.1.2) bounds the number of OSwap calls made in ORShuffle.

Theorem 6. Let $S_2(n)$ count the number of OSwap calls performed by ORShuffle(D), with $|D| = n$. For $n = 2^k, k \in \mathbb{N}$, $S_2(n) = (n/4)(\log_2 n + 1) \log_2 n$. For all n, $S_2(n) < (n/4)(\log_2 n + 1) \log_2 n + n/(6 \ln 2)$, and $S_2(n) \geq \left(2^{\lfloor \log_2 n \rfloor}/4\right)(\lfloor \log_2 n \rfloor + 1)\lfloor \log_2 n \rfloor$.

Proof. Let $S_1(n)$ be the number of OSwap calls for ORCompact; recall from Theorem 3 that $S_1(n) \leq \lfloor (n/2) \log_2 n \rfloor$. The number of OSwap calls satisfies the recurrence $S_2(n) = S_1(n) + S_2(\lceil n/2 \rceil) + S_2(\lfloor n/2 \rfloor)$.

Suppose that $n = 2^k, k \in \mathbb{N}$. Inductively assume that, for all m that are powers of 2 less than n, $S_2(m) = (m/4)(\log_2 m + 1) \log_2 m$. Observe that this assumption holds for $n = 1$ ($S_2(1) = 0$) and $n = 2$ ($S_2(2) = 1$). Then it holds that

$$S_2(n) = \left(\frac{n}{2}\right) \log_2 n + 2S_2\left(\frac{n}{2}\right) \tag{5.1}$$

$$= \left(\frac{n}{2}\right) \log_2 n + \frac{n}{4} \log_2 \left(\frac{n}{2}\right) \left(\log_2 \left(\frac{n}{2}\right) + 1\right) \tag{5.2}$$

$$= \left(\frac{n}{4}\right) \log_2 n(\log_2 n + 1), \tag{5.3}$$

where Line 5.1 follows by replacing $S_1(n)$ using the formula given in Theorem 3 (noting the upper and lower bounds match for powers of 2), and Line 5.2 holds by the inductive assumption.

To prove the lower bound, note that by the preceding argument it holds with equality for n a power of two. For n not a power of two, inductively assume that, for $m < n$, $S_2(m) \geq \left(2^{\lfloor \log_2 m \rfloor}/4\right)(\lfloor \log_2 m \rfloor + 1)\lfloor \log_2 m \rfloor$. Let $n_1 = 2^{\lfloor \log_2 n \rfloor}$. Then it follows that

$$\begin{aligned}
S_2(n) &= S_1(n) + S_2(\lceil n/2 \rceil) + S_2(\lfloor n/2 \rfloor) && (5.4)\\
&\geq (n_1/2)\lfloor \log_2 n_1 \rfloor + S_2(\lceil n/2 \rceil) + S_2(\lfloor n/2 \rfloor) && (5.5)\\
&\geq (n_1/2)\lfloor \log_2 n_1 \rfloor + && (5.6)\\
&\quad 2\left(2^{\lfloor \log_2 (n_1/2) \rfloor}/4\right)(\lfloor \log_2 (n_1/2) \rfloor + 1)\lfloor \log_2 (n_1/2) \rfloor \\
&= (n_1/4)\log_2 n_1 (\log_2 n_1 + 1) && (5.7)\\
&= \left(2^{\lfloor \log_2 n \rfloor}/4\right)\lfloor \log_2 n \rfloor (\lfloor \log_2 n \rfloor + 1) && (5.8)
\end{aligned}$$

where Line 5.5 follows from the lower bound of Theorem 3, Line 5.6 holds by the inductive assumption because $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$ and $n_1 < n$, and Line 5.7 holds by the arguments in Lines 5.1–5.3.

To prove the upper bound for n not a power of 2, observe that, by replacing $S_1(n)$ in the recurrence for $S_2(n)$ using Theorem 3, $S_2(n) \leq (n/2)\log_2 n + S(\lceil n/2 \rceil) + S(\lfloor n/2 \rfloor)$. We can repeatedly expand the recurrence, where the kth expansion adds 2^k terms of the form $(x/2)\log_2 x$. The ith term of the kth expansion is $(t_{k,i}/2)\log_2 t_{k,i}$, where $t_{k,i}$ is the result of a sequence of k applications to n of the functions $\lfloor y/2 \rfloor$ and $\lceil y/2 \rceil$. Observe that we can express the $t_{k,i}$ as $t_{k,i} = \lfloor (n + i - 1)/2^k \rfloor$, that they only take the values $\lfloor n/2^k \rfloor$ and $\lceil n/2^k \rceil$, and that $\sum_i t_{k,i} = n$.

We first consider the sum of these terms, $f(k) = \sum_{i=1}^{2^k} (t_{k,i}/2)\log_2 t_{k,i}$. We can bound it using the concavity of $\log(x)$ and applying Jensen's inequality:

$$f(k) \leq \frac{n}{2}\log_2 \left(\sum_{i=1}^{2^k} \left(\frac{t_{k,i}}{n} \right) t_{k,i} \right) \tag{5.9}$$

Let $c_k = n \bmod 2^k$, which counts the number $t_{k,i}$ values with the value $\lceil n/2^k \rceil$. Then we can simplify Line 5.9 as follows:

$$f(k) \leq \frac{n}{2}\log_2 \left(\left\lfloor \frac{n}{2^k} \right\rfloor + \left(\frac{c_k}{n} \right) \left\lceil \frac{n}{2^k} \right\rceil \right) \tag{5.10}$$

We observe that

$$\left\lfloor \frac{n}{2^k} \right\rfloor = (n - c_k)/2^k,$$

and, when $c_k > 0$,

$$\left\lceil \frac{n}{2^k} \right\rceil = (n + 2^k - c_k)/2^k.$$

These facts allows us to simplify the bound on $f(k)$:

$$f(k) \leq \frac{n}{2}\log_2\left(\frac{n-c_k}{2^k} + \left(\frac{c_k}{n}\right)\left(\frac{n}{2^k} + \frac{2^k - c_k}{2^k}\right)\right) \tag{5.11}$$

$$= \frac{n}{2}\log_2\left(\frac{n}{2^k} + \left(\frac{c_k}{n}\right)\left(1 - \frac{c_k}{2^k}\right)\right). \tag{5.12}$$

Note that that this bound is still valid when $c_k = 0$ due to the c_k/n factor. Let $g(x) = (x/n)(1 - x/2^k)$. The first derivative is $g'(x) = (1/n)(1 - x/2^{k-1})$, and therefore $g(x)$ is maximized at $x = 2^{k-1}$. Using this fact and Line 5.12, we can conclude that

$$f(k) \leq \frac{n}{2}\log_2\left(\frac{n}{2^k} + \left(\frac{2^{k-1}}{n}\right)\left(1 - \frac{1}{2}\right)\right) \tag{5.13}$$

$$= \frac{n}{2}\log_2\left(\frac{n}{2^k} + \frac{2^{k-2}}{n}\right). \tag{5.14}$$

Using the concavity of $\log_2 x$ and its derivative of $1/((\ln 2)x)$, we have that

$$f(k) \leq \frac{n}{2}\log_2\left(\frac{n}{2^k}\right) + \left(\frac{n}{2}\right)\left(\frac{2^k}{(\ln 2)n}\right)\left(\frac{2^{k-2}}{n}\right) \tag{5.15}$$

$$= \frac{n}{2}\log_2\left(\frac{n}{2^k}\right) + \frac{4^{k-1}}{2(\ln 2)n}. \tag{5.16}$$

We now return to the recurrence for $S_2(n)$, which we can bound as

$$S_2(n) \leq \frac{n}{2}\log_2 n + \sum_{k=1}^{\lceil \log_2 n\rceil - 1} f(k) \tag{5.17}$$

$$\leq \frac{n}{2}\log_2 n + \sum_{k=1}^{\lceil \log_2 n\rceil - 1} \frac{n}{2}\log_2\left(\frac{n}{2^k}\right) + \frac{4^{k-1}}{2(\ln 2)n} \tag{5.18}$$

$$\leq \sum_{k=0}^{\lceil \log_2 n\rceil - 1} \frac{n}{2}\log_2\left(\frac{n}{2^k}\right) + \sum_{k=1}^{\lceil \log_2 n\rceil - 1} \frac{4^{k-1}}{2(\ln 2)n} \tag{5.19}$$

$$\leq \frac{n}{4}(\log_2 n + 1)\log_2 n + \frac{4^{\lceil \log_2 n\rceil - 1} - 1}{6(\ln 2)n} \tag{5.20}$$

$$\leq \frac{n}{4}(\log_2 n + 1)\log_2 n + \frac{n^2 - 1}{6(\ln 2)n} \tag{5.21}$$

$$< \frac{n}{4}(\log_2 n + 1)\log_2 n + \frac{n}{6\ln 2}. \tag{5.22}$$

□

5.5.2 Efficiency of BORPStream

BORPStream can always choose a trivial BRN, which makes its runtime no worse than that of ORShuffle, as shown by Theorem 7.

Theorem 7. *BORPStream runs in time* $O(n \log^2 n)$.

Proof. For a given number of items n, let $p_n = (\mathsf{f}_n, \mathsf{d}_n, \mathsf{s}_n)$ be some set of parameters for the BRN. Fix a block size and failure probability, and consider the runtime T_n of BORPStream given n items when p_n is used in place of the parameters produced by BORP_Optimizer (Line 2, Figure 5.6). Suppose $T_n = O(f(n))$ for some function f. Then the runtime of BORPStream is $O(f(n))$ because the optimizer outputs parameters that yield the smallest predicted runtime using a cost model that is linear in the overall number of operations performed.

A trivial setting of $p_n = (1, 1, 1)$ results in a BRN with only one MSN. BORPStream then simply copies its input packets to a single output bucket, runs ORCompact to remove dummy packets, and then runs ORShuffle to shuffle the n items. By Theorems 2 and 5, the resulting runtime is $O(n \log^2 n)$. □

Of course, BORPStream would not offer advantages if it simply devolved into ORShuffle, and our optimizer does in fact return non-trivial parameter settings (see Table 5.1). As described in Section 5.2.4, the runtime of Phase 1 is $O(n\mathsf{sd})$, and in Phase 2 the runtime of $\mathrm{FlushBuffers}()$ is $O(\mathsf{f}^\mathsf{d}\mathsf{s}^2\mathsf{d}^2)$, while cumulatively in Phase 2 the ORCompact calls take time $O(\mathsf{f}^\mathsf{d} \log V)$ and the ORShuffle calls are $O(\mathsf{f}^\mathsf{d} \log^2 V)$, with $V = O(n/\mathsf{f}^\mathsf{d} + \mathsf{sd})$. If, for example, we consider $2 \leq \mathsf{f} = O(1)$ and $\mathsf{d} = \log_\mathsf{f}(n/c)$ for some constant $c \geq 1$, which is a parameterization used by BORP [ACN+20], then the overall runtime becomes $O(n\mathsf{s}^2 \log^2 n)$, with $\mathrm{FlushBuffers}()$ contributing the dominant term. This analysis indicates that (1) a "narrower" parameterization (i.e., with smaller f^d) is needed to outperform ORShuffle; and (2) small buffer size s is crucial to good performance in BORPStream.

5.5.3 BORPStream Correctness and Optimizer

BORPStream fails in the event that any one of the MSNs incurs a buffer overflow; let the probability of this event be P. We implement a failure probability calculator to numerically bound P for a given number of items n, and use it to produce various choices of $(\mathsf{f}, \mathsf{d}, \mathsf{s})$ that are below the desired failure probability threshold $2^{-\lambda}$. In our experiments

Table 5.1: BORPStream parameters produced by the optimizer for modes V1 (min total time) and V2 (min phase 2 time) across sample problem sizes n and item sizes b.

Mode	n	b	f	d	s	λ
V1	2^{20}	8	2	1	32	81
V1	2^{20}	1024	4	3	47	82
V1	2^{24}	8	4	1	48	82
V2	2^{20}	8	4	4	47	80
V2	2^{20}	1024	4	4	47	80
V2	2^{24}	8	4	5	49	82

we use $\lambda = 80$. Table 5.1 lists the BORPStream parameters produced by our calculator for a few choices of n and b.

The failure probability P is bounded by the union bound over all individual MSN's buffer overflow probability. For each individual MSN, we can bound this buffer overflow failure probability by analyzing the underlying Markov process of how the buffer occupancy changes as items are passed through it. Let an item entering a given MSN be real with probability $\frac{1}{2}$ and each MSN have f output streams or "flavours". We perform this analysis under a simplified model where we consider the MSN receives a batch of f items at a time, and then tries to evict one item of each flavour (if possible, else it evicts a dummy); instead of the MSN evicting an item for each item received.

For an MSN to fail in the above model, it is necessary that its buffer has no items of at least one flavour; if the buffer has at least one item of each flavour, even in the worst case when all incoming f items are real, the buffer size does not change. Given this observation, we consider the state of the MSN buffer at any time as a sorted tuple of the number of items of each flavour, and discard the smallest count in this tuple to reduce the state space and arrive at an upper bound of the failure probability.

For a given s, we generate the transition matrix for one batch of f items by starting with the tuple of all 0's as the buffer state, and then computing the probability distribution of all the possible sequences of f items. We then repeat this process until a complete transition matrix for a maximum buffer size of s is constructed. The transition matrix also has a *fail* state which soaks all the events that lead to a buffer overflow and provides an upper bound for failure. Given n, the calculator selects the depth of the network d from a range of candidate choices. We exponentiate the transition matrix by $\frac{2n}{\mathsf{f}^{\mathsf{d}}}$ (the number of f-item batches each MSN in a layer processes), to arrive at the failure probability of each individual MSN.

5.6 Conclusion

In this chapter we presented two new fully oblivious shuffling algorithms: ORShuffle and BORPStream. Both of these algorithms apply to the standard model. BORPStream, however, has even more benefits in the streaming model since BORPStream can partially shuffle each item as it becomes available, resulting in a faster online time to shuffle data once the last item to shuffle has arrived. Both these algorithms have an $O(\beta n \log^2 n)$ complexity, matching the prior state-of-the-art bitonic shuffle, but presents concrete improvements due to lower underlying constants.

Chapter 6

Oblivious Shuffling and Sorting Algorithms in the Offline/Online Model

In the previous chapter we presented the shuffling algorithms ORShuffle and BORP-Stream, where the former consistently outperforms BitonicShuffle, and the latter reduces the online cost of shuffling items by partially shuffling them as they arrive in a streaming setting. While some of the TEE applications [BEM+17, SO19] mentioned in Section 2.3 can leverage the streaming model, it is not generically applicable. However, all the TEE systems we pointed out can obtain a performance improvement from algorithms designed for the *offline/online* model, in which an offline computation phase is performed before the data is available, followed by an online phase that uses the input data. In general, the offline/online model can be useful to reduce query latency for a system that answers queries on existing data [DFD+21, SO19]. It can also be used to reduce analysis time for systems that otherwise idly wait while collecting (a predetermined amount of) data from users [BEM+17]. For systems with repeated batch processing, it can also enable parallelism, as an offline phase can be performed concurrently with the current online phase.

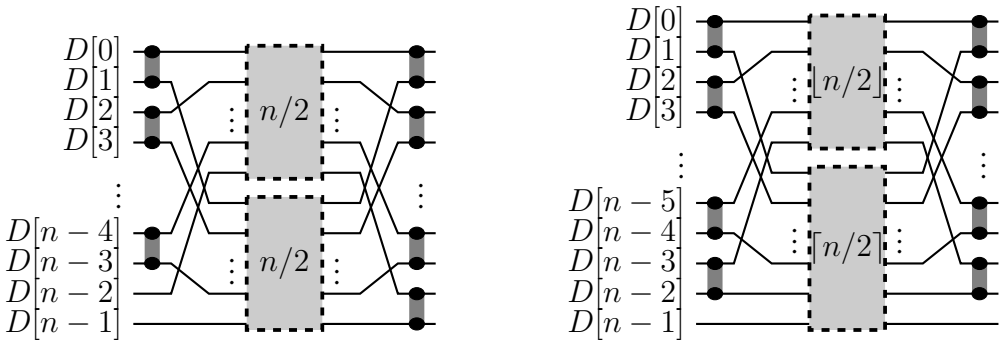
In this chapter we present novel fully oblivious shuffling and sorting algorithms in the offline/online setting. Our algorithms move the bulk of the work for oblivious shuffling and sorting into an offline precomputation that is *independent of the data to be permuted*, resulting in better asymptotic and concrete online costs than prior work. Furthermore, even if one also takes into consideration the offline costs of our algorithms, our *total cost* for shuffling and sorting still outperforms all existing approaches when the items are at

least moderately sized (i.e., larger than about 1400 bytes). Applications permuting items of this size include ML training [OSF+16], where the items to shuffle may be images, and database queries [KKS20], where the items to sort may be entire rows of database tables. Therefore, even without the offline/online split, our algorithms are useful for several applications, and the total costs are modest even for compute-limited servers.

We achieve these performance improvements using Waksman networks [Wak68]. A Waksman network can apply any permutation to a set of items by sending them through a fixed-topology network of switches. Each switch has two inputs, two outputs, and a control bit; the outputs are the inputs in swapped order if the control bit is one and in the original order otherwise. For a set of n input items, there are approximately $n \log_2(n) - n + 1$ switches (and exactly this number when n is a power of two). Obliviously setting the control bits to implement a given permutation poses a challenge, but this process is agnostic to the size of items to permute.

The key observation we make is that to uniformly randomly shuffle items, one can generate the control bits for the Waksman networks *independently of the items to shuffle*. We hence reduce the online cost of shuffling to the $O(\beta n \log n)$ cost of permuting n items each of size β with a Waksman network. Further, the parallel step complexity is only $O(\log n)$. In contrast, existing oblivious shuffling algorithms like BitonicShuffle, ORShuffle, and BORPStream incur a $O(\beta n \log^2 n)$ online complexity, with a parallel step complexity of $O(\log^2 n)$. Finally, while we focus on shuffling and sorting, by definition Waksman networks can be used to apply any permutation, and in scenarios where several sets of data need to be permuted in the same order the cost of setting control bits can be amortized over these sets.

For sorting data, the control bits for a Waksman network would depend on the items to be sorted; i.e., one needs to generate the permutation that results in a sorted output, and then set the control bits accordingly. However, by using the "Scramble-then-Compute" paradigm [HKI+12, DDCO17], one can reduce the online cost of sorting data as well. A Waksman network can be used to perform an oblivious shuffle, and then this shuffled data can be sorted using a fast non-oblivious sort such as quicksort. This algorithm has an online cost of $O(\beta n \log n)$ and outperforms bitonic sorting networks, which have a $O(\beta n \log^2 n)$ online cost, and it similarly outperforms ORShuffle or BORPStream (both having a $O(\beta n \log^2 n)$ complexity) followed by a non-oblivious sort.



(a) Even n (b) Odd n

Figure 6.1: Existing recursive construction for Waksman network of n inputs [BD02]. Crossbars denote switches. Switches are applied left to right on input array D. Dashed rectangles are Waksman subnetworks with the given numbers of inputs. Note that inputs to subnetworks are not contiguous in D.

6.1 Background

Waksman networks. A Waksman network [Wak68] is an arrangement of binary *switches*. Each switch has two inputs and two outputs, and its behavior is governed by a control bit. If the control bit is set (i.e., equal to one), then the switch swaps its inputs. If the bit is unset (i.e., equal to zero), then the switch outputs its inputs in their original order. A Waksman network can apply any permutation to its inputs by setting its control bits appropriately. Setting a Waksman network to a uniformly random permutation is not as simple as setting its control bits independently and uniformly at random because the number of such settings is a power of two and thus is not evenly divided by the number of permutations when the number of inputs is more than two.

The Waksman network construction [BD02] is illustrated recursively in Figure 6.1. The construction is slightly different for even n (Figure 6.1a) and odd n (Figure 6.1b). In both cases, a layer of *input switches* is first applied, the results are used as inputs to *top* and *bottom* Waksman *subnetworks*, and then a layer of *output switches* are applied to the outputs of those subnetworks. The switches in a given layer take disjoint inputs, and so they can be applied in parallel. When $n = 2$, the network is a single switch, and when $n = 1$ there are no switches. Waksman networks are designed to minimize the number of switches while still being able to apply any permutation. The number of switches is exactly $n \log_2(n) - n + 1$ for n a power of two and at most $n \log_2(n) - 0.91n + 1$ for any

n [BD02]. Observe that the topology of the Waksman network is fixed; that is, which switches are applied to which values in what order depends only on n. This property makes Waksman networks useful for oblivious computation.

Waksman [Wak68] describes a $O(n \log n)$ algorithm to set the control bits to implement a given permutation. In this "looping" algorithm, the input and output switches are alternately set, as setting one determines the required setting for a following one, and then the algorithm is recursively called on the subnetworks. Nassimi and Sahni [NS82] give a more parallelizable algorithm to set control bits. It has runtime $O(n \log^2(n))$ but parallel step complexity of $O(\log^2(n))$, improving on the $O(n)$ parallel step complexity of the Waksman algorithm. Bernstein [Ber20] describes how to perform the Nassimi-Sahni algorithm obliviously. Using an oblivious sort with runtime $O(n \log^2(n))$, which is the fastest in practice [Ebb15], the Bernstein algorithm has runtime $O(n \log^4(n))$.

Lee [Lee85, Lee87] presents a different approach to setting the Waksman control bits. In the Lee algorithm, the input switches are set first, then the subnetwork switches, and finally the output switches. The algorithm is not oblivious and has the same $O(n \log n)$ runtime as the looping algorithm. However, an advantage of the Lee algorithm in the context of oblivious computation is that setting the output switches is trivially and efficiently oblivious. We therefore use the Lee algorithm as our starting point in designing a fully oblivious control bit setting algorithm.

6.2 Oblivious Algorithms for Waksman Networks

6.2.1 Preliminaries

When describing algorithms, we use $A[i]$ to indicate the element of array A at index i, with $A[0]$ as the first element. We use $A[i..j]$ to indicate the subarray $(A[i], A[i+1], \ldots A[j])$, and we use $|A|$ to denote the number of items in A. We denote bitwise AND with $\&$, bitwise OR with $|$, and the left (right) shift of x by y bits with $x \ll y$ ($x \gg y$). We assume a global security parameter λ that is available to all algorithms, and we do not require that any algorithm is oblivious to it.

We also make use of several fully oblivious primitives. We assume that random bits can be generated fully obliviously. We assume that arithmetic and bitwise operators are oblivious to their inputs. We denote the oblivious computation of the comparison c as a boolean value with $[\![c]\!]$ (e.g., $[\![x < y]\!]$ has value 1 if $x < y$ and 0 otherwise). We use an oblivious select function, OSelect(x, y, b), which returns y if $b = 1$ and returns x otherwise. OSelect is assumed to be oblivious to its input values. We also use the

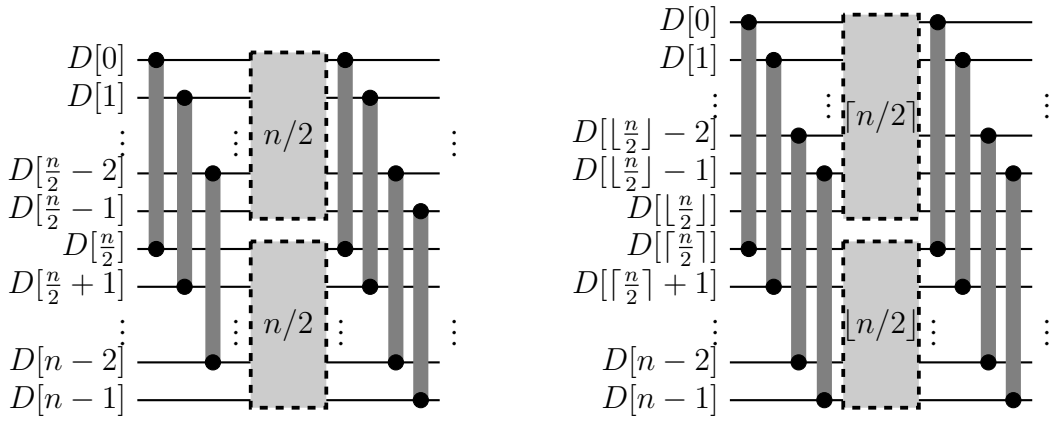
oblivious swap function, OSwap(A[i],A[j], b), which swaps the values of $A[i]$ and $A[j]$ if $b = 1$ and does nothing otherwise. OSwap(A[i],A[j], b) is assumed to be oblivious to A and b and to the contents, but not the addresses, of $A[i]$ and $A[j]$. For items of size β, OSwap is assumed to have runtime $O(\beta)$. These primitives can be implemented efficiently and obliviously using boolean arithmetic or conditional instructions such as `CMOVZ`.

Some additional functionalities we use require other approaches to provide the necessary obliviousness. We use ORand(x) to select a uniformly random integer in $[0, x-1]$, which we assume is fully oblivious with respect to its input. One efficient oblivious implementation with statistical distance $x2^{-\sigma}$ from uniform is to select σ random bits ρ and compute $x\rho \gg \sigma$. We also require a pseudorandom permutation, PRP(x, k), that applies a PRP to x using key k and is fully oblivious to its inputs. A securely implemented PRP (e.g., AES) must have a constant-time implementation and would thereby provide the required obliviousness. We additionally use a hash map data structure with $O(1)$ amortized insertion and lookup. Given hash map H, HashMapInsert(H, k, v) inserts value v with key k, and HashMapGet(H, k) returns the value inserted for key k. We require that, for any sequence of insertions and lookups, the resulting memory and instruction traces can be simulated without knowing the values v (but knowing the keys k). The standard hash map algorithms are already oblivious in this way.

Although a goal of our algorithms is to provide oblivious shuffling and sorting, we do rely on existing fully oblivious shuffling and sorting algorithms. Our algorithms will provide efficiency improvements over applying those algorithms directly to the data. OShuffle(A) is assumed to put the items in array A in a uniformly random order. OSort(A, B) is assumed to sort array A in place in nondecreasing order and also put B in the same final order as A (e.g., OSort could be used to sort data items in B using keys in A). Both OShuffle and OSort are assumed to be fully oblivious to all of their inputs. Hence we recommend using ORShuffle and bitonic sorting networks to instantiate OShuffle and OSort respectively.

6.2.2 Our Waksman Topology

Our algorithms make use of a modified Waksman topology, which is shown in Figure 6.2 for a network with n inputs. As with the standard topology (Figure 6.1), we use a recursive construction with slight differences for even n (Figure 6.2a) and odd n (Figure 6.2b). The base cases remain the same: the network is a single switch for $n = 2$, and there are no switches for $n = 1$. In our topology, however, the input switches take inputs separated by $\lceil n/2 \rceil$ rather than by 1, and output switches yield outputs separated by $\lceil n/2 \rceil$



(a) Even n (b) Odd n

Figure 6.2: Our recursive construction for Waksman networks of n inputs. Crossbars denote switches. Switches are applied left to right on input array D. Dashed rectangles are Waksman subnetworks for the given numbers of inputs. Note that inputs to the subnetworks are contiguous in D.

rather than by 1. The advantage of this topology in our setting is that, after applying the input switches, the inputs to each subnetwork are contiguous in memory rather than fragmented, whereas in the standard topology there is fragmentation that increases with each subnetwork. This design improves the memory locality and thus performance for both setting control bits and applying them to a data array. This topology uses the same number of switches as the standard one. See Section 6.2.4 for a complete example in the $n = 9$ case.

6.2.3 Setting Control Bits

Our fully oblivious algorithm ControlBits(P, d) sets the Waksman control bits to implement a permutation, and it is shown in Figure 6.3. For $n = |P|$, P must be an arrangement of $\{0, \ldots, n-1\}$, with $P[i] = j$ meaning that i is mapped to j. The initial call should be with a recursion depth of $d = 0$. ControlBits is fully oblivious to P but not n or d. ControlBits is similar to the Lee algorithm [Lee87] for setting control bits. The main differences are (1) the switches follow our modified topology, (2) the algorithm explicitly handles array sizes that are not a power of two, and (3) crucially, setting input

Figure 6.3: ControlBits(P, d): Compute Waksman control bits for permutation P at recursion level d. i maps to a value encoded in $P[i]$. Initially call with $d = 0$. Modifies P. Oblivious to values in P but not to $|P|$ or d.

$n \leftarrow |P|$
$k \leftarrow \lceil n/2 \rceil$
if $n < 2$ **then**
    return $\emptyset$
▷ *Compute control bits for input-layer switches*
$C_{\text{in}} \leftarrow$ InBits(P, d)
▷ *Apply input-layer switches to P*
if $n > 2$ **then**
    for $i \leftarrow 0, \ldots, k-2$ **do**
        OSwap$(P[i], P[k+i], C_{\text{in}}[i])$
▷ *Reduce P values modulo k, storing bits to undo later*
for $i \leftarrow 0, \ldots, n-1$ **do**
    $b \leftarrow [\![(P[i] \gg d) \geq k]\!]$
    $P[i] \leftarrow ((P[i] - (\text{OSelect}(0, k, b) \ll d)) \ll 1) \mid b$
▷ *Compute control bits for top and bottom subnetworks*
if $n > 2$ **then**
    $C_{\text{top}} \leftarrow$ ControlBits$(P[0..k-1], d+1)$
    $C_{\text{bot}} \leftarrow$ ControlBits$(P[k..n-1], d+1)$
▷ *Compute control bits for output-layer switches*
Create array C_{out} of length $n - k$.
for $i \leftarrow 0, \ldots, n-k-1$ **do**
    $C_{\text{out}}[i] \leftarrow P[i]$ & 1 ▷ Control bit is the last bit of $P[i]$
▷ *Apply output-layer switches to P*
for $i \leftarrow 0, \ldots, n-k-1$ **do**
    OSwap$(P[i], P[k+i], C_{\text{out}}[i])$
▷ *Undo reduction of P values modulo k*
for $i \leftarrow 0, \ldots, n-1$ **do**
    $b \leftarrow P[i]$ & 1
    $P[i] \leftarrow (P[i] \gg 1) + (\text{OSelect}(0, k, b) \ll d)$
return $(C_{\text{in}}, C_{\text{top}}, C_{\text{bot}}, C_{\text{out}})$

---

switches is modified to make the process oblivious (setting output switches is already oblivious).

The Lee algorithm is described for permutation sizes that are powers of two. First, input switches are set and are applied to their input permutation values, then the subnetwork switches are set recursively, and finally the output switches are set and are applied to their permutation inputs. In the ith subnetwork level (starting at $i = 0$ and incrementing with each subnetwork), the input switches are set so that applying them to the input subarray of P yields as input to each subnetwork a *Complete Residue System modulo* $n/2^{i+1}$, which is a set containing one representative from each residue class modulo $n/2^{i+1}$. That is, if $\hat{P}$ is the subarray of P used as input to a subnetwork, the input switches are set to make $\{x \bmod n/2^{i+1}\}_{x \in \hat{P}} = \{0, \ldots, n/2^{i+1} - 1\}$. After recursively setting the subnetwork switches (which also applies them to elements of P), each output switch is set to the $(\log_2 n - i - 1)$st bit of its input from the top subnetwork, where the zeroth bit is the least-significant bit. As the network topology is fixed, setting the output switches is trivially oblivious. The output switches are then applied to their inputs in P.

ControlBits(P, d) follows a similar process. We first observe that it uses our modified topology. The input switches are applied to items $k = \lceil n/2 \rceil$ apart (Line 10) rather than one apart. Consequently, the subnetwork inputs are contiguous (Lines 17 and 18), which improves memory locality during the recursive calls.

The algorithm is also defined for all n, not just powers of two. To achieve this generalization, we modify the process of modular reduction while setting input switches and of extracting the control bits while setting the output switches. In Lines 12–14, each permutation value is reduced modulo k (accomplished via a shift and a subtraction), and a bit indicating if the value changed is stored at the end (accomplished via a shift and an OR). The result is that at the dth recursive call, each element of P stores a bit array in the d least-significant bits and its reduced values in the remaining (most-significant) bits. After setting the output switches as the least significant bits (Lines 20–22), this reduction process is undone by reversing it (Lines 27–29).

The biggest change to the Lee algorithm comes in setting the input switches (Line 6). In both Lee's algorithm and our oblivious algorithm, each input has a *partner* with which it shares an input switch (in our topology, i and $i \pm k$ are partners). Each permutation value $v \leftarrow P[i] \gg d$ also has an *associate* with which it shares the same residue class modulo k (in our topology, v and $v \pm k$ are associates). A "back-and-forth" process is repeatedly performed between the permutation inputs and outputs. It starts with the input index f that is sent directly to the top subnetwork without entering an input switch (index $\lceil n/2 \rceil - 1$ in our topology). In each iteration, (1) g is set to the fth input value;

(2) r is set to the associate of g; (3) s is set to the index of the input that contains r; (4) the switch that s is an input to is set to put r on the output leading to the bottom subnetwork; (5) f is updated to the partner of s; and (6) if f then has the same value it had in the first iteration (yielding a *cycle* during this process), f is set to an arbitrary input index that has not yet had its input switch set, starting a new cycle. Setting the input switch as in Step 4 ensures that the two values with the same residue class are sent to different subnetworks, ultimately producing the desired Complete Residue Systems as inputs to each subnetwork.

We set the input switches obliviously with InBits (Figure 6.4). The challenging steps to make oblivious are the *forward* map under the given permutation P (Step 1), the *reverse* map under P (Step 3), and starting a new cycle (Step 6). For the forward map, we use CreateForwardLookup (Figure 6.5) to create a lookup table F that contains the n permutation mappings of P. Each mapping is labeled by a PRP applied to its input, and F is (obliviously) sorted by those labels. This structure allows each mapping to be looked up via a binary search on the labels, which is performed by ForwardOrRand (Figure 6.6). For the reverse map, we simply store the n inverse permutation mappings of P in a hash map R with a PRP applied to the input value as the lookup key, produced by CreateReverseLookup (Figure 6.7). Because each permutation mapping is looked up at most once and in either the forward or the reverse direction (but not both), the lookup pattern appears random, which is how it achieves obliviousness. We must use different keys for the PRPs used in F and R because, while each mapping will be looked up in at most one direction, a given integer might be looked up twice (i.e., an f in some iteration might equal an s in some iteration). Also note that we store the input switch settings in the order they are determined along with their switch numbers (Lines 30–31), and then we obliviously sort them by switch number to put them in the required order (Line 33).

We could have used a hash map for the forward direction except for a complication that arises when a back-and-forth cycle ends. After a cycle ends, the next lookup (if any) is in the forward direction, and we need to look up a mapping that has not yet been queried in either direction. To accomplish this, CreateUnselectedCounts (Figure 6.8) creates a data structure U to track which mappings have been queried in either direction and to enable the uniformly random selection of an unqueried one. U is an array containing in its middle element the number of unselected mappings in the first half of the forward-lookup table F, and then its two halves contains the same values recursively for the corresponding halves of F. Therefore, ForwardOrRand can also look up a uniformly random unselected mapping while still performing a binary search, making its execution indistinguishable from querying a specific mapping. Maintaining U requires decrementing its counts with each lookup in either direction, which is accomplished by

Figure 6.4: InBits(P, d): Compute Waksman control bits for input layer of switches for permutation P at recursion level d.

$n \leftarrow |P|$
$k \leftarrow \lceil n/2 \rceil$
Create array C of length $k-1$. ▷ Control bits
if $n \leq 2$ **then**
    return C
Create array S of length $k-1$. ▷ Switch numbers of control bits
Generate λ random bits κ_1. ▷ PRP key for forward lookup
Generate λ random bits κ_2. ▷ PRP key for reverse lookup
$F \leftarrow$ CreateForwardLookup(P, d, κ_1)
$R \leftarrow$ CreateReverseLookup(F, κ_2)
$U \leftarrow$ CreateUnselectedCounts$(n, \emptyset)$
▷ *Perform initial back-and-forth with fixed input* $k-1$
$f, g, \ell \leftarrow$ ForwardOrRand$(F, U, k-1, 0, \kappa_1)$ ▷ Map $k-1$
DecUnselectedCounts(U, ℓ) ▷ Mark $k-1 \to g$ as selected
$c \leftarrow f$ ▷ Cycle start is f (which contains $k-1$)
$r \leftarrow g + \mathrm{OSelect}(-k, k, [\![g < k]\!])$ ▷ r is the associate of g
$s, \ell \leftarrow$ HashMapGet$(R, \mathrm{PRP}(r, \kappa_2))$ ▷ Reverse map of r
DecUnselectedCounts(U, ℓ) ▷ Mark $s \to r$ map as selected
$f \leftarrow s + \mathrm{OSelect}(-k, k, [\![s < k]\!])$ ▷ f is the partner of s
▷ *Repeat the back-and-forth process to compute the control bits*
for $i \leftarrow 0, \ldots, k-2$ **do**
    $b \leftarrow [\![f = c]\!]$ ▷ Indicate if current cycle has ended
    ▷ *If* b*, choose random unselected* $f \to g$ *map, else map* f
    $f, g, \ell \leftarrow$ ForwardOrRand(F, U, f, b, κ_1)
    DecUnselectedCounts(U, ℓ) ▷ Mark $f \to g$ as selected
    $c \leftarrow$ OSelect(c, f, b) ▷ If new cycle begun, update cycle start
    $r \leftarrow g + \mathrm{OSelect}(-k, k, [\![g < k]\!])$ ▷ r is the associate of g
    $s, \ell \leftarrow$ HashMapGet$(R, \mathrm{PRP}(r, \kappa_2))$ ▷ Reverse map of r
    DecUnselectedCounts(U, ℓ) ▷ Mark $s \to r$ as selected
    $C[i] \leftarrow [\![f \geq k]\!]$ ▷ Control bit to put f in top subnetwork
    $S[i] \leftarrow f - \mathrm{OSelect}(0, k, [\![f \geq k]\!])$ ▷ Switch number of f
    $f \leftarrow s + \mathrm{OSelect}(-k, k, [\![s < k]\!])$ ▷ f is the partner of s
OSort(S, C) ▷ Order control bits by increasing switch number
return C

Figure 6.5: CreateForwardLookup(P, d, κ): Create data structure for forward map under permutation in P at recursion depth d using key κ.

```
1: n ← |P|
2: k ← ⌈n/2⌉
3: m ← 2k
4: Create array L of length m.                    ▷ Random labels for mappings
5: for i ← 0, ..., m − 1 do
6:     L[i] ← PRP(i, κ)
7: Create array Q of length m.                    ▷ Permutation mappings
8: for i ← 0, ..., n − 1 do
9:     Q[i] ← (i, P[i] ≫ d)
10: if n ≠ m then                  ▷ Add dummy mapping to make total size even
11:     Q[n] ← (n, n)
12: OSort(L, Q)          ▷ Randomize order of mappings in Q by sorting on L
13: return (L, Q)
```

DecUnselectedCounts (Figure 6.9). Note that DecUnselectedCounts reveals which counts are decremented but maintains obliviousness because doing so reveals no more than which location was queried in F or R, which is both pseudorandom and already apparent in prior memory accesses. We next present an example of running ControlBits.

6.2.4 Worked Example of ControlBits

Figure 6.10 illustrates an example of running our ControlBits algorithm to set the switches of a Waksman network using our topology. The examples is on a network with $n = 9$ inputs. In the figure, the switches are shown as crossbars connecting two inputs, and they are applied to values from left to right. Subnetworks are indicated by a dashed rectangle and labeled by a letter in the upper left. For example, the whole (i.e., $n = 9$) network contains a top subnetwork A with 5 inputs and a bottom subnetwork E with 4 inputs, while subnetwork A itself contains a top subnetwork B with 3 inputs and a bottom subnetwork D with 2 inputs.

The control bits of the network in Figure 6.10 are set to implement the permutation shown: $P = (6, 2, 3, 7, 5, 1, 8, 0, 4)$. A dashed red crossbar indicates a switch that at the end of the algorithm is unset (i.e., a switch with a control bit of zero), and a solid green

Figure 6.6: ForwardOrRand(F, U, f, b, κ): If $b = 0$, look up forward map from f under permutation encoded in F under key κ. Else (i.e., if $b = 1$), look up a uniformly random mapping in F from among those that U indicates are unselected. Returns mapping input f, output g, and index ℓ.

```
1:  L, Q ← F
2:  n ← |L|
3:  h ← PRP(f, κ)                                  ▷ Label to search for in L
4:  i ← 0                                          ▷ Start index of range in L and U to search
5:  j ← n − 1                                      ▷ End index of range in L and U to search
6:  u ← U[n − 1]                                   ▷ Number of unselected items in search range
7:  ρ ← ORand(u)                                   ▷ Random unselected mapping
8:  while true do                                  ▷ Binary search for label h or unselected-item ρ
9:      ℓ ← ⌊(i + j)/2⌋
10:     if i = j then
11:         break
12:     z ← OSelect([[L[ℓ] < h]], [[U[ℓ] ≤ ρ]], b)                    ▷ Search direction
13:     if z = 0 then                              ▷ Continue search in first half of range
14:         j ← ℓ
15:         u ← U[ℓ]
16:     else                                       ▷ Continue search in last half of range
17:         i ← ℓ + 1
18:         u ← u − U[ℓ]
19:         ρ ← ρ − U[ℓ]
20: f, g ← Q[ℓ]
21: return (f, g, ℓ)
```

Figure 6.7: CreateReverseLookup(F, κ): Create hash map containing reverse map of permutation in forward-lookup structure F using key κ.

```
1:  L, Q ← F
2:  n ← |Q|
3:  Create empty hash map R sized to contain n items.
4:  for i ← 0, . . . , n − 1 do
5:      x, y ← Q[i]                          ▷ Q[i] contains forward map x → y
6:      z ← PRP(y, κ)                          ▷ Create lookup key z as PRP of y
7:      HashMapInsert(R, z, (x, i))            ▷ Insert value (x, i) under key z
8:  return R
```

Figure 6.8: CreateUnselectedCounts(n, U): Create data structure recursively storing the total unselected count and the unselected count in the first half. Call initially with $U = \emptyset$.

```
1: if U = ∅ then
2:     Create a new array U of length n.
3:     U[n − 1] ← n                          ▷ Last item contains total unselected count
4: if n < 2 then
5:     return U
6: k ← ⌈n/2⌉
7: U[k − 1] ← k                              ▷ Midpoint contains unselected count in first half
8: ▷ Recursively add counts to each half
9: CreateUnselectedCounts(k, U[0..(k − 1)])
10: CreateUnselectedCounts(n − k, U[k..(n − 1)])
11: return U
```

Figure 6.9: DecUnselectedCounts(U, ℓ): Decrement counts in U to indicate that position ℓ has been selected. Modifies U.

```
1: n ← |U|
2: k ← ⌈n/2⌉
3: if ℓ < k then                             ▷ Selected item is in first half
4:     U[n − 1] ← U[n − 1] − 1
5:     if n > 1 then
6:         DecUnselectedCounts(U[0..(k − 1)], ℓ)
7: else                                      ▷ Selected item is in last half
8:     DecUnselectedCounts(U[k..(n − 1)], ℓ − k)
```

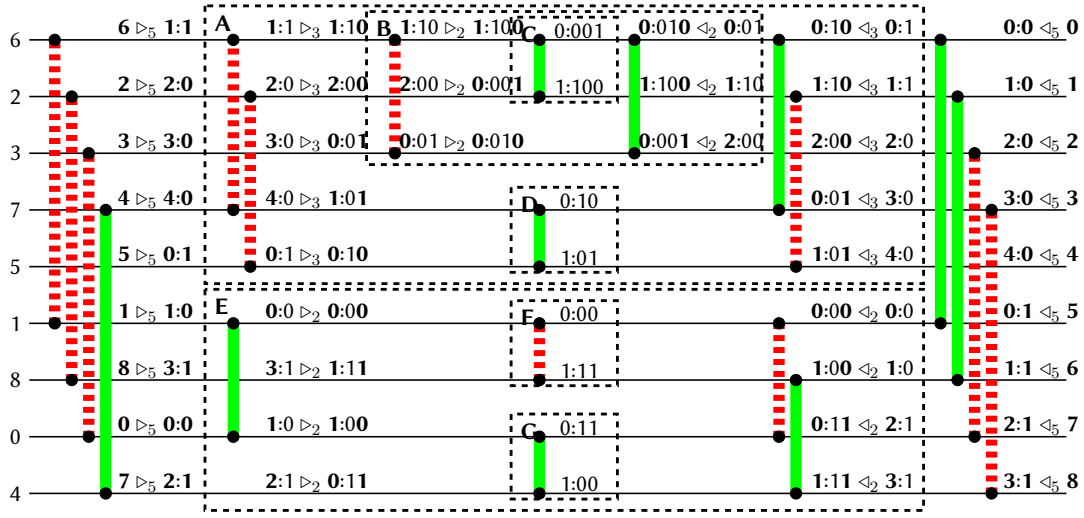


Figure 6.10: A Waksman network in our topology for $n = 9$ with control bits implementing permutation $P = (6, 2, 3, 7, 5, 1, 8, 0, 4)$. A dashed red crossbar indicates switch with a control bit of zero, and a solid green crossbar indicates a switch with a control bit of one. The numeric values are the values of P as they are propagated through the network and modified during ControlBits.

crossbar indicates a switch that ends up set (i.e., a switch with a control bit of one). The numeric values that appear throughout are the values of P as they are propagated through the network during ControlBits. The notation $\mathbf{x} : y \triangleright_k \mathbf{w} : y\mathbf{z}$ indicates that x gets reduced modulo k to w, y is the sequence of bits recording if prior modulo reductions changed the value or not, and z is 1 if x changed under this modulo reduction (i.e., $z = 1$ if $w = x - k$, and $z = 0$ if $w = x$). The bold values emphasize which components of a given entry in P are potentially changed at this step. Similarly, the notation $\mathbf{w} : y\mathbf{z} \triangleleft_k \mathbf{x} : y$ indicates the reversal of the modulo reduction of x by k that yielded w, where y records the action of modulo reduction before the one being reversed and z indicated if x changed under this reduction.

We follow Figure 6.10 from left to right to observe the actions of ControlBits. It initially uses InBits to compute control bits for the first set of input switches. The modulus it uses is $k = \lceil 9/2 \rceil = 5$. Starting with input $f = 4$, which is the input that leads to no switch at this level of recursion, it is mapped to $g = P[4] = 5$, the associate is then $s = 5 - k = 0$, its inverse map is $r = 7$ because $P[7] = 0$, and then f is updated to its partner $f = 7 - k = 2$. This back-and-forth process continues and sets control bits as follows:

1. Set $g = 3$, $r = 8$, $s = 6$. Set the control bit of the switch containing input $f = 2$ (and input 7) to $[\![f \geq k]\!] = [\![2 \geq 5]\!] = 0$, ensuring that the associate values 5 and 0 (i.e., the prior values of g and s) will be inputs to different subnetworks. Set $f = s - k = 1$

2. Set $g = 2$, $r = 7$, and $s = 3$. Then set the control bit of the switch containing input $f = 1$ to $[\![1 \geq 5]\!] = 0$. Set $f = 8$.

3. Set $g = 4$, $r = 9$, and $s = 9$. Then set the control bit of the switch containing input $f = 8$ to $[\![8 \geq 5]\!] = 1$. Set $f = 4$.

4. The cycle has ended because $f = 4$ was the initial value. The remaining mappings are $0 \to 6$ and $5 \to 1$, and ForwardOrRand randomly selects $0 \to 6$ and sets $f = 0$. Set $g = 6$, $r = 1$, and $s = 5$. Then set the control bit of the switch containing input $f = 0$ to $[\![0 \geq 5]\!] = 0$.

The control bits for the first set of input switches are thus set. The switches are applied to the values in P, and those values are reduced modulo $k = 5$ with bits appended recording any reduction are stored. This process is shown in Figure 6.10 by the values after the initial input switches. Observe that the reduced values provided to the top subnetwork (i.e., network A) is a Complete Residue System (CRS) modulo 5, and the reduced values for the bottom subnetwork (i.e., network E) are a CRS modulo 4; that is, those sets of values are $\{0, 1, 2, 3, 4\}$ and $\{0, 1, 2, 3\}$, respectively.

Control bits are then set and applied recursively in the top and bottom subnetworks—networks A and E, respectively. Network A takes 5 inputs, and network E takes 4 inputs. They each contain subnetworks themselves, with the recursion ending when a network has 1 or 2 inputs. A one-input network is null (i.e., has no switches). A two-input network (e.g., network C) consists of a single output switch, which has its control bit effectively set to the value of its first input (by first appending that value to its attached bit array).

After the subnetworks are set and applied, control bits are computed for the last set of output switches. For each pair of inputs to a switch, the control bit is taken to be the last element of the bit array for the first input. Therefore, we see that the output switch containing input 0 is set to 1, the switch for input 1 is set to 1, the switch for input 2 is set to 0, and the switch for input 3 is set to 0. These switches are applied to the input values, and then the reduction modulo 5 is undone, yielding the original set of values $\{0, 1, \ldots, 8\}$ in sorted order. Note that, because each subnetwork also outputs in sorted order the set of values in its input CRS, the two inputs to a given output switch have the same values modulo 5, and therefore setting the control bit based on the reduction bit

Figure 6.11: ApplyPerm(C, D): Apply permutation encoded in Waksman controls bits C to array D. Modifies D. Oblivious to values in C and D but not to $|C|$ or $|D|$.

```
1:  n ← |D|
2:  if n < 2 then
3:      return
4:  C_in, C_top, C_bot, C_out ← C
5:  k ← ⌈n/2⌉
6:  if n > 2 then
7:      for i ← 0, ..., k − 2 do                    ▷ Apply input-layer switches
8:          OSwap(D[i], D[k + i], C_in[i])
9:      ▷ Recursively apply switches in top and bottom subnetworks
10:     ApplyPerm(C_top, D[0..k − 1])
11:     ApplyPerm(C_bot, D[k..n − 1])
12: for i ← 0, ..., n − k − 1 do                    ▷ Apply output-layer switches
13:     OSwap(D[i], D[k + i], C_out[i])
```

of the first input puts those two values in the correct order (and the correct positions) once the reduction modulo 5 is undone.

6.2.5 Applying Permutations

Once the control bits are set, it is straightforward to obliviously apply the permutation that they encode. Figure 6.11 shows the algorithm ApplyPerm, which does so under our Waksman topology. It first applies the input switches, then recursively applies the two subnetworks, and finally applies the output switches. The inverse permutation can also easily be applied, as ApplyInvPerm shows (Figure 6.12), by applying the output switches first and the input switches last. Because the Waksman networks have a fixed topology for a given n, these algorithms are fully oblivious.

6.3 Oblivious Shuffling and Sorting

While our oblivious Waksman algorithms may be useful in many settings, we focus on how they can be used for shuffling and sorting. Waksman networks might achieve faster shuffling and sorting because they perform fewer pairwise swaps of data items than

Figure 6.12: ApplyInvPerm(C, D): Apply inverse of permutation encoded in Waksman controls bits C to array D. Modifies D. Oblivious to values in C and D but not to $|C|$ or $|D|$

```
1:  n ← |D|
2:  if n < 2 then
3:      return
4:  C_in, C_top, C_bot, C_out ← C
5:  k ← ⌈n/2⌉
6:  for i ← 0, ..., n − k − 1 do                    ▷ Apply output-layer switches
7:      OSwap(D[i], D[k + i], C_out[i])
8:  if n > 2 then
9:      ▷ Recursively apply switches in top and bottom subnetworks
10:     ApplyInvPerm(C_top, D[0..k − 1])
11:     ApplyInvPerm(C_bot, D[k..n − 1])
12:     for i ← 0, ..., k − 2 do                    ▷ Apply input-layer switches
13:         OSwap(D[i], D[k + i], C_in[i])
```

other oblivious algorithms. They do, however, carry the additional cost of setting the control bits. Doing so operates only on the *permutation* and, if the permutation does not depend on the data, not the *data items*. Therefore, Waksman networks can admit an *offline* computation phase before the data items are available, in which the control bits are set at a cost independent of the data-item size, and an *online* phase once the data items are available, in which the permutation is applied with relatively few swaps of the data items themselves.

6.3.1 Shuffling

The goal of shuffling is to put an array of data items in a uniformly random order. Shuffling obliviously can hide the resulting order as well as the data contents. Let D be an array of data items, with $|D| = n$. The fully oblivious WaksShuffle(D) algorithm (Figure 6.13) chooses a random permutation by applying an existing oblivious shuffle, OShuffle, to the array $(0, \ldots, n-1)$. This permutation is used to set the Waksman control bits. The resulting network is applied to D, accomplishing the shuffle. Despite using an existing oblivious shuffle as a subroutine, this algorithm can improve efficiency if the data items are large because that shuffle is only applied to integers less than n. Moreover, setting the control bits occurs during an offline phase because the permuta-

Figure 6.13: WaksShuffle(D): Put items in D in a uniformly random order. Divided into an offline phase (where only $|D|$ is needed) and an online phase.

```
▷ Begin offline phase
n ← |D|
Create array P of length n.
for i ← 0, ..., n − 1 do
    P[i] ← i
OShuffle(P)                          ▷ Make permutation uniformly random
C ← ControlBits(P, 0)
▷ Begin online phase
ApplyPerm(C, D)
```

Figure 6.14: WaksSort(K, D): Sort D in increasing order according to the corresponding keys in K. Also sort K.

```
n ← |D|
Create array P of length n.
for i ← 0, ..., n − 1 do
    P[i] ← i
OSort(K, P)          ▷ Make P the inverse of the permutation that sorts K
C ← ControlBits(P, 0)
ApplyInvPerm(C, D)                   ▷ Put D in same order as now-sorted K
```

tion does not depend on the data, though the number of items n must be known.

6.3.2 Sorting

The fully oblivious WaksSort(K, D) algorithm (Figure 6.14) takes an array of keys K and an array of data items D. Its goal is to sort the items in D by increasing value of their corresponding keys, where item $D[i]$ corresponds to key $K[i]$. It uses an existing oblivious sorting algorithm, OSort, to put the keys in increasing order and simultaneously rearrange the array $P = (0, \ldots, n-1)$ in the same way (i.e., if the ith element of K is moved to position j, the same happens to P). The result is that P contains the inverse permutation needed to sort the items of D by their (original) keys. P is used to set the Waksman control bits, and then the inverse permutation is applied to D with ApplyInvPerm. In a setting where the data items are large compared to the keys, WaksSort

can yield a faster sort because the existing oblivious sort is only applied to the keys.

Faster *online* oblivious sorting can be achieved by executing WaksShuffle(D) and then following it with the non-oblivious quicksort. In WaksShuffle+QS, the control bits are set to encode a uniformly random permutation during the offline phase. Then, for the online phase, the Waksman network is applied to D, and quicksort is run on the result. Performing a non-oblivious comparison sort after an oblivious shuffle (assuming keys are distinct) does not violate obliviousness [DDCO17]. Also, even with the added cost of non-oblivious sorting, the online phase may be faster than existing oblivious sorting algorithms because both ApplyPerm and quicksort have $O(\beta n \log n)$ runtime (where β is the size of each data item) with small constants.

6.4 Analysis

We show that WaksShuffle and WaksSort are correct and fully oblivious. By way of doing that, we demonstrate these properties for ControlBits, ApplyPerm, and ApplyInvPerm. These algorithms can be used for applications of Waksman networks outside of shuffling and sorting. We also analyze the efficiency of all of our algorithms. The proofs for all theorems are provided later in Section 6.8.

6.4.1 Correctness

We first show the correctness of our Waksman-network algorithms. Although these algorithms are similar to prior work, we have made non-trivial modifications for both obliviousness and efficiency.

Theorem 8 shows that ControlBits$(P, 0)$ produces control bits C such that ApplyPerm(C, D) applies P to D, and ApplyInvPerm(C, D) applies the inverse of P. Note the permutation representation that P maps i to $P[i]$.

Theorem 8. *Let P be an array, with $|P| = n$, containing the values $\{0, \ldots, n-1\}$ in some order. Let $Q = (0, \ldots, n-1)$. Let P' be a copy of P (as P' will get modified), and let $C =$ ControlBits$(P', 0)$. Then, if we run ApplyPerm(C, P), then $P = Q$ afterwards, and if instead we run ApplyInvPerm(C, Q), then again $P = Q$ afterwards.*

Theorem 9 shows that WaksShuffle(D) shuffles the items of D. Similarly, Theorem 10 shows that WaksSort(K, D) sorts the keys in K in nondecreasing order and puts the data items in the same order.

Theorem 9. *WaksShuffle(D) puts the items in D in a uniformly random order.*

Theorem 10. *WaksSort(K, D) sorts the items in K in increasing order and also applies that permutation to D.*

6.4.2 Obliviousness

We next show that our algorithms are fully oblivious with respect to their secret inputs.

Theorem 11 shows that ControlBits(P, d) is fully oblivious with respect to the input permutation P. Therefore, its operation does not reveal any information about the permutation for which control bits are being computed. Note that this obliviousness holds despite the facts that i) the binary search of ForwardOrRand reveals that location of the item being looked up, and ii) the entries in the reverse-lookup hash map R are inserted in the order they appear in the forward-lookup table F and are thus observably linked to them. These properties do not violate obliviousness because the mappings in F are obliviously shuffled, the keys in R are pseudorandom, and a mapping is looked up exactly once and in only one direction (forward or reverse).

Theorem 11. *ControlBits(P, d) is fully oblivious with respect to P.*

Theorem 12 shows that ApplyPerm(C, D) and ApplyInvPerm(C, D) are fully oblivious with respect to both the input control bits C and the input data D. This property guarantees that the memory and instruction traces hide all information about what permutation is being applied.

Theorem 12. *ApplyPerm(C, D) and ApplyInvPerm(C, D) are fully oblivious with respect to C and D.*

Theorems 13 and 14 show that WaksShuffle and WaksSort are fully oblivious with respect to their inputs. They thus hide the values of the data items and the permutation that is applied.

Theorem 13. *WaksShuffle(D) is fully oblivious with respect to D.*

Theorem 14. *WaksSort(K, D) is fully oblivious with respect to K and D.*

6.4.3 Efficiency

Our algorithms are designed to be practically efficient rather than asymptotically optimal. For an input of n data items each of size β (or for sorting, where a key and data item have total size β), it is possible to perform oblivious shuffling and sorting in $O(\beta n \log n)$ time with a sorting network [AKS83], but the oblivious shuffling algorithm (ORShuffle [SJG22a]) and sorting algorithm (bitonic sort [Bat68, Ebb15]) that are fastest in practice have a $O(\beta n \log^2 n)$ runtime. We use these as our OShuffle and OSort subroutines for practicality, but we note that using a $O(\beta n \log n)$ choice here would improve the asymptotic runtime of our algorithms. Our analysis also assumes that the computation can apply instructions on machine words of size $O(\log n)$ in constant time. For obliviously setting Waksman control bits, the fastest existing algorithm currently is from Nassimi-Sahni [NS82], as made oblivious by Bernstein [Ber20], which has a runtime of $O(n \log^4 n)$.

Theorem 15 shows that the runtime of ControlBits is $O(n \log^3 n)$.

Theorem 15. *Let $|P| = n$. The runtime of ControlBits$(P, 0)$ is $O(n \log^3 n)$.*

The oblivious sorts in ControlBits are major costs in theory and in practice. Let $k = \lceil n/2 \rceil$. There are $\lceil \log_2 n \rceil$ levels of recursion in ControlBits, and in each recursive call an OSort is performed on $2k$ (fixed-size, independent of the item size β) elements to create the forward lookup table, and another OSort is performed on $k - 1$ elements to sort the input control bits. These sorts take time $O(n \log^2 n)$. All other operations in a given recursive call take $O(n \log n)$ time, dominated by the k calls to ForwardOrRand and $2k$ calls to DecUnselectedCounts, where each of those calls performs a $O(\log n)$ binary search. If OSort were implemented with a $O(n \log n)$ algorithm, then ControlBits would have a $O(n \log^2 n)$ runtime.

Theorem 16 shows that, for data items of size β each, the runtimes of ApplyPerm and ApplyInvPerm are both $O(\beta n \log n)$.

Theorem 16. *Let $|D| = n$, and let each item in D be of size β. ApplyPerm(C, D) and ApplyInvPerm(C, D) each have a runtime of $O(\beta n \log n)$.*

The expensive operations in both algorithms are applying the switches to the data items, and each resulting OSwap takes time $O(\beta)$, while there are at most $n \log_2 n$ switches total.

Theorem 17 shows that the offline runtime of WaksShuffle is $O(n \log^3 n)$, while the online runtime for items of size β is $O(\beta n \log n)$.

Theorem 17. *Let $|D| = n$, and let each item in D be of size β. The offline runtime of* WaksShuffle(D) *is* $O(n \log^3 n)$*, and the online runtime of* WaksShuffle(D) *is* $O(\beta n \log n)$*.*

The main cost during the offline phase is ControlBits, which Theorem 15 shows has a $O(n \log^3 n)$ runtime, and the runtime would become $O(n \log^2 n)$ if a $O(n \log n)$ OSort were used. The online phase is simply ApplyPerm. The β factor only appears in the online phase, and so if β is large relative to n (i.e., $\beta = \omega(\log n)$), the total runtime of WaksShuffle can be faster than directly applying the existing oblivious shuffle OShuffle. We can estimate the potential speedup in the online phase compared to the practically fastest ORShuffle (Section 4.3), which has no offline phase, by comparing the number of OSwap operations performed on the data items. In both algorithms, the only operations on the data items are OSwaps, and these operations dominate the runtime of the (online) computation. ORShuffle performs approximately $(n/4)((\log_2 n) + 1) \log_2 n$ OSwaps, and the online phase of WaksShuffle performs approximately $n \log_2 n - n + 1$, where in both cases these counts are exact when n is a power of two. Therefore, the WaksShuffle online phase performs a little over $((\log_2 n) + 1)/4$ times fewer OSwaps than ORShuffle. For $n = 2^{20}$ items, for example, the online speedup would be about $5.3\times$.

Theorem 18 shows that, when data keys are of size α and data items are of size β, the total runtime of WaksSort is $O(n \log^3 n + \alpha n \log^2 n + \beta n \log n)$.

Theorem 18. *Let $|K| = n$, $|D| = n$, each item in K be of size α, and each item in D be of size β. The (online) runtime of* WaksSort(K, D) *is* $O(n \log^3 n + \alpha n \log^2 n + \beta n \log n)$*.*

All of WaksSort is online because the keys are needed to determine the permutation used to set the Waksman control bits. The $O(n \log^3 n)$ term comes from the call to ControlBits, and it would be $O(n \log^2 n)$ if OSort were instead $O(n \log n)$. The $\alpha n \log^2 n$ term is from applying OSort to the keys K, and it would similarly reduce to $O(\alpha n \log n)$ with a $O(n \log n)$ OSort. The $\beta n \log n$ term is from ApplyInvPerm. Note that if the keys are small relative to the items (i.e., $\alpha = O(\beta / \log n)$), and the items are large relative to n (i.e., $\beta = \omega(\log n)$), then the overall runtime is $O(\beta n \log n)$, which compares favorably to the $O(\beta n \log^2 n)$ runtime of the practically efficient bitonic sort. Bitonic sort also only operates on data items through OSwaps, and it uses the same number of them as ORShuffle. Therefore, for large items we again expect to see a speedup factor in practice of a little over $((\log_2 n) + 1)/4$ for WaksSort over bitonic sort. On the other hand, when the keys are relatively large, for example when they are simply the data items themselves, we expect no runtime improvement, as a large fraction of the runtime would be consumed by simply running OSort on the keys to determine the permutation P.

Table 6.1: Analytical comparison contrasting WaksShuffle with existing oblivious shuffling algorithms.

Algorithm	Fully Oblivious	Offline Complexity	Online Complexity	Parallel
Bitonic shuffle [Bat68]	●	⊘	$\beta n \log^2 n$	●
Melbourne shuffle [OGTU14]	○	⊘	$\beta n \log n$	○
Stash shuffle [BEM+17]	○	⊘	$\beta n \log n$	○
ORShuffle (Section 5.1)	●	⊘	$\beta n \log^2 n$	●
BORPStream (Section 5.2)	●	⊘	$\beta n \log^2 n$	●
WaksShuffle (Section 6.3)	●	$n \log^3 n$	$\beta n \log n$	●

WaksShuffle+QS, however, can yield a faster online sort for any size of keys. It has a $O(n \log^3 n)$ offline runtime, and its online runtime is $O((\alpha + \beta)n \log n)$, better than the $O((\alpha + \beta)n \log^2 n)$ online runtime for bitonic sort.

We also observe that ApplyPerm and the online phase of WaksShuffle have relatively low parallel step complexity. In ApplyPerm, applying an input or output layer of switches to items of any size can be performed in a single step, and so ApplyPerm can be executed in $2\lceil \log_2 n \rceil - 1$ parallel steps. The online phase of WaksShuffle has the same step complexity. Thus WaksShuffle improves on the $(\lceil \log_2 n \rceil + 1)\lceil \log_2 n \rceil / 2$ step complexity of ORShuffle by a factor of a little more than $(\lceil \log_2 n \rceil + 1)/4$. For $n = 2^{20}$ items, for example, the speedup in online step complexity would be about $5.3\times$.

Finally, in Table 6.1, we present an analytical comparison of WaksShuffle against prior approaches towards oblivious shuffling.

6.5 Implementation

We compare WaksShuffle and WaksSort against four different algorithms for shuffling and sorting. The first and obvious candidate is (i) a bitonic sorting network [Bat68] for sorting (frequently used in the TEE literature [KKS20, DFD+21, OSF+16, MPC+18]) and bitonic shuffle for shuffling. Our implementation builds on top of our oblivious shuffling algorithms from the last chapter, which presented two fully oblivious shuffle algorithms, namely (ii) ORShuffle and (iii) BORPStream. Finally, we also compare our oblivious control bit setting algorithm for Waksman networks against (iv) Nassimi-Sahni's control bit setting algorithm [NS82]. Bernstein presented an oblivious version of the Nassimi-Sahni

control bit setting algorithm, making it a suitable candidate for fully oblivious shuffling and sorting within TEEs. We implement this algorithm ([Ber20, §7, Fig 7.1]) and evaluate its performance against WaksShuffle and WaksSort. The underlying Waksman network of our Nassimi-Sahni implementation intentionally uses the standard Waksman network layout [Wak68], so that we can contrast the performance improvements obtained by our locality-optimized Waksman network layout. We detail the results of this comparison at the end of Section 6.6.1.

To ensure that our implementation is fully oblivious, we use the Fully Oblivious Assembly Verifier (FOAV) from Section 4.1.1. Using this tool we flag variables containing public (non-secret) values as "safe" within our WaksSort and WaksShuffle implementation. FOAV verified 598 out of 688 of the conditional branches generated in the final binary to be safe. Similar to prior work, we manually inspect the remainder to ensure that these are safe as well. The bulk of the conditional branches that FOAV could not mark as safe arise from branches within the C++ implementation of vectors and hash tables. However, since non-repeating inputs fed into the hash table operations are the outputs of the PRP there is no secret data dependence underlying any usage of the hash tables themselves. Similarly for vectors, the conditional branches are associated to the size of vectors which are publicly known and not hidden.

Our implementation is open-source and publicly available [SJG23b]. We provide additional details on our implementation-level optimizations next.

6.5.1 Implementation Optimizations

We refine our implementation of WaksShuffle and WaksSort with several optimizations. To ensure fair comparisons we apply matching optimizations (whenever applicable as detailed below) for our Nassimi-Sahni implementation. First, in the SGX context performing memory allocations are expensive since they require an Asynchronous Enclave eXit (AEX) operation [CD16]. Hence we perform a one-time memory allocation procedure that allocates memory for all the Waksman subnetworks upfront with one AEX, rather than $n \log n$ recursive memory allocations. We implement this optimization for ControlBits of our Waksman algorithms and for Nassimi-Sahni as well.

For ControlBits, we instantiate both the forward and reverse lookup tables with standard C++ hash maps, and the PRP function with AES-ECB. Since the PRP has to generate unique random labels for the repeated switch number inputs it receives from different Waksman subnetworks, in our implementation every Waksman subnetwork is assigned a unique id. This subnetwork id serves as the high 64 bits of the input to our

PRP, and the switch number occupies the low 64 bits, ensuring unique random labels for the repeated inputs from different Waksman subnetworks.

Since our PRP relies on AES-ECB and uses the same key for all PRP operations, we optimize our PRP usage by maintaining the AES expanded key state and using it to perform PRPs as inputs to permute are available, instead of repeating the key expansion step for each PRP operation individually. Note that this optimization does not extend to the Nassimi-Sahni algorithm as it does not make use of a PRP. Finally to pick a random integer from a range, needed for ForwardOrRand in InBits, we use the simple technique we detailed in Section 6.2.1.

6.6 Experiments

All our experiments use a single core of a 2.3 GHz Intel 8380. Our machine runs Intel SGX2 and has 16 GB of Processor Reserved Memory. All our reported experiments and benchmarks are on a single-threaded implementation, but, as we note in Section 6.4.3, our algorithms can easily be parallelized to further reduce latency. In the experiments reported below we fix the number of items n to 2^{20} and vary the size of each item β to explore performance across different item sizes. We performed these experiments for a variety of n between 2^{15} and 2^{20}, and results for other n appear in Appendix B. Those results are qualitatively identical to those presented in this section.

For shuffling, we run experiments to contrast the performance of the four shuffling algorithms listed in Section 6.5. For sorting, WaksSort and Nassimi-Sahni generate control bits to permute the data to its sorted state. Neither of these can provide a meaningful offline phase, as the permutation to sort is dependent on the data. WaksShuffle+QS, on the other hand, can set the control bits for WaksShuffle in an offline phase, then apply the Waksman network (to shuffle) and perform a non-oblivious quicksort in an online phase. ORShuffle and BORPStream can similarly be converted into offline/online sorting algorithms by using this oblivious-shuffle-then-sort paradigm.

6.6.1 Comparing Shuffling Algorithms

BORPStream's main advantage is its ability to split the computation into two phases: i) a first phase partially shuffles items as they become available by routing those items through a well-parameterized butterfly routing network to random destination buckets, and ii) a second "online" phase compacts (i.e., discards dummy items from) each bucket

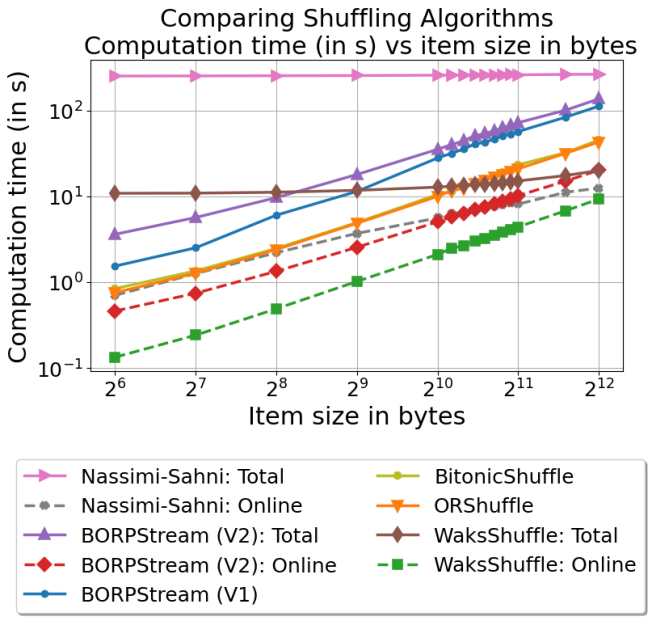


Figure 6.15: Comparing time taken for shuffling against item size with $n = 2^{20}$ items. Both axes are log scale; error bars are too small to see. Dashed lines are the online timing component for algorithms that benefit from offline/online split.

and individually shuffles each resulting bucket. This two-phase division significantly reduces the time to complete a shuffle in settings where the items to shuffle stream in over a long period of time. If all the data to shuffle is available up front, BORPStream offers little advantage as the total work done by BORPStream is more than that of ORShuffle. However, note that BORPStream V1 (tuned to minimize total time) does outperform ORShuffle for large enough item sizes due to its memory locality advantage, as the second phase shuffles small subsets of items at a time.

In comparing our work with BORPStream, we give BORPStream every advantage we can. In particular, recall that WaksShuffle works in an offline/online model, implying that it can always reap the low online cost to shuffle, independent of how the data becomes available. Nonetheless, we compare our online time with BORPStream's online time assuming items are made available in a streaming fashion, and ignoring the cost of its first phase.

Figure 6.15 presents the results of comparing shuffling algorithms, and it demonstrates that WaksShuffle requires the least online time. In comparison with BORP-

Stream (V2)'s online time, at the smallest and largest item sizes of 64 B and 4 KiB that our experiments cover, WaksShuffle's online time provides a 3.5× and 2.2× speedup respectively. However, note that BORPStream can only support this online cost in the streaming setting. In numerous application scenarios, all the data is available simultaneously (like oblivious database joins [KKS20] or oblivious file systems [DFD+21]), and in such cases BORPStream's cost falls back to its V1 variant's total time. In comparison with BORPStream V1, our online time provides speedups of $11.5\times$ and $12\times$ respectively for 64 B and 4 KiB item sizes. Similarly, WaksShuffle demonstrates a 4.6–5.6× online speedup over ORShuffle. Our improvements arise from the fact that applying the Waksman network (once the control bits are generated) is a $O(\beta n \log n)$ process in comparison to the online (and total) costs of all the other algorithms which require $O(\beta n \log^2 n)$ operations.

The oblivious Nassimi-Sahni control bit setting algorithm incurs a $O(n \log^4 n)$ cost and has higher hidden constants than the ControlBits of WaksShuffle, due to the larger number of oblivious sort invocations it requires. The cost of setting control bits is agnostic of the underlying item size, though, shown by the total runtimes that are nearly constant as the item sizes increase for both WaksShuffle and Nassimi-Sahni in Figure 6.15. However, from Figure 6.15 we also observe that ControlBits of WaksShuffle is one to two orders of magnitude faster than that of Nassimi-Sahni.

Indeed, for large enough item sizes, the entirety of WaksShuffle is faster than all the other existing shuffling algorithms even without leveraging the offline/online split, a feat that remained out of reach for the Nassimi-Sahni algorithm across all our experiments. To determine this crossover point we test additional points close to this crossover, and we observe that the crossover point occurs at 1400 B, a modest size of practical relevance for several applications. We also experimentally observe that this crossover point varies only slightly when n is varied between 2^{15} and 2^{20} (see Appendix B).

In terms of the online cost, we intentionally set up the Nassimi-Sahni implementation with the standard Waksman network topology to demonstrate the performance benefits of our topology.[1] We observe that the performance improvements from our topology are larger for smaller items, yielding up to a $5.3\times$ improvement for 64 B items and tapering down to a $1.3\times$ speedup at 4 KiB.

[1] Instantiating Nassimi-Sahni with our topology would result in an online time matching our algorithm and a negligible change in its total time.

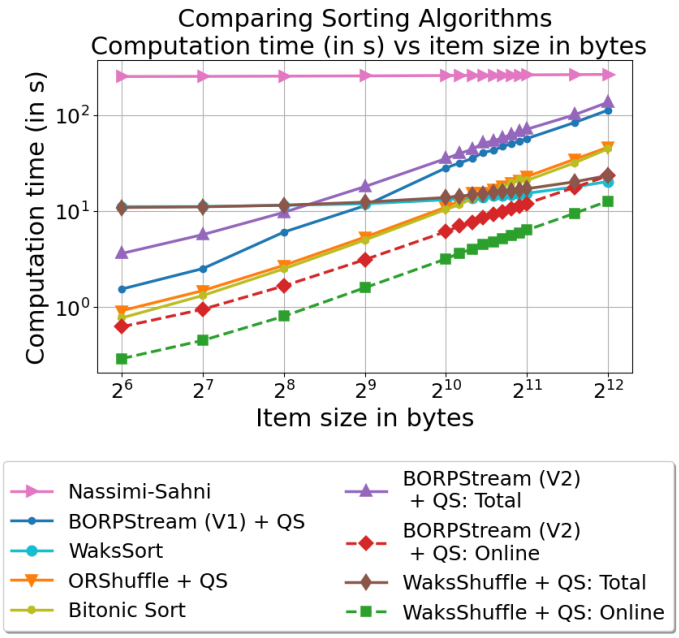
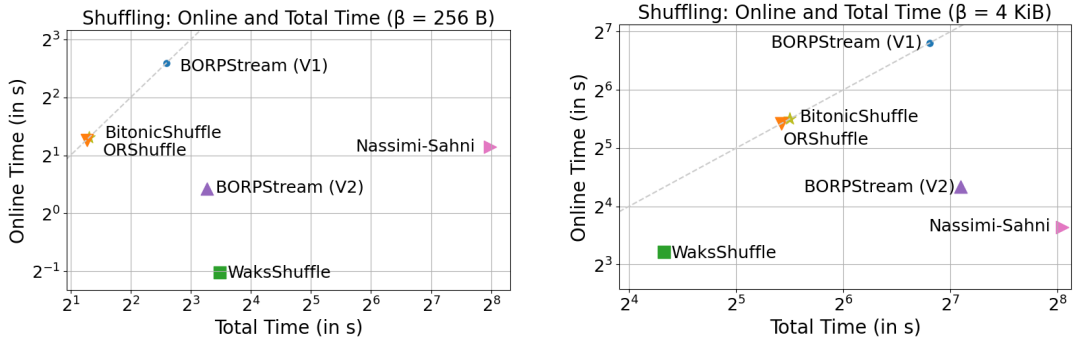


Figure 6.16: Comparing time taken for sorting against item size with $n = 2^{20}$ items. Both axes are log scale; error bars are too small to see. Dashed lines are the online timing component for algorithms that benefit from offline/online split.

6.6.2 Comparing Sorting Algorithms

We present the results of our sorting experiments in Figure 6.16. As mentioned earlier, while we can generate control bits to use the Waksman network to sort a set of items, doing so makes the control bit setting algorithm a part of the online phase, as the permutation that sorts the items depends on the keys of the items. However, for large enough problem sizes, as with shuffling above, eventually the total time for WaksSort (which is almost coincident to that of WaksShuffle+QS) is lower than all currently known oblivious sorting algorithms. Furthermore, the shuffle-then-sort paradigm enables us to generate control bits (the expensive component) of the Waksman network in an offline phase, leaving just applying the Waksman permutation network followed by a non-oblivious sort in the online phase to sort the data. We evaluate ORShuffle and BORPStream for sorting in the same manner.

While we present BORPStream's online cost in Figure 6.16, recall that this offline/online split for BORPStream only applies in settings where items to shuffle arrive intermit-



(a) Shuffling with a small item size (256 bytes).

(b) Shuffling with a large item size (4096 bytes).

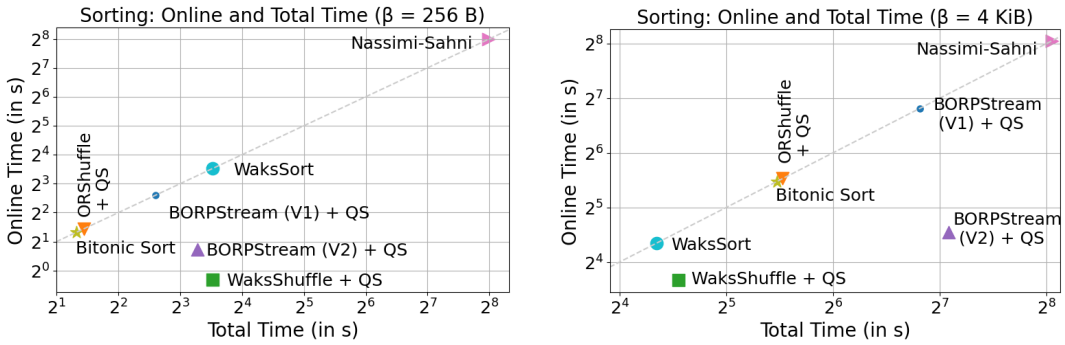
Figure 6.17: Scatter plots comparing online and total time taken (in s) for shuffling algorithms with $n = 2^{20}$. WaksShuffle consistently incurs the least online time irrespective of item size. For large enough item size ($\beta > 1400$ B), the total time of WaksShuffle is lower than all other fully oblivious shuffling algorithms.

tently and is not a generic offline/online split like that supported by WaksShuffle+QS. Assuming that the application setting is compatible for BORPStream, the online time taken by WaksShuffle+QS still demonstrates a 1.8–2.2× speedup over BORPStream's online time to sort. If all the items to sort are available simultaneously, then the best competing algorithm is in fact Bitonic Sort, over which the online time of WaksShuffle+QS provides a 2.7–3.5× speedup as the item size increases from 64 B to 4 KiB. While we focus on demonstrating improvements for a fixed value of $n = 2^{20}$ items in Sections 6.6.1 and 6.6.2, in Appendix B we provide additional experimental results that demonstrate qualitatively similar results for smaller choices of n as well.

6.6.3 Summarizing the State of Affairs

To provide a revised and concrete view of the state of affairs for fully oblivious shuffling algorithms within TEEs, we present scatter plots detailing the online and total time tradeoffs in Figure 6.17. Figure 6.17a shows shuffling algorithms with items of size 256 B, and Figure 6.17b shows 4 KiB items. Notably, WaksShuffle always incurs the least online overhead for shuffling items irrespective of the item size β; for large enough problem sizes ($\beta > 1400$ B) the total cost of WaksShuffle is lower than that of all other fully oblivious shuffling algorithms.

We do the same for oblivious sorting algorithms, and see an analogous result. As seen



(a) Sorting with a small item size (256 bytes) with $n = 2^{20}$

(b) Sorting with a large item size (4096 bytes) with $n = 2^{20}$

Figure 6.18: Scatter plots comparing online and total times (in s) for sorting algorithms with $n = 2^{20}$. WaksShuffle+QS consistently incurs the least online time irrespective of item size. For large enough items ($\beta > 1400$ B), the total times of WaksSort and WaksShuffle+QS are lower than all other fully oblivious sorting algorithms.

in Figures 6.18a and 6.18b, the online time to sort for WaksShuffle+QS is consistently and significantly lower than that of all other existing sorting algorithms. Furthermore, as the item sizes get larger, eventually ($\beta > 1400$ B) WaksSort even has the lowest total times, with WaksShuffle+QS close behind. The algorithms' online times translate to query latency, and the total times translate to server CPU costs, which can determine the monetary cost of deploying these algorithms.

6.7 Related Work

6.7.1 Oblivious Sorting Algorithms

Ebbesen's thesis [Ebb15] experimentally compares the performance of data-oblivious sorting algorithms and demonstrates bitonic sort to be the best deterministic oblivious sorting technique. In the TEE setting, bitonic sorting networks have become the standard approach for oblivious sorting [KKS20, DFD+21, OSF+16, MPC+18, PAS+20, ZDB+17]. Recently, data-oblivious sorting algorithms have received renewed interest [RS21, LS22, ALS22]. These works, however, are theoretical in nature, and while they demonstrate $O(n \log n)$ oblivious sorting algorithms, under the hood they leverage either the AKS

network [AKS83] or expander graphs [Pip93], both of which have large constants hidden in the asymptotics that make them impractical.

6.7.2 Waksman Networks

Beauquier and Darrot [BD02] show that Waksman networks can be extended to work for numbers of items n that are not powers of two. Their work gives Waksman network constructions for such non-power-of-two cases, and it compares the number of required switches to simply using the original Waksman networks for n', where n' is the next power of two greater than n.

Recently, Holland et al. [HOW22] use Waksman networks for efficient permutation of data in the client-server model. Their work is in the setting where a client outsources their data to a cloud server and wishes to permute the data obliviously. To this end, they propose using Waksman networks and provide a control bit setting algorithm to do so with minimal I/O overheads. The control bit setting algorithm is the same as Waksman's "looping" algorithm, but, in order to minimize I/O and client storage overheads, they observe that the client can both configure the Waksman network and route items through it layer by layer. Their insight is that to do so the client need only store at most $2n$ switch settings locally (instead of $n \log_2 n$). They observe that each recursive Waksman subnetwork requires only its ancestors for correct switch setting, thus one only needs to store switch settings along one recursive path of the Waksman network at a time; i.e., once the algorithm recurses into a bottom subnetwork, the control bits of its top subnetwork can be released. As for the exterior (outer) switches, they pack them along with the items in the first half of the Waksman network, thus not requiring to store these switch settings either, nor to perform any switch configuration operations while routing items through the latter half of the Waksman network. Their algorithm to set control bits has a $O(n \log^2 n)$ complexity, but their model requires private unobservable memory ($O(n)$) at the client side, which by the definition of fully oblivious is not permissible in our setting, requiring us to design a novel algorithm.

The existence of unobservable memory makes the general multiparty computation (MPC) setting easier as well. In MPC, we can assume that some parties are honest and thus have local computation (including memory accesses and instructions) that is unobservable to the adversary. In the TEE setting, as we have argued, that assumption is not valid. Therefore, MPC protocols exist that take time linear in the number of items [PSKK+23, EB22], whereas the best fully oblivious shuffling algorithms take superlinear time.

Some applications of Waksman networks in the MPC setting have been proposed. Zahur et al. [ZWR+16] revisit the original square-root ORAM by Goldreich and Ostrovsky [GO96] and leverage Waksman networks to replace the oblivious sorting primitive for efficiency improvements within the ORAM. Smart and Alaoui [STA19] demonstrate how to convert traditional single-party protocols into secret-shared counterparts, and as an application present an MPC protocol to shuffle data by setting and applying the control bits of a Waksman permutation network. In their setting, the control bits are generated for a secret-shared permutation produced by the parties participating in the MPC protocol. Both instances of these MPC-based Waksman networks use the original looping algorithm to set control bits [Wak68], modified to work over secret-shared permutations.

Code-based cryptosystems [Cho17, BCL+17] also leverage Waksman networks. They are used to obliviously apply a secret permutation during decryption. The Waksman control bits are generated during key generation and are also generated obliviously (e.g., with Nassimi-Sahni) to prevent key leakage. Our ControlBits algorithm may speed up this key-generation process in these cryptosystems.

6.8 Proofs

6.8.1 Correctness

The following are proofs of the theorems from Section 6.4.1.

Theorem 8. Let P be an array, with $|P| = n$, containing the values $\{0, \ldots, n-1\}$ in some order. Let $Q = (0, \ldots, n-1)$. Let P' be a copy of P (as P' will get modified), and let $C = \text{ControlBits}(P', 0)$. Then, if we run $\text{ApplyPerm}(C, P)$, then $P = Q$ afterwards, and if instead we run $\text{ApplyInvPerm}(C, Q)$, then again $P = Q$ afterwards.

Proof. We first show that the inputs to each Waksman subnetwork, shifted right by d, contain a Complete Residue System (CRS). That is, for each call $\text{ControlBits}(P, d)$, when either of the two calls $\text{ControlBits}(\hat{P}, d+1)$ are made, where $\hat{P}$ is a subarray of P and $|\hat{P}| = n'$, the values $\{x \gg d+1 | x \in \hat{P}\}$, taken as integers, form the set $\{0, \ldots, n'-1\}$. We recursively assume it is true for the $\text{ControlBits}(P, d)$ call, and the base case (i.e., $\text{ControlBits}(P, 0)$) holds by the requirement that P is initially a permutation on $\{0, \ldots, n-1\}$.

In the $\text{InBits}(P, d)$ call, $\text{CreateForwardLookup}(P, d)$ inserts the values $(i, P[i] \gg d)$ into the forward lookup table F, with (n, n) added if n is odd, where $n = |P|$. Let

$k = \lceil n/2 \rceil$. Then starting from element $f = k - 1$, which is necessarily an input to the top subnetwork, the sequence is calculated that $f \to g$ in F (i.e., $(f, g) \in F$), values g and r are associates (i.e., have the same value modulo k), $s \to r$ under F, and f is updated to the partner of s. In each iteration, the switch that the initial f is an input to (if any) is already set to make $P[f]$ (i.e., g) an input to the top subnetwork. Therefore, to make a CRS to both subnetworks, the associate of $P[f]$ (i.e., r), which has the same value modulo k, must be sent to the bottom subnetwork. f is updated to be the input that shares a switch with r, and then that switch is set to send $P[f]$ to the top subnetwork and r to the bottom subnetwork, as required. If a cycle is encountered during this process, a new f can be arbitrarily chosen to be sent to the top subnetwork, and the process continues.

At the end, each pair of associates in P are sent to opposite subnetworks. In the case that n is odd, the element (n, n) in F can only be queried in the reverse direction with $r = n$ because the partner of n is $k-1$, which is the initial selection for f. Therefore, this element will always end a cycle, and the value modulo k of $k - 1$ will always be sent to the top subnetwork. The top subnetwork thus obtains $\{0, \ldots, k - 1\}$ as inputs modulo k, and the bottom subnetwork obtains $\{0, \ldots, (n - k) - 1\}$ as inputs modulo k, giving each a CRS.

We next show that the outputs from each Waksman subnetwork, shifted right by d, are the sorted values $(0, \ldots, n-1)$ (i.e., a sorted CRS). We recursively assume this claim is true for the outputs of the two calls ControlBits$(\hat{P}, d + 1)$, where $\hat{P}$ is a subarray of P.

When $n = 1$, we have already shown that the shifted input is 0, and there is no output switch applied, and so the claim holds in this case. When $n = 2$, we have already shown that the shifted inputs are $\{0, 1\}$, and in ControlBits the process of shifting left and applying an OR simply moves those values to the last bits. Then the first of these two values is used to set the control bit for the only output switch, which only flips the values if they appear in the order $(1, 0)$. Then bits are then moved back to the most significant position via a right shift and an addition, yielding the desired sorted output.

For $n > 2$, we use the recursive assumption that the output of each recursive ControlBits call is a sorted CRS. We have shown that the input to the current ControlBits call was a CRS, and therefore the least-significant bits of the outputs of the recursive ControlBits calls indicate if the value changed when reduced modulo k. Each output switch takes as inputs the outputs in the same position from the subnetworks, and because those are sorted they must have been partners modulo k. Therefore, one of their least-significant bits is a zero and the other is a one. The values are flipped only if the first input to the output switch has a least-significant bit of one. Thus, after applying all output switches, the first k outputs contain a CRS (ignoring the final $d+1$ bits) in sorted

order with least-significant bits of zero, and the remaining $n-k$ outputs contain a CRS (ignoring the final $d+1$ bits) in sorted order with least-significant bits of one. The final right shifts and additions turns these values into a sorted CRS (ignoring the final d bits), as desired.

Thus, the control bits are set such that applying the resulting network to P sorts it. That is, with $Q = (0, \ldots, n-1)$, after running ApplyPerm(C, P), $P = Q$. Because ApplyInvPerm simply applies the switches in the reverse order that ApplyPerm does, it applies the inverse permutation, and so, after instead running ApplyInvPerm(C, Q), $Q = P$. □

Theorem 9. WaksShuffle(D) puts the items in D in a uniformly random order.

Proof. WaksShuffle first creates the array $P = (0, \ldots, n-1)$. Then, by assumption, the OShuffle(P) call puts P in a uniformly random order, which makes P a uniformly random permutation. By Theorem 9, the call to ControlBits$(P, 0)$ followed by ApplyPerm(C, D) permutes the items in D according to P. Therefore, WaksShuffle puts D in a uniformly random order. □

Theorem 10. WaksSort(K, D) sorts the items in K in increasing order and also applies that permutation to D.

Proof. WaksSort first creates the array $P = (0, \ldots, n-1)$. Then, by assumption, the OSort(K, P) call sorts K and P such that the items in K are nondecreasing and the same permutation is applied to P. If, by sorting K, the key in position i in K is moved to position j in K, then the item in position i in P, which has value i, is moved to position j in P. That is, P now is such that $P[j] \to j$ is the permutation used to sort K, or, put another way, P, interpreted as the permutation $i \to P[i]$ contains the inverse of the permutation used to sort K. Therefore, by Theorem 9, the call to ControlBits$(P, 0)$ followed by ApplyInvPerm(C, D) permutes the items in D according to P. □

6.8.2 Obliviousness

The following are proofs of the theorems from Section 6.4.2.

Theorem 11. ControlBits(P, d) is fully oblivious with respect to P.

Proof. Let $n = |P|$ and $k = \lceil n/2 \rceil$. We will construct a simulator $\mathcal{S}$ that, for all P and d, and conditional on an actual output C (which $\mathcal{S}$ does not get), produces indistinguishable memory and execution traces given only n and d. Such simulators exist by

assumption for the oblivious primitives we use, such as for OSelect, OSwap, OShuffle, and OSort. We recursively assume that such a simulator can be produced for smaller d. For $d \leq 2$, the required traces for ControlBits(P, d) can easily be produced because a deterministic sequence of operations is performed that depends only on n and d (note that InBits returns early in this case).

For $n > 2$, we can use our recursive assumption to simulate the traces for the recursive calls to ControlBits. Other than those calls and the call to InBits, the memory accesses and instructions executed by ControlBits depend only on n and d. Therefore we can focus on InBits(P, d), and we claim that it is fully oblivious with respect to P (but not d).

The obliviousness of InBits relies on the pseudorandomness of the PRP. We (conceptually) replace the calls to PRP with key κ_1 (i.e., those appearing in CreateForwardLookup and ForwardOrRand) with queries to a genuinely random permutation Π_1 on $(0, \ldots, 2k-1)$ (or on any larger set containing $\{0, \ldots, 2k-1\}$). The resulting executions are computationally indistinguishable by reduction to the pseudorandomness assumption. That is, otherwise an adversary could win the game defining pseudorandomness by simulating the execution while forwarding the PRP calls to the challenger, as the resulting executions would be distinguishable. Similarly, we replace the calls to PRP under κ_2 (i.e., those in CreateReverseLookup and sent to HashMapGet) with a genuinely random permutation Π_2 on $(0, \ldots, 2k-1)$ (or on any larger set containing $\{0, \ldots, 2k-1\}$).

In InBits, P is first used in CreateForwardLookup(P, d, κ). We claim that CreateForwardLookup is fully oblivious with respect to P and κ. Fix some output $F = (L, Q)$. The traces from creating and initially populating the arrays L and Q can be produced using the PRP simulator, which are indistinguishable even conditional on the values $L[i]$ that are fixed given F. The traces from the OSort(L, Q) call can be produced using its simulator, again even conditional on the values of L and Q, which are fixed given F. This proves the obliviousness claim for CreateForwardLookup.

We now consider CreateReverseLookup(F, κ), which is the next operation in InBits. We would *not* be able to say that CreateReverseLookup is fully oblivious to F and κ because its hash map insertions depend on them. However, $\mathcal{S}$ can nonetheless simulate the traces that result from the call to CreateReverseLookup, conditional on C, by choosing a random permutation (which is not necessarily the same as Π_2) and using its values on $(0, \ldots, n-1)$ (in order) as the values z used as keys in HashMapInsert$(R, z, (x, i))$. HashMapInsert is by assumption oblivious with respect to the value inserted (here, (x, i)), and so its simulator can be used to produce its traces. $\mathcal{S}$ also actually runs HashMapInsert, starting with an empty R and using the z values it has chosen, an

arbitrary value for each x, and the i taking values $(0, \ldots, n-1)$ in order. Let z_i be the ith z value, which will be needed to produce traces for the hash map queries.

The next call in InBits is to CreateUnselectedCounts$(n, \emptyset)$. This function is deterministic and is only given n, and so its traces can easily be produced by $\mathcal{S}$.

InBits continues with a sequence of back-and-forth iterations. In each iteration, the traces from assignments, arithmetic computations, and OSelect calls are easily produced. The more involved arguments are for the calls to ForwardOrRand, HashMapGet, and DecUnselectedCounts. These arguments will use the following fact about the back-and-forth sequence: each of the n mappings under P (and also the dummy (n, n) mapping for odd n) is looked up only once and in either the forward or reverse direction (but not both). This fact would actually hold whenever partners and associates are defined to each produce a matching of the values $\{0, \ldots, k-1\}$.

Consider each call to ForwardOrRand(F, U, f, b, κ_1). Regardless of the permutation P, the control bits for P that we have sampled, and the previous traces that have been produced, each call looks like a random lookup of an unselected element. That is, given the items that have been looked up in the previous or forward and reverse lookups, a subsequent forward lookup looks like a binary search for a uniformly random unselected item in F. Note that this is true in part because F and R use different random permutations on the lookup values, Π_1 and Π_2, respectively. Therefore, because each ForwardOrRand call is performed on a different value f, its location in F is uncorrelated to a lookup of the same value in R, which is possible. However, it is not possible to do a forward lookup of a *mapping* (ignoring its direction) that has been looked up already in either the forward or reverse direction. Moreover, if ForwardOrRand is called with $b = 1$, then U ensures that a uniformly random is genuinely looked up. This fact holds even given P and C has been fixed, because although P determines when cycles end and C which mappings begin a new cycle, the location of the mappings is uniformly random at each step. Thus, the traces of each forward lookup can be generated by $\mathcal{S}$ as a uniformly random unselected item.

Now consider each HashMapGet$(R, \mathsf{PRP}(r, \kappa_2))$ call. It will yield a value s such that $s \to r$ under the forward mapping. That mapping appears at some random unselected location i in F, and during the creation of R the reverse mapping was the ith insertion into R using a key z_i. We have already simulated that R and the z_i values, and so we can simulate the traces for HashMapGet given z_i. We can simulate choosing the correct z_i value simply by choosing a uniformly random index from among those containing unselected mappings. This simulation is accurate because, given P and C and the simulation so far, the next mapping is unselected and therefore appears in a uniformly random unselected item in F (or, put another way, Π_1 seen as an oracle is at that point forced to

choose a value for an unqueried value). Then, once the simulator selects where it appears in F, the previous choices force the key z_i to use as HashMapGet(R, z_i). Now given both arguments, the traces from HashMapGet can be simulated by executing that function.

For the calls in InBits to DecUnselectedCounts(U, ℓ), we observe that the location ℓ can be computed from the traces (simulated or not) of the lookups preceding the DecUnselectedCounts call (forward or reverse). In the forward direction, ℓ is the final element accessed during the binary search, and, in the reverse direction, ℓ corresponds to the z_ℓ value that has been used to call HashMapGet. The creation of U has been simulated, and U is only modified by DecUnselectedCounts. Therefore, $\mathcal{S}$ can simulate the traces of each DecUnselectedCounts(U, ℓ) call by executing DecUnselectedCounts on the values for U and ℓ that $\mathcal{S}$ has effectively already chosen.

Finally, InBits calls OSort(S, C). The traces of this call, conditioned on C, can be sampled knowing only $|S|$ and $|C|$ using the OSort simulator. This finishes the trace simulation for InBits(P, d), proving the desired claim that it is fully oblivious with respect to P. □

Theorem 12. ApplyPerm(C, D) and ApplyInvPerm(C, D) are fully oblivious with respect to C and D.

Proof. ApplyPerm and ApplyInvPerm each perform a deterministic sequence of OSwap calls that only depend on $n = |D|$. Therefore, the simulator can produce the traces for these calls knowing only n, making use of the OSwap simulator, which is assumed to be oblivious to the values of its data and swap-bit inputs. □

Theorem 13. WaksShuffle(D) is fully oblivious with respect to D.

Proof. Let D' be the shuffled output of WaksShuffle, which does depend on some unknown random bits. D' is unknown to the simulator $\mathcal{S}$. $\mathcal{S}$ can produce the traces to create the initial P, and then for the OShuffle(P) call it uses an assumed simulator to produce its traces without knowing P or what order P must end up in, given D'. By Theorem 11, there exists a simulator that $\mathcal{S}$ can use to produce the traces for the ControlBits$(P, 0)$ call, not knowing the control bits C that must be produced, given D'. Finally, by Theorem 12, there exists a simulator that $\mathcal{S}$ can use to produce the traces for the ApplyPerm(C, D) call, knowing only $|C|$ and $|D|$. □

Theorem 14. WaksSort(K, D) is fully oblivious with respect to K and D.

Proof. Let K' and D' be the sorted outputs of WaksSort(K, D). K' and D' are unknown to the simulator $\mathcal{S}$. $\mathcal{S}$ can produce the traces to create the initial P, and then for the

OSort(K, P) call it uses an assumed simulator to produce its traces without knowing K or P or what order K (and P) must end up in, given K'. By Theorem 11, there exists a simulator that $\mathcal{S}$ can use to produce the traces for the ControlBits$(P, 0)$ call, not knowing the control bits C that must be produced, given K'. Finally, by Theorem 12, there exists a simulator that $\mathcal{S}$ can use to produce the traces for the ApplyInvPerm(C, D) call, knowing only $|C|$ and $|D|$. □

6.8.3 Efficiency

The following are proofs of the theorems from Section 6.4.3.

Theorem 15. Let $|P| = n$. The runtime of ControlBits$(P, 0)$ is $O(n \log^3 n)$.

Proof. Let $k = \lceil n/2 \rceil$. Let the runtime of ControlBits$(P, 0)$ be $T(n)$.

We first consider the runtime of the call to InBits(P, d). By assumption on the runtime of OSort, the call to CreateForwardLookup(P, d, κ_1) takes time $O(n \log^2 n)$. The CreateReverseLookup call takes time $O(n)$ by assumption on the $O(n)$ time to create an empty hash map on n items and the $O(1)$ amortized time for each HashMapInsert call. The CreateUnselectedCounts call takes $O(n)$ time by a straightforward inductive argument. Each ForwardOrRand call and DecUnselectedCounts call performs a binary search and thus takes $O(\log n)$ time, and each HashMapGet call takes $O(1)$ amortized time, and therefore the k back-and-forth iterations take $O(n \log n)$ total time. The final OSort(S, C) takes time $O(n \log^2 n)$ time by assumption on the runtime of OSort and because $|S| = |C| = k - 1$. Therefore the InBits call takes $O(n \log^2 n)$ time.

Subsequently, ControlBits applies input the $k - 1$ input switches and reduces the n values in P modulo k, which together take $O(n)$ total time. The recursive calls to ControlBits take $T(k)$ and $T(n-k)$ time. Then, extracting the $n - k$ control bits takes $O(n)$ total time. Finally, applying the $n - k$ output switches and undoing the modulo reduction of P takes $O(n)$ total time.

Therefore,

$$T(n) = O(n \log^2 n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor).$$

For n a power of 2, we have that

$$T(n) = O(n \log^2 n) + 2T(n/2)$$

so by the Master theorem, $T(n) = O(n \log^3 n)$. If n is not a power of 2, let $N > n$ be the smallest power of 2 greater than n (so that $N < 2n$). Then by monoticity, $T(n) \leq T(N) = O(N \log^3 N) = O(2n \log^3(2n)) = O(n \log^3 n)$. □

Theorem 16. Let $|D| = n$, and let each item in D be of size β. ApplyPerm(C, D) and ApplyInvPerm(C, D) each have a runtime of $O(\beta n \log n)$.

Proof. Let $k = \lceil n/2 \rceil$, and let $T(n)$ be the runtime of ApplyPerm(C, D). ApplyPerm first executes $k - 1$ OSwap calls, which by assumption each take $O(\beta)$ time. Then it calls ApplyPerm on k items and on $n - k$ items. Finally, it executes $n - k$ OSwap calls, each of which take $O(\beta)$ time. Therefore,

$$T(n) = O(\beta n) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor).$$

The same reasoning as in the proof of Theorem 15 above yields that $T(n) = O(\beta n \log n)$, as required. The same argument applies to ApplyInvPerm. □

Theorem 17. Let $|D| = n$, and let each item in D be of size β. The offline runtime of WaksShuffle(D) is $O(n \log^3 n)$, and the online runtime of WaksShuffle(D) is $O(\beta n \log n)$.

Proof. In the offline phase of WaksShuffle, $O(n)$ time is required to create P, and by Theorem 15, $O(n \log^3 n)$ time is used by ControlBits. Therefore, the offline runtime is $O(n \log^3 n)$. In the online phase, by Theorem 16, $O(\beta n \log n)$ time is used by ApplyPerm. □

Theorem 18. Let $|K| = n$, $|D| = n$, each item in K be of size α, and each item in D be of size β. The (online) runtime of WaksSort(K, D) is $O(n \log^3 n + \alpha n \log^2 n + \beta n \log n)$.

Proof. WaksSort is entirely online. $O(n)$ time is required to create P, $O(\alpha n \log^2 n)$ time is assumed to be used by OSort(K, P), ControlBits uses $O(n \log^3 n)$ time by Theorem 15, and ApplyInvPerm uses $O(\beta n \log n)$ time by Theorem 16. Therefore the (online) runtime of WaksSort is $O(n \log^3 n + \alpha n \log^2 n + \beta n \log n)$. □

6.9 Conclusion

In this chapter we revisit Waksman networks and design novel fully oblivious algorithms to set their control bits and permute data items with them. Our algorithms generalize to Waksman networks over any number of items (and are not limited to powers of two unlike prior work), and we also provide a topology that benefits from better memory locality resulting in concrete performance improvements. We use these algorithms to create fully oblivious algorithms for shuffling and sorting data, two frequently used primitives

in privacy-preserving systems using TEEs as we discussed earlier in Section 2.3. The total overheads of our algorithms are lower than the current state-of-the-art algorithms for oblivious shuffling and sorting of moderately sized items (item size exceeding 1400 B). Furthermore, in the offline/online model, our algorithms provide a $O(\log n)$ asymptotic factor speedup, and they yield $> 5\times$ and $> 2.7\times$ concrete speedups in the online costs of shuffling and sorting 2^{20} items, respectively, for any item size.

Chapter 7

TEEMS

In Section 3.4.6, we discussed how no proposed MPCS design simultaneously supports the four important properties of low latency, high throughput, horizontal scalability, and asynchronicity. In this chapter, we remedy the state of affairs of metadata-protected messaging by presenting a novel MPCS design TEEMS (Trusted Execution Environment based Metadata-protected Messaging System). As suggested by its name, TEEMS leverages Trusted Execution Environments (TEEs) towards facilitating metadata-protected messaging. Existing MPCS designs predominantly reside in the distributed-trust model under non-collusion assumptions of servers; shifting the threat model to a TEE-based one has two immediate upshots that lead to our efficient system. First, all the TEE-supported servers can be run by a single service provider in one data center, resulting in low-latency and high-bandwidth links between servers of the system, while protecting metadata from even this service provider. Second, such a system does not face the deployment challenge of finding many non-colluding but cooperating entities. The challenge however with the TEE setting is to be secure against a potentially malicious service provider that may try to infer conversational metadata by exploiting TEE side channels. Hence, TEEMS focuses on the design of several new oblivious protocols that compose together into a novel metadata-protected messaging system with several practical merits.

In the literature, two distinct classes of MPCS systems that hold promise are mixnet systems [LGZ18, TGL+17, JWC+23] and Reverse PIR (RPIR) systems [VSH22, ECGZB21]. However, they have contrasting merits and shortcomings. Mixnet systems enable high throughput and horizontal scalability but lack support for asynchrony and incur higher latency overheads than the RPIR systems. On the other hand, RPIR-based mailbox systems provide asynchrony and low latency for message delivery, but have poor horizontal

scalability and throughput.

To solve these shortcomings while retaining the merits, TEEMS uses an *oblivious mailbox* architecture in which clients drop off messages, the system obliviously routes and stores them, and then clients pick up their waiting messages. Additionally, no existing MPCS designs provide solutions for parallel multiple conversations with support for asynchrony; TEEMS presents the first construction with such practically relevant usability features.

Our design introduces the notion of public and private channels, a new approach for MPCS. Both channels are encrypted and metadata-protected. A private channel provides guaranteed delivery for messages between a pair of "friends". Any user can send a message to another through the public channel, which can be used to bootstrap a private channel, but the public channel does not guarantee message delivery. This approach circumvents the rigid mechanisms of dialing protocols from the literature (Section 3.2.4), and enables users to have multiple metadata-protected conversations in parallel, in line with how messengers operate today. We implement a prototype of our design and provide benchmarks to demonstrate practicality.

Among prior MPCS designs, Sabre [VSH22] (Section 3.3.5) and Boomerang [JWC+23] (Section 3.3.3) are the closest to our work. Recall that Sabre is the current state-of-the-art iteration of a Sender Unlinkable Messaging System (SUMS) [SG24] from the Reverse PIR family of MPCS [CGBM15, ECGZB21]. In a SUMS, all users have a dedicated mailbox. In Sabre, mailboxes are distributed over two non-colluding servers that facilitate a disruption-resistant and write-private database. Senders drop their message into their intended receiver's mailbox using PIR in reverse (private writing instead of the private reading of typical PIR), preventing an adversary from linking messages with their destination mailboxes. Clients periodically retrieve messages directly from their mailbox, without having to hide which mailbox belongs to them. A SUMS can trivially be converted into a CUS, by requiring all participants send and retrieve messages in fixed communication rounds.

Similarly, Boomerang is an efficient representative of a CUS from the line of MPCS constructions based on mixnets and differential privacy [VDHLZZ15, TGL+17, LGZ18]. Additionally, Boomerang leverages TEEs to design a system that has cryptographic security under hardware trust without having to use differentially private guarantees the way its predecessors do. Initiating a conversation requires clients to run a separate dialing protocol. The frequency of dialing to conversation rounds introduces a tunable usability-performance trade off like prior works that require dialing for bootstrapping conversations [VDHLZZ15, AKTZ17, TGL+17, LGZ18, KLD20, EB22, JWC+23]. Ultimately, like its predecessors, Boomerang cannot support client asynchrony, and assumes all clients

will remain online, periodically dialing, and sending and receiving a message in every communication round.

7.1 Overview

In this section, we give a high-level overview of TEEMS and introduce the components and parameters of TEEMS. For the readers' convenience, Table 7.1 lists the parameters used to configure an instance of TEEMS. Each parameter is a number that is set system-wide at initialization and is constant during system operation. Table 7.2 summarizes the remaining notation used in this chapter. These variables are just used by the paper to describe the system and its operation.

Table 7.1: The parameters in TEEMS

Parameter	Description
b	The maximum number of public messages a user can receive
e	Friend request expiration period
f	Maximum size of a user's friend list
g	Number of ingestion servers in a given channel
ℓ	The length of a message
m_1	Max messages in a short-term mailbox
m_2	Max messages in a long-term mailbox
r	Number of routing servers in a given channel
s	Number of storage servers in a given channel
ϕ	Frequency of friend operations

Table 7.2: Notations used for TEEMS

Notation	Description
a	Fraction of users controlled by the adversary
A	An account server
C	A client
c_I	Number of client interactions this epoch at ingestion server I
d	The number of servers done sending inputs this round
F	A friend request
G	A friend response
h_R	Fraction of users sending to accounts assigned to $\mathcal{S}^R$
$\mathcal{I}$	The set of ingestion servers in a given channel
I_i	The ith ingestion server in a given channel
$\mathcal{I}_R$	The set of ingestion servers in a given channel that forward messages to routing server R
I_u	The ingestion server assigned to user ID u in a given channel
K	The global symmetric key
M	A list of messages
$\mathcal{M}$	A message
n	Total number of users in the system
$\mathcal{R}$	The set of routing servers in a given channel
R_i	The ith routing server in a given channel
$\mathcal{S}$	The set of storage servers in a given channel
S_i	The ith storage server in a given channel
$\mathcal{S}_R$	The set of storage servers in a given channel that routing server R forwards messages to
S_u	The storage server in a given channel assigned to user ID u
t	The MAC tag in a long user ID
u, v	User IDs
ζ	A nonce
η	Current epoch
μ	Length of MAC tag in long user ID
π	Boolean indicating if the channel is private
ρ	Current round
τ	Length of a token

7.1.1 System Design and Goals

TEEMS uses an *oblivious mailbox* MPCS design. That is, senders submit messages, which the system stores and then delivers to receivers while remaining oblivious to which users are communicating. Each user has a *mailbox* that will store messages until the user connects and retrieves them. TEEMS does not hide which mailbox belongs to which user, but it does hide if the mailbox receives or contains messages at all, as well as hiding the senders of any such messages. The system operates in synchronous *epochs*. However, the system provides *asynchronous* communication to users, and users may, at any time, retrieve messages stored in their mailbox and submit messages to be stored in the receiver's mailbox by the next epoch.

TEEMS uses TEEs in its servers for metadata protection. The main process in each server runs in a TEE enclave, with any necessary function calls outside the enclave (e.g., to send or receive messages) operating only on non-secret or encrypted data. Remote attestation to the server enclave processes is performed by other servers and clients, but it is done only once to exchange keys that are used from that point on (see Section 7.2.2).

In contrast to distributed-trust based MPCS models, the use of TEEs enables TEEMS to run its servers in the same data center by the same operator while still achieving its security goals. Indeed, such a setup minimizes the number of different parties that are in a position to perform the physical attacks that TEEs are vulnerable to. Therefore, we assume TEEMS runs on a set of TEE-enabled servers within the same data center. As is appropriate for that setting, we can assume that the servers have high-bandwidth, low-latency connections among them.

All TEEs in the system share a symmetric key K that is generated by them and securely stored. This key enables bootstrapping of secure forward-secret enclave-to-enclave communication channels. It is also used during account creation and other tasks as outlined below. Each server generates a public-private key pair, which are used to secure all communication among servers and between clients and servers. Details of key establishment appear in Section 7.2.2.

Threat Model. In our design, we consider an adversary that has OS-level control of the servers, controls network communication with clients and among servers, and controls some of its own clients. Following the TEE threat model, the adversary can perform software attacks on the TEE machines, including the observation of side channels intentionally left open by the TEE, such as timing and memory-access locations. In addition to observing all communications, the adversary can delay, modify, or drop packets arbitrarily. While the adversary may control some clients, the adversary must be a third party to the protected communication. That is, the adversary cannot control the sender

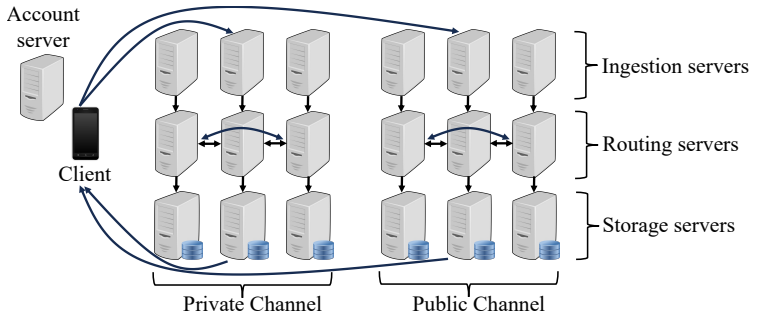


Figure 7.1: An example of the TEEMS architecture. Arrows indicate how messages are sent. Storage servers are shown holding mailboxes with stored messages. Just one client is shown, and it connects to the ingestion server and storage server that it is assigned to in each channel.

or the receiver for our security guarantees to apply.

Like other TEE-based systems [BEM+17, SO19, DFD+21, JWC+23], denial-of-service attacks are out of scope for this work. Microarchitectural attacks like those based on power consumption [MOG+20, LKO+21, CVM+21] or speculative execution [VMW+18, CCX+19] are orthogonal to our work, and receive mitigations from hardware manufacturers directly [Int18, Int19, Int20]. Physical attacks, such as invasively probing the TEE chips to extract keys, are also outside of the threat model.

Goals. A main security goal of TEEMS is unobservable communication. The adversary should not be able to detect if a message is sent at all. Even if the adversary knows some message has been sent, it should not be able to determine the sender, receiver, or message contents. Moreover, the system should ensure integrity of the delivered message, and a receiver should be also able to determine who sent the received message and when.

A main performance goal is horizontal scalability. The system should be able to handle increasing numbers of users by increasing the number of servers. The system should also provide sufficiently low messaging latency for use in interactive conversations, and its client resource requirements should be compatible with those available on typical mobile devices.

7.1.2 Private and Public Channels

The high-level system architecture is illustrated in Figure 7.1. In TEEMS, messages can be sent either through the *private channel* or *public channel*. Each channel type is composed of a separate set of servers, and clients communicate with each set to receive messages on both channels. The same number of servers of each type are present in each channel. The private channel guarantees delivery within one epoch (subject to sufficient room in the receiver's mailbox), but its senders must be selected in advance to share the limited system resources. The public channel does not guarantee delivery, but no prior coordination is required to use it. The public channel thus can be used to send someone a message knowing only their user ID. Because a large number of users may try to send a public-channel message to the same receiver at the same time, however, the public channel may drop some messages to respect the resource limits imposed by the privacy requirement to hide how many messages a user receives.

For private-channel communications, each user u can select up to f other users as their *friends*. Each of those friends will be allowed to simultaneously send a private-channel message to u in the same epoch. Friend designations are persistent across epochs and need no further action from the user until they desire to change them. Friends are also symmetric in that if u is a friend of v, then v is a friend of u. The symmetric relationship will facilitate updating friend relations, as it allows such updates to be sent through the private channel. A client must obtain *tokens* before sending private-channel messages to authorize the use of the private channel for the message receivers. A token is a credential, valid for the current epoch, showing that a given sender and receiver are friends.

In contrast, no such authorization is required to send public-channel messages. Each user is limited to sending at most one and receiving at most b public-channel messages in a single epoch. A public-channel message may be dropped, but, if not, it will be delivered within that epoch. Public-channel messages will have a priority attached to them by the system, based on how frequently a user has sent other public-channel messages, and the system will prefer to drop lower-priority messages when dropping is necessary.

The public channel is also used to set up and modify friend relationships. Friend requests are sent through the public channel, which can then be confirmed through the private channel. Friend revocations are sent through the private channel, which is possible because the friend relation is symmetric.

This two-channel design provides several improvements over prior MPCS approaches. The public channel circumvents the requirement of a dialing protocol that many MPCS schemes require. Dialing protocols add latency and rounds of overhead to actual com-

munications. Moreover, dialing only allows for online metadata protections; once clients go offline, they can no longer participate in a dialing protocol, and consequently they cannot receive messages asynchronously. The private channel in TEEMS requires a one-time setup to establish a friendship, which can be accomplished asynchronously, and then afterwards requires no coordination at all. We note that either the private channel or the public channel could be used by an MPCS in isolation. The public channel is completely independent of the private channel, but the private channel would need some other method to set up friendships.

7.1.3 Clients

Clients in TEEMS follow a fixed connection schedule to send and receive messages. The schedule can be randomized and may vary across clients, but it should only depend on information that the adversary already knows (or is allowed to know). In particular, it should not depend on if any message was actually sent or received during prior connections. Clients will send and receive *dummy* messages during a connection to hide how many real messages were sent and received. To provide low latency when the client is messaging, the connection schedule should have little time between connections. However, if a client is unavailable at a scheduled connection time, such as due to their device being off or disconnected from the network, this deviation from the schedule need not affect the security guarantees, as long as the unavailability is not related to their prior messaging behavior.

At each scheduled connection time, the client retrieves its stored messages from each channel and sends any waiting messages from the user to each channel. Every epoch, each online client will receive exactly f messages on the private channel and b on the public channel, and it will send exactly one message over each channel, where any of those messages may be dummies.

7.1.4 Servers

There are four types of servers in TEEMS:

Account server: An account server facilitates the creation and modification of user accounts. It takes requests for new accounts from users and assigns them ingestion and storage servers.

Ingestion server: Each client in TEEMS is assigned an ingestion server, and this assignment is considered public. An ingestion server collects messages from its clients.

At the start of each epoch, these servers forward their messages to the routing servers.

Routing server: The routing servers jointly perform an oblivious route and pad protocol. At the end of the protocol, the routing servers end up with the exact number of maximum messages designated for each clients' private and public mailbox. The routing servers then forward these messages to the storage servers.

Storage server: Each mailbox in the system is assigned to a storage server, and the system makes these assignments such that each storage server has about the same number of mailboxes. The system does not try to hide the mapping between clients and their mailbox, including both its storage server and its location on that server. Online clients will receive all the messages from their mailbox every epoch, while offline clients will have their messages accumulate (up to a maximum size), and will receive their mailbox the next time they connect.

We note that the servers are logical entities, and physical machines could run multiple servers. For example, in an initial deployment, one physical machine could run all the servers, and then the servers could be migrated to multiple machines as the number of users grows. Running multiple servers of the same or different type on a many-core machine is beneficial, as each server uses only a small number of cores, as will be seen in Section 7.4.

7.2 Design Details

We assume that the underlying network provides asynchronous, message-oriented communication channels between each pair of servers and between each client-server pair. Let $\mathcal{I}$, $\mathcal{R}$, and $\mathcal{S}$ be the sets of ingestion servers, routing servers, and storage servers, respectively, for a given channel. When necessary, we will use superscripts to indicate the channel (e.g., $\mathcal{I}^{\text{priv}}$ and $\mathcal{I}^{\text{pub}}$).

Let $r = |\mathcal{R}|$ be the number of routing servers, and let $s = |\mathcal{S}|$ be the number of storage servers. Let the routing servers be ordered: $\mathcal{R} = \{R_1, R_2, \ldots, R_r\}$. For a given user u, let I_u and S_u be the ingestion server and storage server assigned to u, respectively. The routing servers each have a set of ingestion servers that forward messages to them and a set of storage servers that they forward messages to. No ingestion server forwards to more than one routing server, and no storage server is forwarded to by more than one routing server. For a given routing server R, let $\mathcal{I}_R$ be the set of ingestion servers that forward messages to R, and let $\mathcal{S}_R$ be the set of storage servers that R forwards messages to. We assume for simplicity that the same number of ingestion servers forwards to

each routing server and that each routing server forwards to the same number of storage servers. Let the storage servers be ordered, $\mathcal{S} = \{S_1, S_2, \ldots, S_s\}$, such that for all $R \in \mathcal{R}$ the servers in $\mathcal{S}_R$ have a consecutive numbering.

7.2.1 User Identifiers

Each TEEMS client has a *user identifier* (user ID). The user identifier is a value composed of the index of the storage server (under a canonical ordering) followed by a unique number for that client. Both of those components are of fixed sizes (e.g., 10 bits for the server index followed by 22 bits for the per-server user number). The largest value that can be represented in that fixed size is reserved for internal use. Each identifier is thus unique. Its structure also facilitates routing because sorting messages by the receiver's user ID simultaneously sorts them by the index of the receiver's storage server and by the index of the routing server that forwards to that storage server. The user identifier may be guessable, and so, to give the user some control over who can send them messages on the public channel, the system maintains a *long* form of the user ID that also includes the MAC of the user ID under a key derived from the shared key K. The users must therefore distribute their long user IDs in order to receive public-channel messages, which they may accomplish narrowly (e.g., only to select contacts) or broadly (e.g., publicly via a website). To prevent ambiguity, we will always specify when the long form of the user ID is the one being used.

7.2.2 Account Server

The account server coordinates key establishment for the system and facilitates user-account management. It maintains the following information about the other servers in the system: (1) their addresses; (2) their public keys; and (3) a count of the number of users assigned to each ingestion server and storage server. It also creates the system-wide key K. The other types of servers in the system use a one-time two-way remote attestation to securely connect to the account server and obtain K. That key is then used to secure a separate connection on which the public key of the server is shared with the account server and the public keys of the other servers are obtained from the account server. Clients use a one-time remote attestation with the account server to securely obtain its public key, which they then use to secure communications with it.

The account server stores an account for each user in the system. Each account consists of the following: (1) the long user ID; (2) authenticating information, such as

password hash or a public verification key; (3) the assigned ingestion server; and (4) the assigned storage server. This data is expected to be relatively small (less than 100 bytes) and infrequently accessed. The number of servers is expected to be many thousands of times less than number of users. We expect that a single account server can accommodate billions of users, although the account database could also simply be split across servers. It is not necessary to hide which account is being accessed because only the owner accesses it.

The account server A supports the following queries: CREATE, UPDATE, and DELETE. Each of these queries is made by a user to create or modify their account. In CREATE, the client provides its authenticating information α. A chooses the least-loaded ingestion servers, I^{priv} and I^{pub}, and storage servers, S^{priv} and S^{pub}. It contacts S^{priv} to obtain an available user ID u (users are assigned the same server indices in each channel). Recall that the high bits of u will be the index of S^{priv}. A sends (u, α) to I^{priv}, I^{pub}, S^{priv}, and S^{pub}, which they all store. Then A computes a MAC tag t on u under the global shared key K. It subsequently sends to the client the following: (u, t), the assigned servers $(I^{\text{priv}}, I^{\text{pub}}, S^{\text{priv}}, S^{\text{pub}})$, and the public keys of those servers. In UPDATE, user u submits new authenticating data α, which A updates locally and forwards to the assigned servers of u. In DELETE, A deletes the account of the user u and forwards the query to the assigned servers of u for them to delete. None of these queries is performed obliviously, and the system does not hide if or how accounts are created or modified.

7.2.3 Messages

A message is a data structure containing metadata of the message as well as the content. Let $\mathcal{M}_{\text{src}}$ be the user ID of the source of a message, $\mathcal{M}_{\text{rec}}$ be the user ID of the receiver, $\mathcal{M}_{\text{pri}}$ be the message priority, and $\mathcal{M}_{\text{data}}$ be the content. A private-channel message is $\mathcal{M} = (\mathcal{M}_{\text{rec}}, \mathcal{M}_{\text{src}}, \mathcal{M}_{\text{data}})$, and $\mathcal{M} = (\mathcal{M}_{\text{rec}}, \mathcal{M}_{\text{pri}}, \mathcal{M}_{\text{src}}, \mathcal{M}_{\text{data}})$ for a public-channel message. The entire message has a fixed total length of ℓ (for both the private and public channels) to hide the true length of $\mathcal{M}_{\text{data}}$. Note that $\mathcal{M}$ may be a *dummy* message, indicated by $\mathcal{M}_{\text{src}} = \perp$, or a *real* message, in which $\mathcal{M}_{\text{src}}$ is a user ID. Dummy messages are sent by clients and between servers as needed to hide the number of real messages being sent.

7.2.4 Clients

Clients send and receive messages according to an interaction schedule. The schedule may be unique to the client and need not be coordinated with any other client. However,

Figure 7.2: ClientInteract($\mathcal{M}_{\text{priv}}, \mathcal{M}_{\text{pub}}, u$): Interaction procedure executed by user u. Retrieves messages; sends private-channel message $\mathcal{M}_{\text{priv}}$ and public-channel message $\mathcal{M}_{\text{pub}}$.

Trigger: Timer goes off for next scheduled interaction.
Algorithm:

1. Retrieve private-channel messages and tokens from S_u^{priv}.
2. Retrieve public-channel messages from S_u^{pub}.
3. Send $\mathcal{M}_{\text{priv}}$ with related token, or a dummy message and token if no such pair exists, to I_u^{priv}.
4. Using long IDs for receivers, send $\mathcal{M}_{\text{pub}}$, or dummy message if $\mathcal{M}_{\text{pub}}$ is empty, to I_u^{pub}.

the schedule must be independent of the actual client messaging behavior, that is, it should not depend on if a client has a real message to send or receive. Note that a client's schedule can thus accommodate periods of disconnection, such as when a mobile device is turned off or loses network connectivity.

During each interaction, the client sends and receives messages via both the public and private channels. Some ordering among those actions is needed, for example retrieving a token before sending a private-channel message. Figure 7.2 shows the ClientInteract algorithm that a client uses during an interaction. Note that Steps 1 and 2 can be performed in parallel, as can Steps 3 and 4.

7.2.5 Ingestion

Ingestion servers in both channels first authenticate a connecting client using the stored user IDs and authentication data. The ingestion servers receive a list of messages from the user, which they validate by checking that the source user ID is correct and then store until the next epoch starts. When the next epoch starts, each ingestion server forwards its stored messages to its routing server.

In the private channel, the client always sends one message and token, and if the client has none, it adds a dummy message and token. The ingestion server validates the token by checking that it contains the same receiver ID as the message, that is valid for

the upcoming epoch, and that it has a valid MAC tag. Any invalid messages are replaced with dummy messages. The tokens are discarded and are not forwarded further.

In the public channel, the client always sends one message, using a dummy message if necessary. The client includes the long user ID (i.e., with a MAC tag) in that message. The ingestion server validates the MAC tag in the receiver long user IDs, turning the message into a dummy message if the check fails. Then it discards that MAC tag from the message, as it is no longer needed. A public-channel ingestion server also maintains a record of the last time a real message was sent by each of its assigned users, which will be used to assign priorities to help determine which messages to drop, if necessary. That record is updated obliviously when a message is received from a client. When a message is received, the ingestion server copies into the message (using space in $\mathcal{M}_{\text{data}}$ reserved for a priority value) how many epochs ago the user last sent a public-channel message. This value gives higher priority to messages from users that have not recently used the public channel.

7.2.6 Routing

We use similar routing algorithms for public and private channels. Our routing algorithm, ColumnRoute, is derived from columnsort [Lei84], which has been used in other TEE-based distributed systems [ZDB+17] to provide oblivious distributed sorting. Distributed sorting is similar to our routing problem, in that we are attempting to sort messages by their receiver's storage servers. The routing problem, however, is slightly easier. In a distributed sort, the value of a given record's sorting key in relation to all the keys being sorted determines the final server it should end up at. In the routing case, the recipient user ids directly imply the server the message should end up at. We can therefore obtain some efficiency improvements, including in particular a reduction from four rounds of communications to three (or two noting that the third round is a simple forwarding between server types).

However simply routing all the messages in a round to their respective storage servers is not sufficient. This trivially reveals the distribution of messages across storage servers and even the active recipients of messages in the system if one naively delivers messages to their recipients' mailbox. Hence TEEMS presents an *oblivious-pad-and-route* protocol. Starting with a batch of messages for a round, the objective of the oblivious-pad-and-route protocol is for the routing servers to simultaneously route and generate sufficient padding messages in a distributed fashion obliviously for this batch, such that at the end of the protocol all storage servers (and even each client within each storage server)

Figure 7.3: State and events of the ColumnRoute algorithm for private- or public-channel routing.

State

- i: Routing server index (i.e., server is R_i)
- π: Boolean indicating if the channel is private
- ρ: The current round number (initialized to 1).
- d: The number of servers done sending messages this round (initialized to 0).
- M: A list of messages to process in the current round (initialized to $\emptyset$).

Events

- Message $\mathcal{M}$ received.

---

receives exactly the same number of messages; all while hiding which messages are real or dummy from the adversary.

Let n be the total number of users. ColumnRoute is a distributed and asynchronous algorithm, and so we present the state maintained in Figure 7.3 and then give routines that are triggered upon the receipt of a message in Figures 7.4–7.6. The state maintains the current *round* ρ, each of which consists of receiving messages, performing computation, and then sending messages. We also note that the private-channel routing servers set the π flag to true, and public-channel servers set it to false.

In round one (Figure 7.4), routing server R_i receives messages from its ingestion servers $\mathcal{I}_{R_i}$. The number of messages from each depends on the number of clients that have sent messages since the last epoch, and so the ingestion servers indicate all messages have been sent with a `DONE` message. It is safe to reveal this number, because the adversary is allowed to observe the clients connecting to the ingestion servers. R_i then obliviously sorts the messages by the receiver's user ID, which also puts them in order of the storage servers of the receivers because each user ID begins with the index of that user's storage server. ColumnRoute then performs a "scatter" of the messages, distributing the messages in round-robin order to the routing servers. This step (corresponding to the "transpose" step of columnsort) scatters the messages intended for each storage server nearly evenly across the routing servers.

In round two (Figure 7.5), R_i receives messages from the other routing servers, and the number is again variable (but assumed known to the adversary) depending on which ingestion servers forward to a given routing server. The server obliviously counts how

many messages have receivers at each storage server via a linear scan on the messages. In the private-routing case, the tokens have already enforced that no user receives more than f messages in an epoch, and therefore, due to the even scattering from the previous step, we can be sure that there are at most $y = \lfloor f\lceil n/s \rceil / r \rfloor + r$ messages for each storage server. The r additive term is due to the scatter step imperfectly distributing messages, possibly leading to two routing servers differing by at most r in the number of messages with receivers assigned to a given storage server. This added number of messages is sufficient to ensure each message can be routed to the correct routing server in this step by argument analogous to that for the correctness of columnsort [Lei84]. The server obliviously computes the number of dummy messages needed to reach the value y for each storage server, adds that number of dummies at the end, and then obliviously shuffles the messages. Since there are now guaranteed to be exactly y messages destined for each storage server, the shuffle allows us to take advantage of the Scramble-then-Compute [DDCO17] paradigm to *non-obliviously* send each message in the shuffled buffer to the routing server that forwards to the storage server for its receiver.

In the public routing case for round two, there may be too many messages going to a single storage server to accommodate scalably (e.g., all messages are real and have the same receiver). If there were no such excess messages, then, due to the scatter, the routing server would be guaranteed to have at most $y = \lfloor b\lceil n/s \rceil / r \rfloor + r$ messages with a given receiver storage server. To remove messages in excess of this number, the routing server orders the messages by increasing storage server index, and then, for messages with a given receiver storage server, converts the lowest-priority excess messages to dummies. This conversion can be done in place because every excess message for some storage server implies a deficit for another storage server, and so the routing server uses this fact to update the receiver storage server of the new dummy messages such that ultimately there are at most y messages with any given receiver storage server. This is the same guarantee as for the private channel case, and so the public channel routing server proceeds with the private channel routing logic. Note that because excess messages are determined locally, the dropped messages are not guaranteed to have the lowest priority globally.

In round three (Figure 7.6), the routing server first waits for a known number of messages to arrive. Recall that the aim of the oblivious-pad-and-route protocol is to have the storage servers receive exactly f or b messages for each client in the private and public channel case respectively; i.e., a total of $\lfloor f\lceil n/s \rceil \rfloor$ or $\lfloor b\lceil n/s \rceil \rfloor$ messages at each storage server. But due to the imperfect scatter step in round 1 (followed by the padding step in round 2), routing servers in round 3 receive up to $r(\lfloor f\lceil n/s \rceil / r \rfloor + r)$ or $r(\lfloor b\lceil n/s \rceil / r \rfloor + r)$ messages in the private and public channel cases respectively. Hence

Figure 7.4: ColumnRoute$_1(\mathcal{M})$: First round of ColumnRoute, used for private- or public-channel routing.

Trigger: $\rho = 1$ and message $\mathcal{M}$ received from $I_j \in \mathcal{I}$.

Algorithm:

1: If $\mathcal{M} \neq$ `DONE`, append $\mathcal{M}$ to M and **return**.
2: $d \leftarrow d + 1$, and if $d < |\mathcal{I}_{R_i}|$ then **return**.
3: Obliviously sort M in increasing order by the user ID of the receiver.
4: For $1 \leq x \leq |M|$, $y \leftarrow (x \bmod r) + 1$. Send $M[x]$ to R_y.
5: Send `DONE` to each R_j.
6: $d \leftarrow 0$, $M \leftarrow \emptyset$, and $\rho \leftarrow 2$.

Figure 7.5: ColumnRoute$_2(\mathcal{M})$: Second round of ColumnRoute, used for private- or public-channel routing.

Trigger: $\rho = 2$ and message $\mathcal{M}$ received from $R_j \in \mathcal{R}$.

Algorithm:

1: If $\mathcal{M} \neq$ `DONE`, append $\mathcal{M}$ to M and **return**.
2: $d \leftarrow d + 1$, and if $d < r$ then **return**.
3: If π is true: Let $y = \lfloor f\lceil n/s \rceil / r \rfloor + r$.
4: If π is false: Let $y = \lfloor b\lceil n/s \rceil / r \rfloor + r$.
5: Via a linear scan of M, obliviously count, for each storage server S_j, the number x_j of messages in M such that S_j is the receiver's storage server.
6: For each storage server S_j, let $z_j = y - x_j$.
7: If π is false: Obliviously sort M in increasing order by the receiver's user ID and then the message priority.
8: If π is false: Via a linear scan of M, for every S_j with $z_j < 0$, obliviously convert the first $-z_j$ messages with a receiver storage server of S_j to a dummy message. When converting a message to a dummy, obliviously choose an arbitrary storage server S_k currently with $z_k > 0$ and set it as the receiver's storage server, afterwards obliviously decrementing x_j and z_k and incrementing x_k and z_j.
9: For each storage server S_j, obliviously append z_j dummy messages to M with S_j as the receiver's storage server.
10: Obliviously shuffle M, and then, for each $\mathcal{M} \in M$, send $\mathcal{M}$ to the routing server that forwards messages to the storage server for $\mathcal{M}_{\text{rec}}$.
11: $d \leftarrow 0$, $M \leftarrow \emptyset$, and $\rho \leftarrow 3$.

Figure 7.6: ColumnRoute$_3(\mathcal{M})$: Third round of ColumnRoute, used for private- or public-channel routing.

Trigger: $\rho = 3$ and message $\mathcal{M}$ received from $R_j \in \mathcal{R}$.
Algorithm:

1: If π is true: Let $y = r(\lfloor f\lceil n/s\rceil/r\rfloor + r)$, and let $x = f\lceil n/s\rceil$.
2: If π is false: Let $y = (\lfloor b\lceil n/s\rceil/r\rfloor + r)$, and let $x = b\lceil n/s\rceil$.
3: Append $\mathcal{M}$ to M. If $|M| < (s/r)y$, **return**.
4: Let $z = y - x$. Let v be the reserved value that is the highest possible ID. Via a linear scan through M, for each $S \in \mathcal{S}_{R_i}$, set to v the receiver user ID of the first z dummy messages that have S as the receiver storage server.
5: Obliviously shuffle M, and, for each $\mathcal{M} \in M$, send $\mathcal{M}$ to the storage server for $\mathcal{M}_{\text{rec}}$, skipping those with a receiver user ID of v.
6: $M \leftarrow \emptyset$, and $\rho \leftarrow 1$.

---

the routing server first reduces the messages with the same receiver storage server by eliminating the added r from the scatter process (Step 4). This is possible because, while those extra messages might be needed by some routing servers in the previous round, we know that after aggregation they must all be dummy messages. Next, the routing server obliviously shuffles the messages to hide which messages were removed in the previous step, which could reveal information about how many real messages were received from each routing server in the previous round (and ultimately information about how many real messages each ingestion server had received). Finally, the routing server simply forwards each message to the storage server of its receiver.

We note that in TEEMS as presented all routing is accomplished by the routing servers. However, we can reduce the rounds and communication by having ingestion and storage servers involved in the first and second rounds of routing, respectively, instead of merely forwarding messages to and from them. This variant can also reduce computation by eliminating the oblivious shuffle in round three of ColumnRoute, because it is made redundant by the initial message sort done by the storage server. We use this variant in our implementation because it is faster in the small deployments we consider experimentally. The original variant, on the other hand, is both simpler to describe and improves bandwidth utilization in larger deployments when throughput limits on the all-to-all communication pattern are reached.

7.2.7 Storage

Storage servers in each channel maintain a mailbox for each user assigned to them. Each mailbox has a short-term and a long-term component, where messages are initially stored in the short-term mailbox and are eventually moved to the long-term mailbox if not yet sent to the client. The short-term and long-term mailboxes are both simple lists of messages. However, the short-term mailbox is an append-only list that starts empty and increases to a maximum length of m_1 messages, while the long-term mailbox is fixed-size list of m_2 messages. The private-channel storage servers also maintain, for each of their users, a list of the user IDs of their friends.

During an epoch, a storage server receives a list of messages from its routing server. The server attaches a current timestamp to each of the received messages to enable clients to determine when the message was received. The storage server also obliviously sorts the messages by increasing receiver user ID and priority to facilitate inserting them into the mailboxes.

In the public channel, the storage server might have for a given user u more than the maximum b messages that u can receive in an epoch. The messages are already sorted by user ID and priority, and so the storage server can drop the excess by performing a linear scan in reverse and, after finding b for a given user u, turn those afterwards also for u into dummies. The server then obliviously compacts the real (non-dummy messages) to the beginning of the list, which results in the dummy messages compacted to the end of the list.

Finally, storage server S_j counts the number of messages intended for each user via a linear scan. It performs a reverse compaction so that the messages whose receiver is the ith user assigned to S_j appear in sequence starting in position $f \cdot i$ (or $b \cdot i$ for public routing), where the dummy messages will end up filling any gaps between those positions. Then, via another linear scan, S_j appends all of the messages for each user, including dummy messages, to the short term mailbox. When that short-term mailbox exceeds m_1 messages, the storage server concatenates it with the long-term mailbox, obliviously compacts the result to move any dummy messages to the end, and then drops enough messages at the end to make it of length m_2.

When a client runs ClientInteract, the storage server sends the client a copy of the short-term mailbox and deletes all messages from it. If the short-term mailbox has been emptied into the long-term mailbox since the last client interaction, the server also sends the client a copy of the long-term mailbox and then overwrites the long-term mailbox with dummy messages.

7.2.8 Friends

Friend relationships are symmetric, and the process to establish one begins with a friend request. The request must be accepted within e epochs of the request being issued, where e is set to create a relatively short expiration period (e.g., a day). Several of the steps involved in managing friend requests require extra communication or extra computation by the various parties involved in the process. Such communication and computation must be performed (or at least simulated) every time such an action might be necessary to avoid leaking whether a friend relationship is being changed. However, the urgency of completing such changes is lower than delivering a message, and so we set a frequency ϕ at which friend-update messages or operations may be performed (e.g., $\phi = 10$ indicates such operations will happen once every 10 epochs).

The process for u to establish a new friend relationship with v is:

1. u reserves one of their friend slots for v at the private-channel storage server S_u^{priv}, recording in that slot the current epoch and assigning it a nonce ζ. A slot reservation may be made in a free slot or one reserved more than e epochs prior. S_u^{priv} creates a friend request F, which includes the current epoch, the source u, the receiver v, the nonce ζ, and a MAC of those contents under key K.

2. u submits a friend request to v via the public channel as the contents of a normal message to v.

3. If the public-channel storage server S_v^{pub} receives a friend request F with a valid MAC and with the current epoch, it delivers the request to v's mailbox.

4. After v's client receives F, the user can manually approve it (if it has a friend slot free) or reject. If accepted and the epoch in the request is within e of the current one, v's client sends the response G to u via the private channel. G contains the request F and a response flag, and is sent in a normal private-channel message.

5. When private-channel ingestion server I_v^{priv} receives a friend response G, it validates the MAC of the contained request F and checks that the request was sent less than e epochs in the past. If those checks pass, then it forwards G as with a normal message with a valid token. At the same time, it directly contacts v's private storage server S_v^{priv} to inform it that a friend relationship is being confirmed with u, and S_v^{priv} adds u as a friend of v.

6. When G is received by the storage server S_u^{priv}, the server detects that it is such a response, validates the MAC key, and checks for the expiration of the contained request F. If either check fails, the response is ignored. Otherwise, S_u^{priv} identifies the reserved

friend slot with a matching nonce. If the response is to accept, it records the friendship as confirmed. Otherwise, and the response is to reject, it removes the slot reservation.

If user u wants to revoke an existing friendship with user v:

1. u creates a revocation request F and sends it on the private channel included in a normal message. Note that because u and v are friends, u is able to obtain the token needed for a private-channel message to v.

2. When the next epoch starts after receiving F, the ingestion server I_u^{priv} directly contacts the storage server S_u^{priv} to instruct it to remove v from the friend list of u.

3. When F arrives at the storage server S_v^{priv}, the server detects the revocation request, removes u from the friend list of v, and delivers F to the mailbox of v as with a normal message.

4. When the client of v receives F, it notifies the user that the friendship with u has been revoked.

As noted, the maintenance of friendships requires some computation and communication beyond what is required for normal messaging, and the system must perform these obliviously to hide not just which users have friendships but also if a user is updating their friendships at all. The system performs these additional actions only once every ϕ epochs, due to their lower urgency and frequency. We further limit each user to performing one friend update during such an epoch. We summarize the added actions that clients and servers must perform:

1. When contacting their private storage server, a user sends a (possibly dummy) command to reserve a friend slot, the server obliviously uses it to update the friend list and responds with a (possibly dummy) friend request.

2. A private-channel ingestion server I_u^{priv} obliviously scans incoming messages for a friend response or revocation, validates the one identified (or a dummy), and contacts the corresponding storage server S_u^{priv} for u with a friend update (a response or revocation), which S_u^{priv} obliviously uses to update the friend list of u.

3. A private-channel storage server S_u^{priv} obliviously scans received messages for a friend response or a revocation, validates the (possibly dummy) update, and obliviously updates the friend list.

Table 7.3: Computation per epoch for a client or server

Entity	Runtime
C	$O(\ell)$
I	$O(\ell c_I)$
R (offline)	$O((n/r + rs)\log^3(n/r + rs))$
R (online)	$O(\ell(n/r + rs)\log(n/r + rs))$
S (offline)	$O((n/s)\log^3(n/s))$
S (online)	$O(\ell(n/s)\log(n/s))$

7.3 Analytical Evaluation

7.3.1 Efficiency

We focus only on costs during messaging, as the costs of the maintenance procedures for accounts and friends are much lower. Let C be a client, and let c_x be the number of clients that have interacted with server(s) x in a given epoch. Let μ be the length of a MAC tag used in long user IDs, and let τ be the length of a token. We analyze runtimes asymptotically in terms of the message length and the numbers of users and servers. We assume the runtimes of the oblivious subroutines used in our implementation. For y items of length z, the runtime for oblivious compaction is $O(zy \log y)$ (Section 4.3.2), and for both oblivious shuffle and oblivious sort it is $O(zy \log y)$ online and $O(y \log^3 y)$ offline (Section 6.4.3).

The runtimes of individual clients and servers per epoch are given in Table 7.3. There is no asymptotic difference in the runtimes between the private and public channel, and so we list them generically for both channels. For clients, the runtime is linear in the message size because for each channel they send and receive a fixed-size batch of messages. For ingestion servers, we see that the runtime is linear in the number of client interactions and the message length. For routing servers, the costs are dominated by the message processing in round two, where there are $O(n/(sr) + r)$ messages for each of the s storage servers, and the most costly operation on them is the oblivious shuffle. For storage servers, there are $O(n/s)$ messages to process, and that processing is dominated by an oblivious sort of them. Note that the offline operations are not on the messages themselves (and are instead on message indices), and so the ℓ factor only appears in the online runtimes.

The total communication of each type of server per epoch is given in Table 7.4. In general, the private and public channels differ by the inclusion of MAC tags between

Table 7.4: Communication per epoch for a client or server

Entity	Bytes received	Bytes sent
C	$\ell(f+b)$	$2\ell+\mu+\tau$
I^{priv}	$c_{I^{\text{priv}}}(\ell+\tau)$	$c_{I^{\text{priv}}}\ell$
I^{pub}	$c_{I^{\text{pub}}}(\ell+\mu)$	$c_{I^{\text{pub}}}\ell$
R^{priv}	$(c_{\mathcal{I}_{R^{\text{priv}}}}\ell)+$ $\ell s(\lfloor f\lceil n/s\rceil/r\rfloor+r)$	$\ell s(\lfloor f\lceil n/s\rceil/r\rfloor+r)+$ $(s/r)f\lceil n/s\rceil+$
R^{pub}	$(c_{\mathcal{I}_{R^{\text{pub}}}}\ell)+$ $\ell s(\lfloor b\lceil n/s\rceil/r\rfloor+r)$	$\ell s(\lfloor b\lceil n/s\rceil/r\rfloor+r)+$ $\ell(s/r)b\lceil n/s\rceil$
S^{priv}	$\ell f\lceil n/s\rceil$	$c_{S^{\text{priv}}}f\ell$
S^{pub}	$\ell b\lceil n/s\rceil$	$c_{S^{\text{pub}}}b\ell$

the clients and ingestion servers and by the size of a per-client batch of messages (f and b, respectively). Clients send and receive fixed-size batches of messages. Ingestion servers send and receive a single message for each of their client interactions, and so their load increases just with the number of online clients rather than the number of assigned clients. Routing server communication is dominated by the second round, in which each one sends and receives $O(n/r+sr)$ messages, where the second term is due to the r messages added for each storage server to ensure enough space to send messages to the appropriate routing server. Storage servers send and receive $O(n/s)$ messages because mailboxes are evenly distributed across storage servers.

To consider how the system as a whole scales with the number of users, assume for simplicity that $s=r$ and that n/s is integral. Then in the private channel the total communication and computation per routing server is proportional (up to logarithmic factors) to the total number of messages that server sent or received, which is $3fn/r+r^2$. To minimize this value, we set r as a function of n: $r=(3fn/2)^{1/3}$. At this value, the number of users per routing server becomes $(2n^2/(3f))^{1/3}=O(n^{2/3})$, and similarly for the public channel with b in the place of f.

This analysis indicates that the load per router grows with the number of users. Ingestion and storage servers are easy to keep at constant load, as an arbitrary number of them can connect to a single routing server. Although the increasing load on routing servers does limit system scalability, our experimental results (Section 7.4) indicate that the system can scale to at least tens of millions of users. Those results, made in a 13 Gbps network, show that we can obtain message latencies of less than 3 seconds when each (single-core) server has up to 2^{18} users. Following the previous analysis, we can stay at or below this load per routing server with as many as $n=164$ M users, which requires

$r = 628$. At this size, TEEMS could host the estimated 40–100 million users of the Signal messaging app [HSQ+23], which is popular among the privacy-conscious.

Scaling TEEMS to even larger sizes may be possible by changing its routing algorithm. The routing servers essentially sort the messages by the storage server of the recipient, and networks of constant node degree exist for such distributed sorting, including butterfly-routing and sorting networks [ACN+20, Bat68]. However, such networks require at least $\log_2 n$ rounds compared to the two rounds of ColumnRoute, which for, say 2^{28} users would increase the round latency and the total number of bytes sent by 14 times.

7.3.2 Security

We informally present and argue for the security claims for TEEMS.

We consider a game in which the adversary chooses the client interaction schedules, selects which users send messages and to whom, and designates a challenge message (including its sender and receiver). The adversary will be able to observe and modify network communication, observe and execute its clients, and observe server execution and memory traces. The challenger randomly chooses a challenge bit and then executes the honest clients and server, varying if the challenge message is submitted or not based on the challenge bit. The adversary must then guess if the challenge message was submitted by its user. This game is an instance of the Communication Unobservability game as defined by Kuhn et al. [KBS+19]. If a probabilistic polynomial-time adversary is unable to guess the challenge bit wih probability differing non-negligibly from a random guess, then the system is said to be Communication Unobservable (CU), which TEEMS satifies according to Claim 1.

Claim 1. In TEEMS, the private channel is CU, and the public channel is CU if no storage server receives more than bn/s messages.

The argument for Claim 1 is that the two possible distributions of observations made by the adversary, depending on the challenge bit, are computationally indistinguishable. Network messages are all sent on secure channels. The number of bytes exchanged with clients depends only on public parameters and the client interaction schedules. The number of bytes sent by ingestion servers and routers similarly depends only on those things, as well as which messages the adversary allows to be delivered by clients. The number of bytes from storage servers depends only on the client interaction schedules.

Adversarial clients can only access their own mailboxes because storage servers authenticate users, and so they cannot see if other users receive messages. For messages to adversarial users, in the private channel, they are only dropped predictably in response to adversarial network interference. However, in the public channel, the CUS guarantee requires that no storage server receives more than bn/s messages because otherwise adversarial clients may observe their messages being dropped due to the presence of the challenge message. If the fraction of adversarial users is a and if, in a given epoch, the fraction of users assigned to S^{pub} that receive on the public channel from honest users is $h_{S^{\text{pub}}}$, then the condition for security is satisfied if $a + h_{S^{\text{pub}}} b/s < b/s$, for all $S^{\text{pub}} \in \mathcal{S}^{\text{pub}}$. We expect this condition to hold most if not all epochs in the system at scale; for example, at our suggested maximum size of TEEMS of $n = 164M$, if even 10% of users at a storage server receive in a given epoch, the adversary would need to control and coordinate over 235K users to exploit this potential leakage. Moreover, the information leak is at most the number of messages destined for a given storage server.

The memory and instruction traces at the servers are also the same regardless of the challenge bit because the server algorithms are oblivious to the contents of messages, including to whether those messages are dummies or real. We note that at the storage servers, the operations on the short- and long-term mailboxes depend only the client interactions, which are known to the adversary.

Claim 2. TEEMS provides message integrity, including its content, sender, and timestamp.

To show Claim 2, we observe the ingestion servers authenticate users and ensure that the authenticated identity matches the senders in the submitted messages. TEE correctness and the use of secure channels between enclaves ensures that the sender and content stays correct until it is delivered to the user. Messages are delivered in the epoch after they are received (if they are delivered at all), and so the TEEs at the storage servers ensure that the correct timestamp is attached before delivery to the client.

7.4 Experimental Evaluation

We implemented both the private and public channel protocols of TEEMS, and present benchmarks for TEEMS in this section. Our empirical evaluations evidence TEEMS' ability to support low-latency and high-throughput metadata-protected communications, while being able to scale horizontally as more users join the system. The two key questions we address with our experiments are:

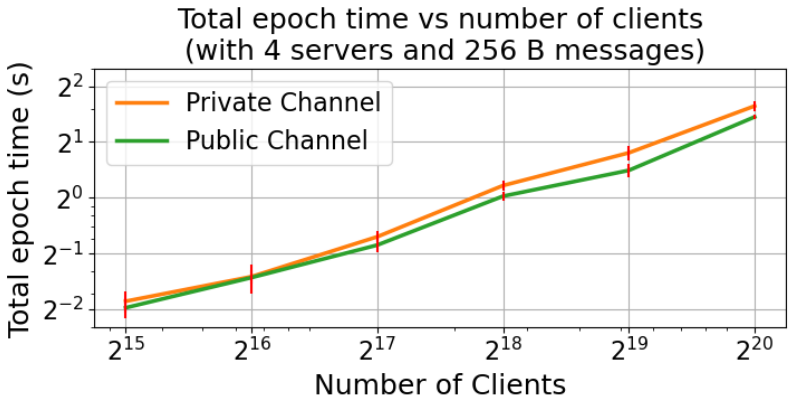


Figure 7.7: Measuring private and public channel epoch times as the number of clients increases with a fixed set of 4 single-core servers. Both axes are log scale.

What is the messaging latency for TEEMS? With 2^{20} clients, TEEMS can deliver E2E metadata-protected messages in under 3.1 s and 2.7 s for private and public channels respectively, with as few as 4 servers (implemented as 4 cores on a single machine). In contrast, Boomerang with 8 servers (8 cores each) takes 15.5 s without support for asynchrony.

How well does TEEMS scale horizontally? With 2^{20} clients, by simply adding more servers into TEEMS, its performance can improve from 8.3 s and 7.1 s with just one server for private and public channel respectively, all the way down to 0.4 s for each channel with 72 servers (implemented as 72 cores on a single machine). Boomerang with 32 servers (8 cores each, for a total of 256 cores) takes 7.8 s in comparison.

Experiment Setup All our experiments are performed using an Intel SGX machine with two Xeon 8380 CPUs with 40 cores each, and 2.30 GHz processor speed. We simulate multiple servers by instantiating each logical server (that runs all three server processes of ingestion, routing, and storage) on our machine with a single core assigned to it; we set aside 8 cores for simulating clients on the same machine. Consequently, the largest of our experiments present instantiations of TEEMS with up to 72 servers. We envision a real-world instantiation of TEEMS would have all servers in a data center and assume a 13 Gbps of network bandwidth between servers (like prior work [JWC+23, DFD+21]). Hence all our reported timing measurements include latency overheads based on this expected network bandwidth, even though all our servers are situated on the same machine. We also note that this running of multiple servers on the same machine

is not just an artifact of our experiment; a real deployment of TEEMS could also run multiple TEEMS servers on a single many-core machine. Since all of our benchmarks are simulated on the same server, we incur paging overheads as we run larger experiments. We emphasize that this implies our results are conservative and in practice TEEMS would yield better performance when instantiated as a distributed system.

In our implementation, we use our ORCompact (Section 4.3) for oblivious compaction. For reverse compaction, we design and implement ORExpand, which runs the ORCompact algorithm in the reverse direction. For efficiency, TEEMS leverages our WaksShuffle and WaksShuffle+QS algorithms (Section 6.3) for oblivious shuffling and sorting. Our timing results presented only incur the online phase of these oblivious shuffles and sorts. In practice, as long as the server has a few additional cores available, they can be used in parallel with each round of public/private route to perform the precomputation required for the next round. For instance, with 2^{20} clients over 32 single-core servers, each server would need $\approx$ 1.8 and $\approx$ 2.5 additional cores for private and public channels respectively, for a total of 64 servers and 202 cores (for both channels in total) to complete all the precomputation before the start of the next epoch. Since we have a limited number of cores in our experimentation setup, we perform these precomputations up front before an epoch to measure TEEMS' scalability.

In our experiments, in both the private and public channel case, we set the message size to 256 bytes, and users send and receive exactly one message in every communication round for both private (f=1) and public channel (b=1) cases. These parameters can be tuned to allow for a clients to receive more messages at the expense of more server computation. In our experiments, all TEEMS servers perform all three roles of ingestion, routing, and storage. Allocating servers designated roles also presents another axis that can be tuned to improve performance and scale horizontally.

7.4.1 Latency and throughput with fixed servers

In Figure 7.7, we observe the latency overheads of TEEMS with 4 single-core servers, as the number of clients in the system increases. Even in this small-scale experiment, TEEMS can enable metadata-protected messaging for 2^{16} clients in under 0.4 s, and a throughput of 164K messages per second. As the number of users in the system increases, message latency grows linearly with a constant < 1, in the client range we evaluate; i.e., doubling the number of clients with the same number of servers results in slightly less than $2\times$ the round latency. At 2^{20} clients with 4 servers, TEEMS can perform one round of private or public channel communications in under 3.1 s, exhibiting a throughput of $\approx$338K messages per second.

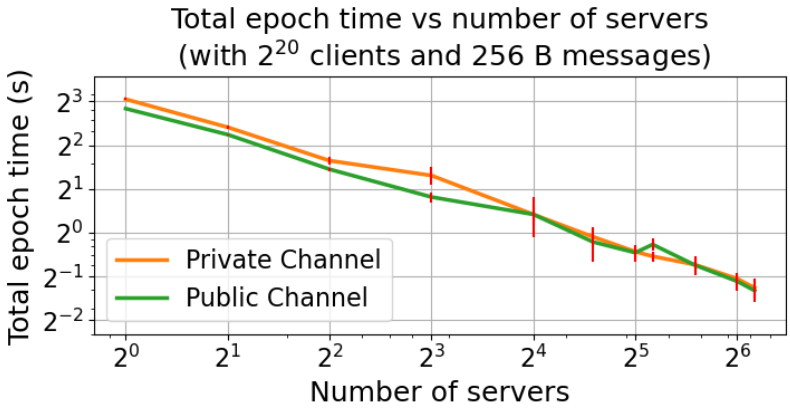


Figure 7.8: Measuring private and public channel epoch times for TEEMS as the number of single-core servers increases with a fixed set of 2^{20} clients. Both axes are log scale.

7.4.2 Horizontal Scalability

Next, we analyze how well TEEMS scales horizontally. In Figure 7.8, we observe the total time taken to route private and public channel messages of 2^{20} clients, as a function of the number of servers. We observe that with fixed number of clients, doubling the number of servers consistently results in ≈30–40% reductions in the round latency. Routing messages from 2^{20} clients gradually drops down from 8.3 s and 7.1 s (throughput of ≈126K and ≈145K messages per second) for private and public channels respectively with just one (single-core) server, all the way down to 0.4 s (throughput of ≈2.6M messages per second) with 72 (single-core) servers—clearly exhibiting TEEMS' ability to scale horizontally to maintain low latency and high throughput as the user base grows. While our results demonstrate the ability of TEEMS to deliver messages in under 0.5 s, in practice a latency in the order of a few seconds, about the time it takes to actually type a message, would suffice.

7.5 Related Work

Over the last two decades several novel MPCS designs have been proposed as we saw in Chapter 3. However, a substantial fraction of these constructions target Sender Unlinkable Broadcast Systems [CGBM15, KLDF16, KCGDF17, APY20, NSSD22, LSSD22, LK22]. Such systems enable clients to publish a message to all other users of the system with-

Table 7.5: Comparing MPCS for metadata-protected messaging. 'O' implies the system has a receiver anonymity set of currently online users. 'A' indicates that all users (irrespective of whether they are online or offline) are part of the receiver anonymity set. The ◐ for Loopix's asynchronous property is due to its trusted service provider assumption (detailed in Section 3.3.2). Clarion, XRD, Karaoke, and Boomerang require setup in the form of a dialing protocol. The experiment numbers below do not account for dialing. In Clarion, dialing incurs about the same overhead as its communication protocol. The dialing protocol used by Boomerang and Karaoke [LZ16] takes ≈16 s for 1 million users. XRD requires dialing, but no existing dialing protocol fits the system. Dialing has to be run every few communication rounds before any actual message data can be transmitted.

System	Low Latency	High Throughput	Horizontally Scalable	Async	Receiver Anonymity Set	Experiment Reported				
						Clients	Servers (Total cores)	Msg size (bytes)	Latency	Throughput (msgs/sec)
Clarion [EB22]	○	○	○	○	O	10^6	3 (48)	160	80 s	12.5 K
XRD [KLD20]	○	○	○	○	O	10^6	200 (7200)	256	128 s	8 K
Karaoke [LGZ18]	○	●	●	○	O	10^6	100 (3600)	256	6 s	166 K
Loopix [PHE+17]	●	○	●	◐	A	3×10^4	5 (80)	224	1.9 s	16 K
SealPIR [ACLS18]	●	○	○	●	A	2.6×10^5	1 (16)	288	0.51 s	2
Sabre [VSH22]	●	○	○	●	A	2^{18}	3 (48)	1000	0.05 s	20
Boomerang [JWC+23]	○	●	●	○	O	2^{20}	32 (256)	256	7.8 s	134 K
TEEMS	●	●	●	●	A	2^{20}	64 (202)	256	0.74 s	2.87 M

out leaking any metadata that links the sender to their message, and is orthogonal to our work. In contrast, TEEMS aims to achieve a Communication Unobservable System (CUS) that can facilitate end-to-end metadata-protected messaging. Table 7.5 presents a concise view of the start-of-the-art MPCS that be used towards our goal.

Clarion [EB22] is the latest iteration of an SMC-based CUS [AKTZ17], but it incurs high latency and lacks scalability in comparison to mixnet-based designs. Mixnet-based designs have produced many high-throughput constructions, and some of them have low latency and are horizontally scalable as well, but they all lack asynchronicity. XRD [KLD20] presents a mixnet-based CUS that is more efficient than prior pure mixnet based approaches [KLDF16, CDJ+17]. Mixnet-based designs drastically improved latency and throughput by incorporating differential privacy to hide the metadata of adversary observable parameters of the system. This line of works started with Vuvuzela [VDHLZZ15], and was followed by Stadium [TGL+17] and Karaoke [LGZ18] which are horizontally scalable variants of Vuvuzela. In Table 7.5, we present Karaoke as a representative for this line of works.

The improvements in throughput and latency, however, are at the expense of shifting the underlying privacy guarantee to a differentially private one rather than a crypto-

graphic one. Boomerang [JWC+23] inspired by Karaoke, presents an MPCS that leverages TEEs to avoid using differential privacy. However, just like all the other aforementioned mixnet designs they cannot support asynchronous clients, and they all have to incur the additional overheads of a dialing protocol before any actual communication. Loopix [PHE+17] represents an outlier among other mixnet-based MPCS designs as it can support asynchronous clients. However, asynchrony is achieved at the expense of trusting special 'service provider' nodes in the system, and adversarial providers can subvert clients' metadata privacy. Moreover, Loopix has the weakest form of the manytrust assumption due its underlying stratified network structure as we detailed in Section 3.4.3.

In the literature, designs that can support asynchronous clients have stemmed from PIR-based techniques [CSM+20, AS16, ACLS18, VSH22, SCM05]. Pung [AS16] and its successor SealPIR [ACLS18] present a CUS through single-server computational PIR, and consequently incur significant computational overheads. Express [ECGZB21] presents a Reverse PIR based mailbox system for end-to-end metadata protected communications, and its successor Sabre [VSH22] further improves the performance of this style of mailbox systems. In Table 7.5, we present Sabre and SealPIR as representatives for these two different PIR-based approaches. Ultimately, such PIR based schemes can offer low latency for message delivery and retrieval, but they have to handle each message individually and consequently have poor throughput. Moreover, both latency and throughput exacerbates as more users join the system, as they cannot efficiently scale horizontally.

In comparison to the existing MPCS literature, TEEMS presents the first MPCS construction to simultaneously achieve all the four crucial properties of (i) low latency, (ii) high throughput, (iii) horizontal scalability, and (iv) asynchronicity, desirable of any messaging system. In addition to supporting improved functionality over prior approaches, as evidenced by Table 7.5, even with our small-scale experiments, TEEMS presents more than an order of magnitude improvement in latency and throughput over prior work.

7.6 Conclusion

In this chapter, we present TEEMS, the first MPCS that targets metadata-protected messaging and supports the four fundamental properties of (i) low latency, (ii) high throughput, (iii) horizontal scalability, and (iv) asynchronicity required for any messaging service to be deployed in practice. In comparison to prior work, TEEMS presents more than an order of magnitude improvement in terms of latency and throughput, requires fewer servers for scaling, and affords more desirable properties.

Chapter 8

Conclusion

Myriad MPCS designs have been proposed over the last four decades with varying privacy and functionality goals, and spanning several systems architecture choices and security assumptions. In Chapter 3 we presented a detailed systematization of prior constructions, and highlighted the dearth of MPCS constructions that can attain four fundamental properties desirable of any communication system. Motivated by our findings from the systematization, the thesis statement in Section 1.1 sets out our claim that we can design an end-to-end metadata protected communication system that can simultaneously achieve low latency, high throughput, horizontal scalability, and asynchronous client support. Our MPCS design TEEMS presents the first MPCS design that achieves this claim.

Our construction leverages advancements in TEEs towards privacy-preserving computations as a starting point. However, existing research demonstrates that TEEs are vulnerable to numerous side-channel attacks, and historically this has led to ad-hoc defenses, and an arms race of attacks and defenses, without addressing the root problem. To this end, we first establish the notion of fully oblivious algorithms that are safe to run within such TEEs in Section 4.1. Fully oblivious algorithms address the concerns raised by software side channels at their inception point, since by definition such algorithms do not induce any secret-dependent software side channels.

Efficient and secure building blocks lay the foundations for efficient and secure systems. From the MPCS literature, promising high-throughput designs often leverage shuffling and sorting algorithms towards facilitating metadata-protected communications. Hence, it is no surprise that we would require efficient counterparts for shuffling and sorting in the TEE realm with the added constraint of full obliviousness towards designing an oblivious routing protocol that is safe to run within TEEs. Since existing

well-known approaches for fully oblivious algorithms for fundamental data operations like shuffling and sorting induce significant overheads, we first explore and design novel fully oblivious algorithms for these core building blocks.

In Chapter 4, we presented ORCompact which provides concrete improvements over the state-of-the-art compaction protocols, and furthermore it is efficiently parallelizable. Chapter 5 presented ORShuffle and BORPStream, which are fast fully oblivious shuffling algorithms for the standard and streaming models respectively, and both these algorithms demonstrate improvements in concrete performance over the state-of-the-art bitonic shuffle. Finally, in Chapter 6 we presented fully oblivious offline/online shuffling and sorting algorithms that provide asymptotic and concrete improvements in the online time to shuffle and sort data, and even in the total time when item sizes are at least moderately large. We leverage these building blocks in our MPCS design TEEMS, but these primitives by themselves have several other applications for privacy-preserving computations broadly as we point out in their respective chapters.

Finally in Chapter 7, this dissertation introduces TEEMS, a new direction and construction for facilitating metadata-protecting communication systems. TEEMS introduces a notion of private and public channels for metadata-protected communications, where private channels provide guaranteed message delivery channels for correspondents that have a symmetric friend relationship already established. Public channels can be used to bootstrap private channels by sending a friend request, and can even be used for users to send cold messages to other users (provided they have their long user id), albeit without guaranteed delivery. Internally, TEEMS operates in synchronous rounds while allowing for asynchronous clients.

At the heart of TEEMS is an oblivious pad and route protocol ColumnRoute, which ensures that at the end of a round all TEEMS clients receive exactly the same (tunable system parameter) number of messages without inducing any side-channel leakages. ColumnRoute draws upon the technique of column sort for distributed sorting, but takes advantage of the observation that the routing problem is slightly easier than sorting in the TEE setting, as the routing servers can observe the final destination of messages. TEEMS hence introduces ColumnRoute, an oblivious route and pad protocol, which results in reducing the number of rounds required from four to three (or two if we elide the simple forwarding step from routing to storage servers). All clients in TEEMS always receive the exact same number of messages each round for both public and private channels, and messages from each round for a client can be accumulated by the storage server in that client's mailbox.

Our oblivious route and pad approach presents a new direction towards addressing the problem of asynchronous message storage that we discussed in Section 3.4.4. While

prior RPIR based approaches enable asynchronous message storage, they do so at the expense of performing linear computation in the number of clients for each message in the system to hide which mailbox the message arives at. Unfortuntaely, this quickly impedes latency, throughput, and scalability of such systems. Our approach can be viewed as a way to amortize this expense for a batch of messages, by ensuring that all clients receive the same number of messages in a round via the oblivious route and pad protocol instead. In Table 8.1, we present our MPCS comparison table from the systematization in Chapter 3, but with the addition of an entry for TEEMS (highlighted with a shaded background).

While TEEMS presents a step forward in MPCS designs targeting metadata-protected messaging systems, much work remains to be done towards a real-world deployment of such systems. Several avenues of future work still remain for making MPCS practically viable. For instance, (i) TEEMS as presented here does not consider how to handle robustness of the system under server churn conditions. Moreover, (ii) the current TEEMS design is tailored for end-to-end metadata-protected conversations, but expanding such systems to support efficient metadata-protected group chats also present a promising avenue for future research. Finally, (iii) researching techniques for incremental rollout of metadata-protected messaging systems over existing popular messaging systems also remains unexplored.

System	Year	Type	Family	Assumption	Robust	AS Protection	RE Resistant	Disruption Resistant	Disconnection Impact	Setup	Parallel Conversations	Asynchronous	Low Latency	High Throughput	Horizontally Scalable	Audit Complexity	Client Overhead	Message [Compute if Different] Complexity	Messages
					Protections					Usability						Performance			
Vuvuzela [VDHLZZ15]	2015	CUS (Mix)	DP	$(1:m)$-S: DP	○	○	●	●	▼	📞	○	○	○	○	○	×	m	$n \cdot m + n \cdot m$	n
Pung [AS16]	2016	CUS (RUMS)	PIR	RLWE	○	○	●	●	▼	📞	●	○	●	○	k^2	×	n	$n^2 + n$	1
Stadium [TGL+17]	2017	CUS (Mix)	DP	f-S: DP	○	○	●	●	▼	📞	○	○	○	●	k	×	ℓ	$(n+m_d) \cdot m_d + n \cdot \ell + m^3$	n
MCMix [AKTZ17]	2017	CUS	SMC	$(1:3)$-S	○	○	●	●	▼	📞	○	○	●	○	○	×	1	$n \cdot \log n + n \cdot \log n$	n
AnonPOP [GHL18]	2018	CUS (Mix)	M	f-S: DP	○	○	●	●	≈	🤝	●	○	○	○	k	×	ℓ	$n \cdot \ell$	n
SealPIR [ACLS18]	2018	CUS (RUMS)	PIR	RLWE	○	○	●	●	▼	📞	●	○	●	○	k^2	×	n	$n^2 + 1$ [$n+n$]	1
Karaoke [LGZ18]	2018	CUS (Mix)	DP	f-S: DP	○	○	●	●	▼	📞	○	○	●	●	k	×	ℓ	$(n+m_d) \cdot m_d + n \cdot \ell + m^3$	n
XRD [KLD20]	2020	CUS	M	f-S	○	○	●	●	▼	📞	◐	○	○	○	k^2	×	$\sqrt{m}$	$n^2 + n \cdot \sqrt{m}$	1
Clarion [EB22]	2022	CUS	SMC	$(2:3)$-MA	○	○	●	●	♉	📞	○	○	○	○	○	×	1	$n \cdot \log n + n$	n
Groove [BKL+22]	2022	CUS (Mix)	DP	f-S: DP	○	○	●	●	≈	📞	●	○	●	●	k	×	ℓ	$(n+m_d) \cdot m_d + n \cdot \ell + m^3$	n
Boomerang [JWC+23]	2023	CUS	M	TEE	○	○	●	●	▼	📞	○	○	●	●	k	×	1	$(n+m_d) \cdot m_d + n$ [$n \cdot \log^2(n/m)$]	n
TEEMS	2024	CUS	RPIR	TEE	○	○	●	●	▼	🤝	●	●	●	●	k	×	1	n [$n \cdot \log^3(n/m) + n \cdot \log(n/m)$]	n
DC-nets [Cha88]	1988	CUS/SUBS	DC	None	○	●	●	○	🚫	🤝	○	○	○	○	○	×	n	$n^2[n]$	1
Dissent [CGF10]	2010	CUS/SUBS	DC	None	○	○	●	○	🚫	🤝	○	○	○	○	○	n^2	n	$n^2 + n^2$ [$n^2 + n$]	1
D3 [WCGFJ12a]	2012	CUS/SUBS	DC	$(1:m)$-S	○	○	●	◐	▼	🤝	○	○	○	○	○	$n^2 \cdot m$	m	$n \cdot m + n \cdot m$	1
Dissentv2 [WCGFJ12b]	2012	CUS/SUBS	DC	$(1:m)$-S	○	○	●	◐	▼	🤝	○	○	○	○	○	$n^2 \cdot m$	m	$n \cdot m + n \cdot m$	1
Verdict [CGWF13]	2013	CUS/SUBS	DC	$(1:m)$-S	○	○	●	●	▼	🤝	○	○	○	○	○	$n^2 \cdot m$	m	$n \cdot m + n \cdot m$	1
Riposte [CGBM15]	2015	SUBS	RPIR	$(2:3)$-S	○	○	○	●	▼	×	🚫	●	●	○	k^2	n	$\sqrt{n}$	$\sqrt{n}$ [n]	1
Riffle [KLDF16]	2016	SUBS	M	$(1:m)$-S	○	○	●	●	▼	×	🚫	●	○	○	○	×	m	$n \cdot m$	n
cMix [CDJ+17]	2017	SUBS	M	$(1:m)$-S	○	○	●	●	▼	×	🚫	○	○	○	○	×	m	$n \cdot m$	n
Atom [KCGDF17]	2017	SUBS	M	f-S	●	◐	●	●	▼	×	🚫	●	○	○	k	×	1	$n \cdot m$	n
AsynchroMix [LYK+19]	2019	SUBS	SMC	$(2n/3:n)$-M	◐	●	●	●	▼	×	🚫	●	○	○	○	×	m	$m \cdot n^2 \cdot \log^2 n + m \cdot n \cdot \log^2 n$	n
PowerMix [LYK+19]	2019	SUBS	SMC	$(2n/3:n)$-M	◐	●	●	●	▼	×	🚫	●	○	○	○	×	m	$n^2 + m \cdot n$ [$n^2 + n^3 + m$]	n
Blinder [APY20]	2020	SUBS	SMC	$(3n/4:n)$-M	●	●	●	●	▼	×	🚫	●	○	○	○	$m \cdot \sqrt{n}$	$m \cdot \sqrt{n} m \cdot n \cdot \log n + m \cdot n$ [$m \cdot n \cdot \log n + n^2$]	n
Spectrum [NSSD22]	2022	SUBS	RPIR	$(1:m)$-S	○	●	●	●	▼	×	🚫	●	○	○	○	1	m	m [$m \cdot b$]	1
Trellis [LSSD22]	2022	SUBS	M	f-M	●	●	●	●	🚫	×	🚫	○	○	○	k	×	$\log m$	$n \cdot \log^2 m + n \cdot \log m$ [$n \cdot m$]	n
RPM [LK22]	2022	SUBS	SMC	$(3n/4:n)$-M	●	●	●	●	▼	×	🚫	●	○	○	○	×	m	$m \cdot n^{1.5} + n$ [$m \cdot n^2 + n^{1.5}$]	n
Express [ECGZB21]	2021	SUMS	RPIR	$(1:2)$-S	○	○	○	●	▼	🤝	●	●	●	○	k^2	n	λ	1 [n]	1
Sabre [VSH22]	2022	SUMS/SUBS	RPIR	$(1:2)$-S	○	○	●	●	▼	🤝	●	●	●	○	k^2	$\log n$	$\log n$	1 [n]	1
Mixnets [Cha81]	1981	RUS (Mix)	M	$(1:m)$-S	○	○	●	●	🚫	×	○	○	○	○	○	×	m	$n \cdot m$	n
Pynchon Gate [SCM05]	2005	RUS (RUMS)	PIR	$(1:m)$-S: TTP	○	○	●	●	▼	×	●	●	○	○	○	×	$n \cdot m$	$n \cdot m$	1
Loopix [PHE+17]	2017	RUS (Mix)	M	f-S: TTP	○	○	●	●	▼	🤝	●	○	○	●	k	×	1	ℓ	1
Talek [CSM+20]	2020	RUS (RUMS)	PIR	$(1:m)$-S	○	○	●	●	▼	🤝	●	○	○	○	○	×	m	$n \cdot m$	1

Table 8.1: (I) *Notation:* (i) *Family*: DC = DC-nets, M = Mixnets, RPIR = Reverse-PIR, DP = Differential Privacy, SMC = Secure Multiparty Computation. (ii) *Variables*: n=#clients, m=#servers, m_d=#servers in dialing protocol, f=fraction of malicious servers, ℓ=#servers on an anytrust chain (ℓ=$g(f,m)$ where g depends on the system design to ensure that at least one server in every anytrust chain is honest), b=#broadcasters. (iii) *Assumptions*: TTP: Trusted Third Party. RLWE: Ring Learning With Errors. (x,y)-T: At least x of y servers should be honest, with servers of type T: S - Semihonest, M - Malicious, MA - Malicious but Available. f-T: At most fraction f of m servers can be malicious, with servers of type T. (iv) *Disconnection Impact*: 🚫: system fails, ♉: correspondent leaked, ▼: anonymity set reduction, ≈: temporary anonymity set preservation. (v) *Setup*: 🤝: clients exchange information out of band, 📞: 🤝 + dialing protocol. (vi) *Performance*: Dialing or shuffling (for slot scheduling) costs in blue. Preprocessing/offline costs in gray.

(II) *Meanings of* ◐*:* (i) Robust: non-robust offline phase. (ii) Disruption resistant: disruption resistant scheduling, but not conversations. (iii) Asynchronous: The RUS underlying Pung and SealPIR is asynchronous, but as a CUS it is not asynchronous. Groove allows users to disconnect for a small constant number of rounds. Loopix's asynchrony has a privacy expense (Section 3.3.2). Talek asynchronous messages can be lost by its design (Section 3.4.4).

(III) The blue-dashed box highlights the four crucial properties required for a metadata-protecting messaging systems.

References

[ACL14] ACLU of Northern California. Metadata: Piecing Together a Privacy Solution, 2014. https://www.aclunc.org/sites/default/files/Metadata%20report%20FINAL%202%2021%2014%20cover%20%2B%20inside%20for%20web%20%283%29.pdf. Accessed August 2024.

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with Compressed Queries and Amortized Query Processing. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[ACN+20] Gilad Asharov, TH Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Bucket oblivious sort: An extremely simple oblivious sort. In *Symposium on Simplicity in Algorithms (SOSA)*, 2020.

[AGJS13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. https://software.intel.com/content/www/us/en/develop/articles/innovative-technology-for-cpu-based-attestation-and-sealing.html, 2013. Accessed August 2024.

[AKL+18] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal Oblivious RAM. Cryptology ePrint Archive, Report 2018/892, 2018.

[AKL+20a] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, Elaine Shi, and Yuval Ishai. OptORAMa: Optimal Oblivious RAM. In *Advances in Cryptology (EUROCRYPT)*, 2020.

[AKL+20b] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Oblivious Parallel Tight Compaction. In *Conference on Information-Theoretic Cryptography (ITC)*, 2020.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ Sorting Network. In *ACM Symposium on Theory of Computing (STOC)*, 1983.

[AKTZ17] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous Messaging via Secure Multiparty Computation. In *USENIX Security Symposium*, 2017.

[ALS22] Gilad Asharov, Wei-Kai Lin, and Elaine Shi. Sorting short keys in circuits of size $o(n \log n)$. *SIAM Journal on Computing*, 2022.

[ALT18] Sebastian Angel, David Lazar, and Ioanna Tzialla. What's a Little Leakage Between Friends? In *Workshop on Privacy in the Electronic Society (WPES)*, 2018.

[AMBFK16] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private Information Retrieval for Everyone. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016.

[APY20] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder – Scalable, Robust Anonymous Committed Broadcast. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[AS16] Sebastian Angel and Srinath Setty. Unobservable Communication Over Fully Untrusted Infrastructure. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[ASC21] A K M Mubashwir Alam, Sagar Sharma, and Keke Chen. SGX-MR: Regulating Dataflows for Protecting Access Patterns of Data- Intensive SGX Applications. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2021.

[BA16] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. *International Journal of Information Security*, 2016.

[Bat68] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of American Federation of Information Processing Societies (AFIPS)*, 1968.

[BBC+19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *Advances in Cryptology (CRYPTO)*, 2019.

[BBC21] BBC News. Putin critic Navalny jailed in Russia despite protests, 2021. https://www.bbc.com/news/world-europe-55910974. Accessed August 2024.

[BCL+17] Daniel J Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, et al. Classic mceliece: conservative code-based cryptography. *NIST submissions*, 1(1), 2017.

[BD02] Bruno Beauquier and Eric Darrot. On Arbitrary Size Waksman Networks and Their Vulnerability. *Parallel Processing Letters*, 2002.

[BEM+17] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong Privacy for Analytics in the Crowd. In *Symposium on Operating Systems Principles (SOSP)*, 2017.

[Ber20] Daniel J. Bernstein. Verified fast formulas for control bits for permutation networks. Cryptology ePrint Archive, Paper 2020/1493, 2020.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing: Improvements and Extensions. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[BKL+22] Ludovic Barman, Moshe Kol, David Lazar, Yossi Gilad, and Nickolai Zeldovich. Groove: Flexible Metadata-Private Messaging. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[Blo70] Burton H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. 1970.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[Buz20] Buzzfeed News. The FinCEN Files, 2020. https://www.buzzfeednews.com/article/jasonleopold/fincen-files-financial-scandal-criminal-networks. Accessed August 2024.

[CBC+23] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves. In *USENIX Security Symposium*, 2023.

[CC18] Scott D Constable and Steve Chapin. libOblivious: A C++ Library for Oblivious Data Structures and Algorithms. Technical report, Syracuse University Electrical Engineering and Computer Science, 2018.

[CCX+19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX Explained. Technical report, 2016.

[CDJ+17] David Chaum, Debajyoti Das, Farid Javani, Aniket Kate, Anna Krasnova, Joeri De Ruiter, and Alan T Sherman. cMix: Mixing with Minimal Real-Time Asymmetric Cryptographic Operations. In *International Conference on Applied Cryptography and Network Security (ACNS)*, 2017.

[CGB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2017.

[CGBM15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An Anonymous Messaging System Handling Millions of Users. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[CGF10] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable Anonymous Group Messaging. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2010.

[CGH+18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *Advances in Cryptology (CRYPTO)*, 2018.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private Information Retrieval. In *IEEE Foundations of Computer Science (FOCS)*, 1995.

[CGWF13] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively Accountable Anonymous Messaging in Verdict. In *USENIX Security Symposium*, 2013.

[Cha81] David L. Chaum. Untraceable Electronic mail, Return Addresses and Digital Pseudonyms. *Communications of the ACM (CACM)*, 1981.

[Cha88] David Chaum. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *Journal of Cryptology*, 1988.

[Cho17] Tung Chou. Mcbits revisited. In *International Conference on Cryptographic Hardware and Embedded Systems*, 2017.

[CSM+20] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private Group Messaging with Hidden Access Patterns. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[CVM+21] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. Voltpillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *USENIX Security Symposium*, 2021.

[CWJ+16] Feng Chen, Shuang Wang, Xiaoqian Jiang, Sijie Ding, Yao Lu, Jihoon Kim, S Cenk Sahinalp, Chisato Shimizu, Jane C Burns, Victoria J Wright, Eileen Png, Martin L Hibberd, David D Lloyd, Hai Yang, Amalio Telenti, Cinnamon S Bloss, Dov Fox, Kristin Lauter, and Lucila Ohno-Machado. PRINCESS: Privacy-protecting Rare disease International Network Collaboration via Encryption through Software guard extensionS. *Bioinformatics*, 2016.

[CWWZ10] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[CZRZ17] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2017.

[Dan03] George Danezis. Statistical Disclosure Attacks. In *IFIP International Information Security Conference*, 2003.

[DDCO17] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. Privacy-Preserving Computation with Trusted Computing via Scramble-then-Compute. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2017.

[DDS09] George Danezis, Claudia Diaz, and Paul Syverson. Systems for Anonymous Communication. *CRC Handbook of Financial Cryptography and Security*, 2009.

[Deu19] Deutsche Welle. Iran defends execution of gay people, 2019. https://www.dw.com/en/iran-defends-execution-of-gay-people/a-49144899. Accessed August 2024.

[DFD+21] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2021.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography Conference (TCC)*, 2006.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, 2004.

[DO20] Sam Dittmer and Rafail Ostrovsky. Oblivious tight compaction in O(n) time with smaller constant. In *Conference on Security and Cryptography for Networks (SCN)*, 2020.

[EB22] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *Network and Distributed System Security Symposium, (NDSS)*, 2022.

[Ebb15] Kris Vestergaard Ebbesen. On the practicality of data-oblivious sorting. Master's thesis, Aarhus Universitet, Datalogisk Institut, 2015.

[ECGZB21] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy. In *USENIX Security Symposium*, 2021.

[EZ19] Saba Eskandarian and Matei Zaharia. ObliDB: Oblivious Query Processing for Secure Databases. *Proceedings of the VLDB Endowment*, 2019.

[FO21] Brett Hemenway Falk and Rafail Ostrovsky. Secure Merge with O (n log log n) Secure Operations. In *Information-Theoretic Cryptography (ITC 2021)*, 2021.

[GESM17] J. Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. *European Workshop on Systems Security (EuroSec)*, 2017.

[GHL18] Nethanel Gelernter, Amir Herzberg, and Hemi Leibowitz. Two Cents for Strong Anonymity: The Anonymous Post-office Protocol. In *Cryptology and Network Security (CANS)*, 2018.

[GI14] Niv Gilboa and Yuval Ishai. Distributed Point Functions and Their Applications. In *Advances in Cryptology (EUROCRYPT)*, 2014.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)*, 1996.

[Goo11] Michael T Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

[Gua13] The Guardian. NSA Files Decoded: What the revelations mean for you, 2013. https://www.theguardian.com/world/interactive/2013/nov/01/snowden-nsa-files-surveillance-revelations-decoded#section/1. Accessed August 2024.

[GWC+23] Tianyao Gu, Yilei Wang, Bingnan Chen, Afonso Tinoco, Elaine Shi, and Ke Yi. Efficient Oblivious Sorting and Shuffling for Hardware Enclaves. Cryptology ePrint Archive, Paper 2023/1258, 2023. https://eprint.iacr.org/2023/1258.

[Hås06] Johan Håstad. The square lattice shuffle. *Random Structure Algorithms*, 2006.

[HKI+12] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *International Conference on Information Security and Cryptology (ICISC)*, 2012.

[HKI+13] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *Information Security and Cryptology (ICISC)*, 2013.

[HOW22] William Holland, Olga Ohrimenko, and Anthony Wirth. Efficient Oblivious Permutation via the Waksman Network. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2022.

[HSJ86] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 1986.

[HSQ+23] Justin Hendrix, Caroline Sinders, Cooper Quintin, Leila Wagner, Tim Bernard, and Ami Mehta. What is secure? analysis of popular messaging apps. *Tech Policy Press*, June 2023.

[IET18] IETF. TLS v1.3, 2018. https://www.rfc-editor.org/rfc/rfc8446.txt. Accessed August 2024.

[Int12] Intel. Intel Trusted Execution Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html, 2012. Accessed August 2024.

[Int18] Intel. Q3 2018 Speculative Execution Side Channel Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html, 2018. Accessed August 2024.

[Int19] Intel. Intel Processors Voltage Settings Modification Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html, 2019. Accessed August 2024.

[Int20] Intel. 2020.2 IPU - Intel RAPL Interface Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00389.html, 2020. Accessed August 2024.

[JWC+23] Peipei Jiang, Qian Wang, Jianhao Cheng, Cong Wang, Lei Xu, Xinyu Wang, Yihao Wu, Xiaoyuan Li, and Kui Ren. Boomerang: Metadata-Private Messaging under Hardware Trust. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[KAL+15] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit Fingerprinting Attacks: Passive Deanonymization of Tor Hidden Services. In *USENIX Security Symposium*, 2015.

[KBS+19] Christiane Kuhn, Martin Beck, Stefan Schiffner, Eduard Jorswieck, and Thorsten Strufe. On Privacy Notions in Anonymous Communication. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2019.

[KCGDF17] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally Scaling Strong Anonymity. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[KCY23] Fumiyuki Kato, Yang Cao, and Masatoshi Yoshikawa. OLIVE: Oblivious Federated Learning on Trusted Execution Environment Against the Risk of Sparsification. *Proceedings of the VLDB Endowment*, 2023.

[KKS20] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient Oblivious Database Joins. *Proceedings of the VLDB Endowment*, 2020.

[KLD20] Albert Kwon, David Lu, and Srinivas Devadas. XRD: Scalable Messaging System with Cryptographic Privacy. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2020.

[KLDF16] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2016.

[KPW16] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. Technical report, 2016. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[Lee85] Kyungsook Yoon Lee. On the rearrangeability of $2(\log_2 n) - 1$ stage permutation networks. *IEEE Transactions on Computers*, 1985.

[Lee87] Kyungsook Yoon Lee. A New Benes Network Control Algorithm. *IEEE Transactions on Computers*, 1987.

[Lei84] Tom Leighton. Tight bounds on the Complexity of Parallel Sorting. In *ACM Symposium on Theory of Computing (STOC)*, 1984.

[Lew18] Sarah Jamie Lewis. Cwtch: Privacy Preserving Infrastructure for Asynchronous, Decentralized, Multi-Party and Metadata Resistant Applications, 2018. https://cwtch.im/cwtch.pdf.

[LGZ18] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[LJF+20] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *USENIX Security Symposium*, 2020.

[LK22] Donghang Lu and Aniket Kate. RPM: Robust Anonymity at Scale. Cryptology ePrint Archive, Paper 2022/1037, 2022. https://eprint.iacr.org/2022/1037.

[LKO+21] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[LKS+19] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanovic. Keystone: An Open Framework for Architecting TEEs, 2019.

[LMS97] Tom Leighton, Yuan Ma, and Torsten Suel. On probabilistic networks for selection, merging, and sorting. *Theory of Computing Systems*, 1997.

[LS22] Wei-Kai Lin and Elaine Shi. Optimal Sorting Circuits for Short Keys. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2022.

[LSG+17] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*, 2017.

[LSSD22] Simon Langowski, Sacha Servan-Schreiber, and Srinivas Devadas. Trellis: Robust and Scalable Metadata-private Anonymous Broadcast. Cryptology ePrint Archive, Paper 2022/1548, 2022. https://eprint.iacr.org/2022/1548.

[LSX19] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the n log n barrier for oblivious sorting? In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2019.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[LYK+19] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and Its Application to Anonymous Communication. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[LZ16] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping Secure Communication Without Leaking Metadata. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[MB09] Prateek Mittal and Nikita Borisov. ShadowWalker: Peer-to-peer Anonymous Communication using Redundant Structured Topologies. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[MB12] Prateek Mittal and Nikita Borisov. Information Leaks in Structured Peer-to-Peer Anonymous Communication Systems. *ACM Transactions on Information and System Security (TISS)*, 2012.

[MD04] Nick Mathewson and Roger Dingledine. Practical Traffic Analysis: Extending and Resisting Statistical Disclosure. In *International Workshop on Privacy Enhancing Technologies*, 2004.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[MOG+20] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[Mox17] Moxie Marlinspike. Technology preview: Private contact discovery for Signal, 2017. Technologypreview:PrivatecontactdiscoveryforSignal. Accessed August 2024.

[MPC+18] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An Efficient Oblivious Search Index. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, 1995.

[MVH+20] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *USENIX Security Symposium*, 2020.

[MZ14] John C Mitchell and Joe Zimmerman. Data-oblivious data structures. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, 2014.

[NS82] David Nassimi and Sartaj Sahni. Parallel Algorithms to Set Up the Benes Permutation Network. *IEEE Transactions on Computers*, 1982.

[NSSD22] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-Bandwidth Anonymous Broadcast with Malicious Security. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2022.

[NW06] Arjun Nambiar and Matthew Wright. Salsa: A Structured Approach to Large-Scale Anonymity. In *ACM conference on Computer and Communications Security (CCS)*, 2006.

[NYR14] NYR Daily. We Kill People Based on Metadata, 2014. https://www.nybooks.com/daily/2014/05/10/we-kill-people-based-metadata/. Accessed August 2024.

[OGTU14] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. The Melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2014.

[OSF+16] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, 2016.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *RSA Conference (CT-RSA)*, 2006.

[PAS+20] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *USENIX Security Symposium*, 2020.

[PHE+17] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix Anonymity System. In *USENIX Security Symposium*, 2017.

[Pip93] Nicholas Pippenger. Self-Routing Superconcentrators. In *ACM Symposium on Theory of Computing (STOC)*, 1993.

[PLP+16] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website Fingerprinting at Internet Scale. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. 2004.

[PRR09] Andriy Panchenko, Stefan Richter, and Arne Rache. NISAN: Network Information Service for Anonymization Networks. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[PSKK+23] A Pranav Shriram, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Ruffle: Rapid 3-party shuffle protocols. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 3, 2023.

[RBO89] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In *ACM Symposium on Theory of Computing (STOC)*, 1989.

[Reu18] Reuters. UN Experts says Egypt systematically targets rights activists, 2018. https://www.dw.com/en/iran-defends-execution-of-gay-people/a-49144899. Accessed August 2024.

[RLT15] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium*, 2015.

[Rog15] Phillip Rogaway. The Moral Character of Cryptographic Work. 2015. https://eprint.iacr.org/2015/1162.

[RP02] Marc Rennhard and Bernhard Plattner. Introducing MorphMix: peer-to-peer based anonymous Internet usage with collusion detection. In *ACM workshop on Privacy in the Electronic Society (WPES)*, 2002.

[RS21] Vijaya Ramachandran and Elaine Shi. Data Oblivious Algorithms for Multicores. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2021.

[SCA18] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[SCM05] Len Sassaman, Bram Cohen, and Nick Mathewson. The Pynchon Gate: A Secure Method of Pseudonymous Mail Retrieval. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2005.

[SCNS16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[SDH+10] Max Schuchard, Alexander W. Dean, Victor Heorhiadi, Nicholas Hopper, and Yongdae Kim. Balancing the Shadows. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2010.

[SDS02] Andrei Serjantov, Roger Dingledine, and Paul Syverson. From a Trickle to a Flood: Active Attacks on Several Mix Types. In *International Workshop on Information Hiding*, 2002.

[SG24] Sajin Sasy and Ian Goldberg. SoK: Metadata-Protecting Communication Systems. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2024.

[SGF18] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[Sha79] Adi Shamir. How to Share a Secret. *Communicatios of the ACM*, 1979.

[Sig13] Signal Foundation. Advanced Cryptographic Ratcheting, 2013. https://signal.org/blog/advanced-ratcheting/. Accessed August 2024.

[SJG22a] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast fully oblivious compaction and shuffling. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[SJG22b] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast fully oblivious compaction and shuffling. https://crysp.uwaterloo.ca/software/obliv/, 2022. Software artifact.

[SJG23a] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Waks-On/Waks-Off: Fast Oblivious Offline/Online Shuffling and Sorting with Waksman Networks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[SJG23b] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Waks-On/Waks-Off: Fast Oblivious Offline/Online Shuffling and Sorting with Waksman Networks. https://crysp.uwaterloo.ca/software/obliv/, 2023. Software artifact.

[SLHS+18] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. Efficient parallel random sampling—vectorized, cache-efficient, and online. *ACM Transactions on Mathematical Software (TOMS)*, 2018.

[SLKP17] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[SO19] Sajin Sasy and Olga Ohrimenko. Oblivious Sampling Algorithms for Private Data Analysis. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[SSA+18] Fatemeh Shirazi, Milivoj Simeonovski, Muhammad Rizwan Asghar, Michael Backes, and Claudia Diaz. A Survey on Routing in Anonymous Communication Protocols. *ACM Computing Surveys*, 2018.

[Sta90] Ernst Stadlober. The ratio of uniforms approach for generating discrete random variates. *Journal of computational and applied mathematics*, 1990.

[STA19] Nigel P Smart and Younes Talibi Alaoui. Distributing any Elliptic Curve Based Protocol. In *IMA International Conference on Cryptography and Coding (IMACC)*, 2019.

[SZB+24] Florian Sieck, Zhiyuan Zhang, Sebastian Berndt, Chitchanok Chuengsatiansup, Thomas Eisenbarth, and Yuval Yarom. TeeJam: Sub-Cache-Line Leakages Strike Back. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, 2024.

[TB06] Parisa Tabriz and Nikita Borisov. Breaking the Collusion Detection Mechanism of MorphMix. In *Privacy Enhancing Technologies (PET)*, 2006.

[TGL+17] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A Distributed Metadata-Private Messaging System. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[TGS23] Afonso Tinoco, Sixiang Gao, and Elaine Shi. EnigMap: External-Memory oblivious map for secure enclaves. In *USENIX Security Symposium*, 2023.

[USO14] An Open Letter from US Researchers in Cryptography and Information Security, 2014. http://masssurveillance.info/. Accessed August 2024.

[VBPS17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution*, 2017.

[VDHLZZ15] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[VMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[VSH22] A. Vadapalli, K. Storrier, and R. Henry. Sabre: Sender-Anonymous Messaging with Fast Audits. In *IEEE Symposium on Security and Privacy (S&P)* , 2022.

[VWK+17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*, 2017.

[Wak68] Abraham Waksman. A permutation network. *Journal of the ACM (JACM)*, 1968.

[Was18] Washington Post. Leak charges against Treasury official show encrypted apps only as secure as you make them, 2018. https://www.washingtonpost.com/news/powerpost/paloma/the-cybersecurity-202/2018/10/18/the-cybersecurity-202-leak-charges-against-treasury-official-show-encrypted-apps-

only-as-secure-as-you-make-them/5bc74eaa1b326b7c8a8d1a3a/. Accessed August 2024.

[WCGFJ12a] David I Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Scalable Anonymous Group Communication in the Anytrust Model. In *European Workshop on System Security (EuroSec 2012)*, 2012.

[WCGFJ12b] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in Numbers: Making Strong Anonymity Scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[WCN+14] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *USENIX Security Symposium*, 2014.

[WIR14] WIRED. Whatsapp Switches to End-to-End Encryption, 2014. https://www.wired.com/2014/11/whatsapp-encrypted-messaging/. Accessed August 2024.

[WMB10] Qiyan Wang, Prateek Mittal, and Nikita Borisov. In Search of an Anonymous and Secure Lookup: Attacks on Structured Peer-to-Peer Anonymous Communication Systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[ZDB+17] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[ZH11] Bassam Zantout and Ramzi Haraty. I2P Data Communication System. In *International Conference on Networks (ICN)*, 2011.

[ZWR+16] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting Square-Root ORAM: Efficient Random access in Multi-Party Computation. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

Appendices

Appendix A

Largest Experiment Reported for prior MPCS

Table A.1 reports the details of the largest experiment conducted by each of the systems we discussed in Section 3.3. This provides a concrete timing benchmark for each system albeit constrained by their choices of experimental setup. This prevents us from an apples-to-apples comparison across designs since each of these works i) leverage different underlying machines for their experiments, ii) choose experiment setups that highlight their system's advantages, and iii) have nuanced goals and assumptions. For instance, timing benchmarks of asynchronous and synchronous systems typically have very different implications. Systems from the DP family like Karaoke transmit one real message from each client in a round implying a high throughput. On the flip side, private-write based systems like Sabre perform low-latency delivery of a single message from a single client, and their throughput is fundamentally limited by n, the number of clients in the system and the time taken to process a single request, which scales with n.

Nonetheless these timing benchmarks give us a concrete sense of the state of affairs of current MPCS designs, and the stark gap in performance with popular messenger choices today that can accommodate billions of users with latencies in the order of milliseconds. We also note that the timings reported in Table A.1 are a lower bound since the timing benchmark quoted for several of the designs in the table do not account for the overheads of setup phases such as the shuffling of keys in the DC-net based schemes, or the cost of the dialing protocol in synchronous schemes. The cost of these phases would further increase the timing benchmark, although such costs are typically amortized over several rounds of communication. Next we contrast benchmarks of representative can-

didates from the different families of metadata-protecting communication systems we have discussed.

Synchronous Constructions. Systems from the DC-net, mixnet, DP, and SMC based families are typically synchronous in nature. The exception is Loopix, which supports asynchrony at the expense of trusting special service provider nodes in the system. DC-net based schemes scale very poorly as one would expect. The most efficient construction we have seen so far is Verdict [CGWF13] with 1000 clients, and 24 servers, takes more than $10\,\mathrm{s}$ to transmit one message of 128 bytes, and note that this is in fact assuming no client is actively attempting to disrupt the network, nor accounting for the shuffle protocol that is required to slot all the participants in the network for its communication round, both of which further exacerbates the time to transmit a message. Furthermore as detailed in Section 3.4, none of these construction can scale horizontally.

SMC-based constructions can serve more clients than the DC-net based ones, but note that more servers do not translate to reduced latency or higher throughput in these SMC-based systems; i.e., these schemes too do not scale horizontally. In terms of performance, Clarion in their prototype demonstrate that they can transmit 1 message of 160 bytes from each of the 10^6 clients to its intended recipient in $80\,\mathrm{s}$ with a three-server instantiation of their protocol.

In contrast, recent constructions from the mixnet based and DP based families have been designed with the explicit goal of horizontal scalability. For instance, the mixnet based scheme XRD's experiments demonstrates that with 2×10^7 clients all sending a 100-byte message to another client in the system takes $251\,\mathrm{s}$. While Karaoke the most recent work in the DP based family in their experiments demonstrates that with 1.6×10^7 clients (a similar number of clients as XRD) and each sending a message to another client in the system, can transmit all of these messages in a 100-server instantiation of their protocol in $28\,\mathrm{s}$; i.e., almost an order of magnitude faster, at the expense of the weaker privacy guarantees due to DP as we detailed in Section 3.3.3. Ultimately the biggest drawback in these schemes remains the synchronicity assumption. Schemes like Karaoke can serve close to 17 million clients in tens of seconds, but the expectation that all these clients will remain online without any disruption in perpetuity is extremely unrealistic, and disconnections of a user during a conversation harms the privacy of both participants of the conversation.

Asynchronous Constructions. PIR can be leveraged to design asynchronous MPCS, as seen in Express [ECGZB21], Talek [CSM+20], and Sabre [VSH22]. The only other

Table A.1: Details of the largest experiments reported (if experimental results were provided) by the systems detailed in Section 3.3.

System	Year	Type	Family	Largest Experiment Reported $(n\|s\|m\|b\|t)$				
				Clients (n)	Messages Sent (s)	Servers (m)	Message Size (b)	Time (t)
Vuvuzela [VDHLZZ15]	2015	CUS (Mix)	DP	2×10^6	2×10^6	3	256	55 s
Pung [AS16]	2016	CUS (RUMS)	PIR	10^6	1	1	288	1.3 s
AnonPOP [GHL18]	2016	CUS (Mix)	M	5×10^5	5×10^5	4	1000	30 s
Stadium [TGL+17]	2017	CUS (Mix)	DP	5×10^7	5×10^7	100	136	200 s
MCMix [AKTZ17]	2017	CUS	SMC	10^5	10^5	3	256	100 s
SealPIR [ACLS18]	2018	CUS (RUMS)	PIR	2.56×10^5	1	1	288	0.51 s
Karaoke [LGZ18]	2018	CUS (Mix)	DP	1.6×10^7	1.6×10^7	100	256	28 s
XRD [KLD20]	2019	CUS	M	8×10^6	8×10^6	100	256	$>$ 16 m
Clarion [EB22]	2021	CUS	SMC	10^6	10^6	3	160	80 s
Groove [BKL+22]	2022	CUS (Mix)	DP	1.5×10^8	1.5×10^8	100	56	80 s
Boomerang [JWC+23]	2023	CUS	TEE	2^{20}	2^{20}	32	256	10 s
Dissent [CGF10]	2010	CUS/SUBS	DC	40	1	0	10^6	$>$ 14 m
D3 [WCGFJ12a]	2012	CUS/SUBS	DC	576	1	10	256	$>$ 2 m
Dissent v2 [WCGFJ12b]	2012	CUS/SUBS	DC	1000	1	24	128	$>$ 10 s
Verdict [CGWF13]	2013	CUS/SUBS	DC	1000	1	8	128	$>$ 10 s
Riposte [CGBM15]	2015	SUBS	RPIR	3.8×10^5	1	3	160	0.35 s
Riffle [KLDF16]	2016	SUBS	M	1.2×10^6	10^6	3	160	$>$ 10 s
Atom [KCGDF17]	2017	SUBS	M	10^7	10^7	1024	32	28 m
AsynchroMix [LYK+19]	2019	SUBS	SMC	4096	4096	100	32	120 s
PowerMix [LYK+19]	2019	SUBS	SMC	1024	1024	100	32	140 s
Blinder [APY20]	2020	SUBS	SMC	10^6	10^6	20	160	8 m
Spectrum [NSSD22]	2021	SUBS	RPIR	10^5	1	2	10^6	$>$ 500 s
RPM [LK22]	2022	SUBS	SMC	1.6×10^5	1.6×10^5	4	16	48.14 s
Sabre [VSH22]	2022	SUMS/SUBS	RPIR	2^{18}	1	2	1000	0.05 s
Express [ECGZB21]	2021	SUMS	RPIR	2^{18}	1	2	1000	0.5 s
cMix [CDJ+17]	2016	RUS (Mix)	M	500	1000	5	256	4.6 s
Loopix [PHE+17]	2017	RUS (Mix)	M	500	3×10^4	10	224	1.9 s
Talek [CSM+20]	2020	RUS (RUMS)	PIR	3.2×10^4	1	3	1000	1.7 s

asynchronous design is Loopix (mixnet), but it requires trusting service provider nodes to support asynchrony.

Sabre is the latest iteration of private-write based metadata-hiding communication networks. In their work, their experiments demonstrate that in a two-server instance of Sabre with 2^{18} registered mailboxes a client can deliver a message to any of these mailboxes in 0.05 s, which is the smallest latency (ignoring throughput) by almost two orders of magnitude in comparison with the other works we have detailed. However as we mention in Section 3.3.5, as more users enter such systems the cost to deliver a message increases linearly. Furthermore, these systems incur quadratic overheads for scaling horizontally.

Loopix on the other hand attains asynchronous communication capabilities by leveraging a semi-trusted service provider node in their construction. Their experimental evaluation is limited to a 500-client instantiation with a network comprising of 4 service providers and 6 mixnet nodes. Their end-to-end latency in this setup is 1.9 s with other experiments that justify that with increase in clients and servers this end-to-end cost would remain unaffected, and is mostly dependent on the system parameter that tunes the delay for a message at each of these mixes. However, as we detail in Section 3.3.2, a malicious service provider node undermines recipients' metadata privacy.

Appendix B

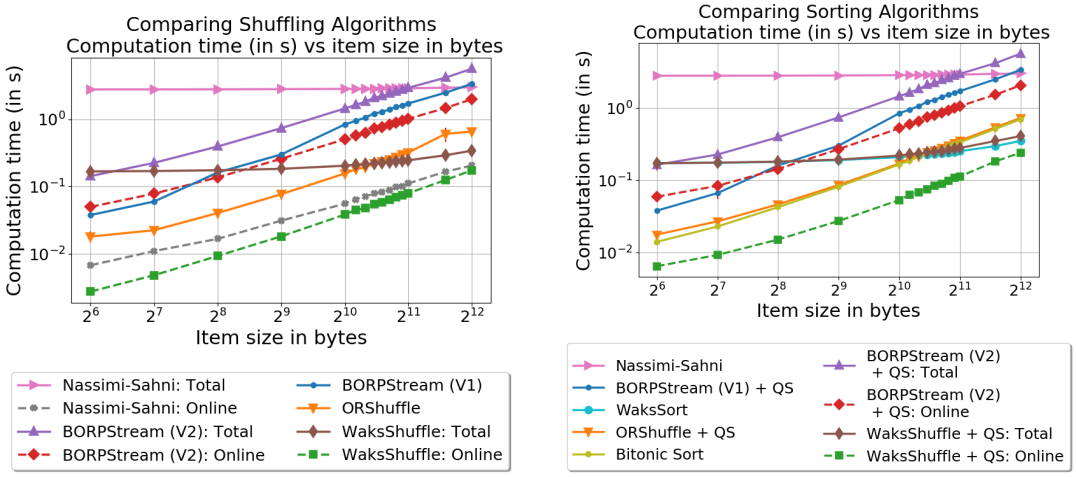
Additional Experiments for WaksShuffle and WaksShuffle+QS

B.1 Varying the Item Size

Figures B.1a and B.1b show shuffle and sort times for varying item sizes and $n = 2^{15}$ items. We observe qualitatively similar results to those for $n = 2^{20}$ items. In particular, we see the crossover points, at which the total times for WaksShuffle and WaksSort become the lowest, are still at an item size of about $\beta = 1400$ bytes. We also see that the online times for WaksShuffle and WaksShuffle+QS remain the lowest.

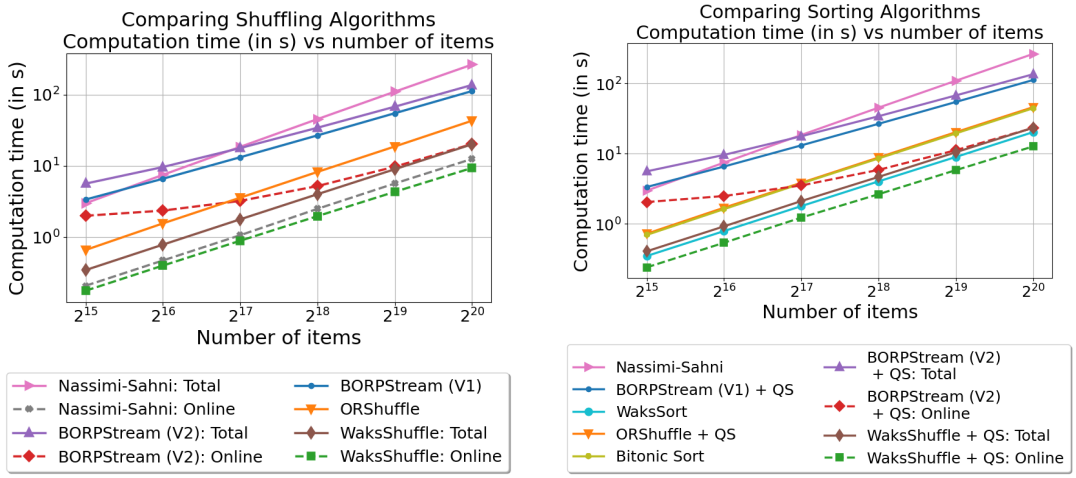
B.2 Varying the Number of Items

Figures B.2a and B.2b show the shuffle and sort times for varying numbers of items and an item size of $\beta = 2^{12}$ bytes. For all n tested, WaksShuffle and WaksSort have the lowest total times for shuffling and sorting, respectively. Similarly, WaksShuffle and WaksShuffle+QS always have the lowest online times for shuffling and sorting, respectively.



(a) Shuffling (b) Sorting

Figure B.1: Comparing time taken for shuffling/sorting against item size with $n = 2^{15}$ items. Both axes are log scale; error bars are too small to see. Dashed lines are the online timing component for algorithms that benefit from offline/online split.



(a) Shuffling (b) Sorting

Figure B.2: Comparing time taken for shuffling against number of items with an item size of $\beta = 2^{12}$ bytes. Both axes are log scale; error bars are too small to see. Dashed lines are the online timing component for algorithms that benefit from offline/online split.