UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Achieving Practical Access Pattern Privacy in Data Outsourcing

Permalink
https://escholarship.org/uc/item/8672k3v6

Author
Dautrich, Jonathan L.

Publication Date
2014

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Achieving Practical Access Pattern Privacy in Data Outsourcing

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Jonathan Jack Laurence Dautrich Jr.

June 2014

Dissertation Committee:

Dr. Chinya Ravishankar, Chairperson
Dr. Marek Chrobak
Dr. Tao Jiang
Dr. Vassilis Tsotras

Copyright by
Jonathan Jack Laurence Dautrich Jr.
2014

The Dissertation of Jonathan Jack Laurence Dautrich Jr. is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my advisor, Professor Chinya Ravishankar, for his invaluable guidance and support over the past six years. I also wish to acknowledge my collaborators Emil Stefanov and Elaine Shi for their insights and assistance with Burst ORAM.

My work was supported in part by the National Physical Science Consortium Graduate Fellowship and by grant N00014-07-C-0311 from the Office of Naval Research.

The contents of Chapter 2, *Compromising Privacy in Precise Query Protocols*, were published in [28]. Chapter 3, *Security Limitations of Using Secret Sharing for Data Outsourcing*, was published in [27]. Chapter 4, *Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns*, will appear in [29].

To my wife Daniella, whose extraordinary patience and selfless support have been a constant inspiration to me:

You are a treasure.

To my parents Jon and Linda, who taught me hard work, discipline, and balance:

Thank you for the innumerable gifts with which you have blessed me.

To all my family and friends whose undying confidence has kept me going:

Your encouragement means more to me than you know.

To Emil Stefanov, a great friend and collaborator:

You will be missed.

ABSTRACT OF THE DISSERTATION

Achieving Practical Access Pattern Privacy in Data Outsourcing

by

Jonathan Jack Laurence Dautrich Jr.

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2014
Dr. Chinya Ravishankar, Chairperson

Cloud computing allows customers to outsource the burden of data management and benefit from economy of scale, but privacy concerns hinder its growth. Even when the stored data are encrypted, access patterns may leak valuable information. We consider the challenge of providing efficient, privacy-preserving access to data outsourced to an untrusted cloud provider.

As motivation for providing data outsourcing protocols with strong privacy guarantees, we introduce two novel attacks against existing "secure" schemes. The first compromises a protocol based on Shamir's Secret Sharing Algorithm that makes invalid security claims. The second sorts encrypted records by their plaintext query-attribute values, and can be applied to any protocol that supports range queries and returns the precise set of encrypted records needed to satisfy each query.

Oblivious RAM (ORAM) protocols guarantee full access pattern privacy, but even the most efficient ORAMs proposed to date incur large bandwidth costs and high response times. We present two novel ORAM protocols. The first minimizes up-front bandwidth costs and achieves near-optimal response times during bursts of requests, while maintaining total bandwidth costs competitive with the best existing ORAM.

The second combines ORAM with Private Information Retrieval (PIR) techniques in order to achieve the lowest total bandwidth cost of any ORAM protocol known to date.

Finally, we introduce a generalized form of ORAM called Tunably-Oblivious Memory, which relaxes ORAM's privacy guarantees to allow a bounded amount of information leakage in exchange for lower bandwidth costs. We propose a novel special-purpose Tunably-Oblivious Memory protocol that achieves bandwidth costs lower than those of the best existing ORAM for suitable workloads, while leaking only a few bits of information per query.

Contents

List of Figures **xiii**

List of Tables **xvii**

1 Introduction **1**

1.1 Simple Encryption and Access Pattern Leakage 2
1.2 Oblivious RAM and Private Information Retrieval 3
1.3 Tunably-Oblivious Memory 5

2 Compromising Privacy in Precise Query Protocols **7**

2.1 Introduction 7
2.1.1 Our Contributions 9
2.2 Related Work 10
2.2.1 Precise Query Protocol Schemes 10
2.2.2 Imprecise Query Protocols 12
2.2.3 Prior Work on Privacy Loss 12
2.3 Attack Model and Outline 13
2.3.1 Precise Query Protocols (PQPs) 13
2.3.2 Permissible Permutations and Loci 15
2.3.3 Using PQ-Trees to Maintain $\mathcal{P}$ 17
2.3.4 Characteristic Examples 18
2.4 Identifying Permissible Loci 20
2.4.1 Algorithm Outline and Terminology 20
2.4.2 Identifying Loci for Children of Q-Nodes 22
2.4.3 Identifying Loci for Children of P-Nodes 22
2.4.3.1 Each Child Considered Separately 24
2.4.3.2 All Children Considered Together 25
2.4.4 Analysis and Space-Time Tradeoff 28
2.4.5 The κ-Pruning Variant 31
2.5 Measuring Privacy Loss 33
2.5.1 Alternate and Related Metrics 33
2.6 Experiments and Evaluation 35
2.6.1 Progress Before Privacy Compromise 36
2.6.2 Higher Thresholds and Larger κ 38
2.6.3 Permutation Entropy 40
2.6.4 Effects of Indexes on PQP Privacy 41

2.6.5 Consequences and Alternatives . . . 43
2.6.5.1 Assuming Attribute Distributions are Hidden . . . 43
2.6.5.2 Abandoning PQPs Altogether . . . 43
2.7 Conclusion . . . 44

3 Security Limitations of Using Secret Sharing for Data Outsourcing 45
3.1 Introduction . . . 45
3.1.1 Our Contribution . . . 47
3.2 Data Outsourcing Using Secret Sharing . . . 48
3.2.1 Shamir's Secret Sharing . . . 48
3.2.2 Data Outsourcing via Secret Sharing . . . 49
3.2.3 Security . . . 51
3.2.4 Supporting Range and Aggregation Queries . . . 51
3.3 Attack Description . . . 52
3.3.1 Recovering Secrets when p is Known and $\vec{X}$ is Private . . . 53
3.3.2 Recovering p when $\vec{X}$ and p are Private . . . 54
3.3.2.1 Computing δ_1, δ_2. . . . 54
3.3.2.2 Size of δ_1, δ_2. . . . 55
3.3.2.3 Recovering p from δ_1, δ_2. . . . 56
3.3.3 Attack Complexity . . . 57
3.3.4 Example Attack for $k = 2$ . . . 57
3.4 Aligning Shares and Discovering Secrets . . . 58
3.4.1 Aligning Shares . . . 59
3.4.2 Discovering $k + 2$ Secrets . . . 59
3.4.3 Inferring Order in the HJ Scheme . . . 60
3.5 Attack Implementation and Experiments . . . 62
3.5.1 Time Measurements . . . 62
3.5.2 Failure Rate Measurements . . . 64
3.6 Attack Mitigations . . . 66
3.7 Related Work . . . 67
3.8 Conclusion . . . 68

4 Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns 70
4.1 Introduction . . . 70
4.1.1 Burst ORAM Contributions . . . 72
4.2 Related Work . . . 74
4.3 Preliminaries . . . 75
4.3.1 Bandwidth Costs . . . 75
4.3.2 Response Time . . . 76
4.3.3 ObliviStore ORAM . . . 77
4.4 Overview of our Approach . . . 80
4.5 Prioritizing and Reducing Online IO . . . 82
4.5.1 Prioritizing Online IO . . . 83
4.5.2 XOR Technique: Reducing Online IO . . . 84
4.5.2.1 XOR Technique Details . . . 84
4.5.2.2 Handling early shuffle reads . . . 86
4.5.2.3 Comparison with ObliviStore . . . 86
4.6 Scheduling and Reducing Shuffle IO . . . 87

4.6.1 Shuffle Jobs 88
4.6.2 Prioritizing Efficient Jobs 89
4.6.3 Reducing Shuffle IO via Level Caching 90
4.6.3.1 Level Caching in Burst ORAM 91
4.7 Detailed Burst ORAM Design 91
4.7.1 Overall Architecture 92
4.7.2 Semaphores 94
4.7.3 Detailed System Behavior 95
4.8 Pseudocode 98
4.9 Evaluation 100
4.9.1 Methodology 100
4.9.1.1 Baselines 100
4.9.1.2 Metrics 101
4.9.1.3 Workloads 103
4.9.2 Simulator 104
4.9.3 Endless Burst Experiments 105
4.9.4 Two-Burst Experiments 107
4.9.5 NetApp Workload Experiments 109
4.10 Reducing Online Bandwidth Costs of SR-ORAM 111
4.11 Conclusion 113

5 Combining ORAM with PIR to Minimize Bandwidth Costs 114
5.1 Introduction 114
5.1.1 Our Contributions 116
5.2 Related Work 117
5.3 Preliminaries 118
5.3.1 Private Information Retrieval (PIR) 118
5.3.2 Oblivious RAM (ORAM) 120
5.3.3 ObliviStore 121
5.3.3.1 Partition and Level Structure 121
5.3.3.2 Requests and Evictions 122
5.3.3.3 Shuffling 123
5.3.3.4 Early Shuffle Reads 124
5.3.3.5 Level Compression 125
5.3.3.6 Bandwidth Costs 125
5.4 Integrating PIR 126
5.4.1 Choosing a PIR Technique 128
5.4.2 Trostle-Parrish PIR 128
5.5 Altering Level Size Factors 129
5.5.1 Effects of Increasing Level Size Factors 130
5.5.2 Non-Uniform Level Size Factors 132
5.5.2.1 Practical Limits on Level Size Factor Growth 133
5.6 Eliminating Unused Dummies 134
5.6.1 Causes of Eviction/Request Difference 134
5.6.2 Source of Unused Dummies 134
5.6.3 Proposed Mitigations 135
5.7 Bandwidth Cost Enhancements 136
5.7.1 Enhancement Techniques 137
5.7.2 Effects and Tradeoffs 137

5.8 Experiments . . . 138
5.8.1 Simulators and Implementations . . . 138
5.8.1.1 ORAM Simulator . . . 138
5.8.1.2 PIR Implementation . . . 139
5.8.2 Evaluating OS+PIR for Mobile Devices . . . 139
5.8.3 Varying Block Count N . . . 141
5.8.4 Varying Level Size Configuration K . . . 142
5.8.5 Varying Eviction Rate . . . 142
5.8.6 Evaluating Individual Enhancements . . . 143
5.9 Conclusion . . . 144

6 Tunably-Oblivious Memory: Generalizing ORAM to Enable Privacy-Efficiency Tradeoffs 146
6.1 Introduction . . . 146
6.1.1 Our Contributions . . . 148
6.2 Related Work . . . 149
6.2.1 ORAM Protocols . . . 149
6.2.2 Partial Access Pattern Protection . . . 150
6.3 Tunably-Oblivious Memory . . . 151
6.3.1 ORAM Review . . . 151
6.3.2 TOM Model: Trading Off Obliviousness for Efficiency . . . 153
6.3.3 TOM Security Definition . . . 155
6.3.4 Paddable TOM Protocols . . . 157
6.3.5 Log-Spacing Strategy for Paddable Protocols . . . 158
6.4 Staggered-Bin TOM . . . 159
6.4.1 SBT Architecture . . . 160
6.4.2 SBT Operation . . . 161
6.4.3 SBT Security . . . 163
6.4.4 SBT Performance . . . 165
6.5 SBT Variants . . . 166
6.5.1 The 2-Choice SBT Variant . . . 166
6.5.1.1 Random Round Robin Algorithm . . . 167
6.5.1.2 2-Choice SBT Security . . . 168
6.5.1.3 2-Choice SBT Performance . . . 168
6.5.2 The SBT+ORAM Variant . . . 169
6.5.2.1 SBT+ORAM Security . . . 169
6.5.2.2 SBT+ORAM Performance . . . 170
6.5.3 The Multi-SBT Variant . . . 171
6.6 Performance Analyses and Proofs . . . 172
6.6.1 Problem Transformation . . . 173
6.6.2 SBT Analysis . . . 174
6.6.3 2-Choice SBT Analysis . . . 175
6.6.4 SBT+ORAM Analysis . . . 176
6.6.4.1 Balls and Urns Problem . . . 176
6.6.4.2 Blocks and Bins Problem . . . 181
6.7 Evaluation . . . 182
6.7.1 Bandwidth Cost Experiments . . . 182
6.7.1.1 Uniform Random Block Queries (Figures 6.4–6.6) . . . 183
6.7.1.2 Uniform/Zipf Fixed Sequence Queries (Figures 6.7–6.12) 183

6.7.1.3 Other Observations 185
6.7.2 Maximum Queue Length Measurements 185
6.7.3 Simulator Details 186
6.8 Conclusion 186

7 Conclusion **188**

List of Figures

2.1 *Database As a Service* with encrypted data, queries, and results. 8
2.2 Employee records in a PQP. Salaries are known, but not employee-salary relationships. 10
2.3 *Permissible loci* of etuples e_1 and e_2, given permissible permutations $\mathcal{P} = \{\pi_1, \pi_2, \pi_3, \pi_4\}$. 16
2.4 Permutation π_1 is permitted but π_2 is excluded as C_1 is split by $N \notin C_1$ in π_2. 19
2.5 PQ-tree with frontier $LMNKIJ$ reduced using clusters $\{I, J, K\}$, $\{L, M\}$. 19
2.6 Permutation π_1 is permitted but π_2 is excluded as C_2 is split by $I, J \notin C_2$ in π_2. 19
2.7 PQ-tree with frontier $NMKLJI$ reduced using $\{I, J, K, L\}$, $\{K, L, M, N\}$. 19
2.8 The clusters in $\mathcal{C}$ permit *only* π_c and $\overline{\pi_c}$. 20
2.9 PQ-tree fully reduced using all size-2 clusters. 20
2.10 Labeled PQ-tree (Example 4). 21
2.11 The loci Λ_{z_3} are offset from Λ_y by $\overleftarrow{\eta_{z_3}}$ when y's children are ordered left-to-right, and by $\overrightarrow{\eta_{z_3}}$ when ordered right-to-left. 22
2.12 For P-node y, each subset of children to the left of z_3 yield an offset η of Λ_{z_3} from Λ_y. 23
2.13 Identifying permissible loci for children of P-node y. Each arrow indicates expansions using the labeled spreads. Eight expansions are performed at each of three levels. (Example 6) 27
2.14 *Query Count* before *Privacy Compromise*, Random dataset. 36
2.15 *Total Return Count* before *Privacy Compromise*, Random dataset. 37
2.16 Random dataset, Gaussian widths: $N(10^5, 5 \times 10^4)$. 39
2.17 Random dataset, Uniform widths: $U(0, 10^8)$. 39
2.18 Salary dataset, Gaussian widths: $N(2 \times 10^4, 10^4)$. 40
2.19 Salary dataset, Uniform widths: $U(0, 373071)$. 40
2.20 Drop rates of permutation entropy for Gaussian and Uniform query width distributions on the Random and Salary datasets. 41
2.21 Effects of a binary range tree index on the average *Total Return Count* reached before *Privacy Compromise*. 42

3.1 Secret (salary) data from an employee table is split into shares and distributed to multiple data servers. A trusted client queries shares from the data servers and combines them to recover the secrets. 48

3.2 Range queries indicate that the secrets of shares y_1, y_2 are contiguous, as are those of y_3, y_2. Thus, the secret of y_2 falls between the secrets of y_1 and y_3, though either y_1 or y_3 may have the smallest secret. 61
3.3 Riverside dataset times, varied b 63
3.4 Riverside dataset times, varied k 63
3.5 Synthetic dataset times, varied b 64
3.6 Synthetic dataset times, varied k 64
3.7 Attack failure rates for varied k and β 65

4.1 **Effective IO.** Simplified scheme with sequential IO and contrived capacity for delaying offline IO. Three requests require same online IO (2), offline IO (5), and overall IO (7). Online IO for R1 can be handled immediately, so R1's effective IO is only 2. R2 must wait for 2 units of offline IO from R1, so its effective IO is 4. R3 must wait for the rest of R1's offline IO, plus one unit of R2's offline IO, so its effective IO is 6. . . . 77
4.2 **Reducing response time.** Because Burst ORAM (right) does much less online IO than ObliviStore (left) and delays offline IO, it is able to respond to ORAM requests much faster. In this (overfly simplified) illustration, the bandwidth capacity is enough to transfer 4 blocks concurrently. Both ORAM systems do the same amount of IO. 80
4.3 **Reducing online cost.** In ObliviStore (left) the online bandwidth cost is $O(\log N)$ blocks of IO on average. In Burst ORAM (right), we reduce online IO to only one block, improving handling of bursty traffic. . . . 83
4.4 XOR Technique Steps 85
4.5 **Burst ORAM Architecture.** Solid boxes represent key system components, while dashed boxes represent functionality and the effects of the system on IO. 92
4.6 **Burst ORAM Client Space Allocation.** Fixed client space is reserved for the position map and shuffle buffer. A small amount of overflow space is needed for blocks assigned but not yet evicted to partitions (*data cache* in [77]). Remaining client space is managed by *Local Space* semaphore and contains evictions, early shuffle reads, and the level cache. 93
4.7 Online bandwidth costs as a burst lengthens. Burst ORAM maintains low online cost regardless of burst length, unlike ObliviStore. 105
4.8 Effective bandwidth costs as a burst lengthens. Burst ORAM can handle most bursts with ∼1X effective cost. Each scheme's effective cost converges to its overall cost for long bursts. 106
4.9 Response times during two same-size bursts of just over 2^{17} requests spread evenly over 72 seconds. Client has space for at most 2^{18} blocks. No level caching causes early spikes due to extra early shuffle reads. . . 108
4.10 (Top) Burst ORAM achieves short response times in bandwidth-constrained settings. Since ObliviStore has high effective cost, it requires more available client-server bandwidth to achieve short response times. (Bottom) Burst ORAM response times are comparable to those of the insecure (without ORAM) scheme. 110

4.11 (Top) Insecure baseline (no ORAM) p-percentile response times for various p. (Bottom) Overhead (difference) between insecure baseline and Burst ORAM's p-percentile response times. Marked nodes show that when baseline p-percentile response times are < 100ms, Burst ORAM overhead is also < 100ms. 111
4.12 To achieve shorter response times, Burst ORAM incurs higher overall bandwidth cost than ObliviStore, most of which is consumed during idle periods. Level caching keeps bandwidth costs in check. Job prioritization does not affect overall cost, but does reduce effective costs and response times (Figures 4.8, 4.10). 112

5.1 In ObliviStore, when shuffles cascade upward, all levels ready to be reshuffled are downloaded at once, shuffled together with eviction blocks, and uploaded to a higher level. 124
5.2 In an OS+PIR request, since only one removed block is real, PIR reduces the response cost for each request to a constant or near-constant size. During shuffling, only blocks remaining in each level (most of which are real) need to be downloaded. Because of level compression, we need only transfer data of size equivalent to the real blocks being uploaded. In all, dummy transfers in OS+PIR are "free" — we effectively pay only to transfer real blocks. 127
5.3 Level configurations with size factors $k = 2$ and $k = 4$, both with a 15 real-block capacity. When shuffling, all sub-levels in a main level are combined to form one new sub-level in the next largest main level. The $k = 4$ configuration has fewer main levels, and thus lower shuffling costs. The $k = 2$ has fewer dummies and sub-levels, and thus lower disk and PIR costs. 131
5.4 Timing and bandwidth costs of Trostle-Parrish PIR implementation for varying numbers of blocks in PIR database, using $s = 512$ bits noise and 2MiB block size. 140

6.1 Client issues secret accesses (reads/writes) to the ORAM/TOM protocol, which translates them to a sequence of public accesses (stores/fetches) to the untrusted server. 152
6.2 The SBT in its initial state, with $n = 5$ blocks on the client, and $n(n+1)/2$ on the server. The empty server-side *incoming* bin will be filled in, one block at a time, by the n blocks from the client. 162
6.3 SBT after 3 steps. Server bins are accessed in a round-robin fashion. Blocks L, F, D have been fetched to the client-side *incoming* bin, and blocks R, Q, P stored to server-side *incoming* bin. 162
6.4 Uniform random block queries, varying N 184
6.5 Uniform random block queries, varying ℓ 184
6.6 Uniform random block queries, varying λ 184
6.7 Uniform fixed sequence queries, varying N 184
6.8 Uniform fixed sequence queries, varying ℓ 184
6.9 Uniform fixed sequence queries, varying λ 184
6.10 Zipf fixed sequence queries, varying N 184
6.11 Zipf fixed sequence queries, varying ℓ 184
6.12 Zipf fixed sequence queries, varying λ 184

6.13 Confirmation of analysis. Maximum observed H asymptotically dominated by analytically predicted values. $\mu = \ell/n$ 185

List of Tables

2.1 Asymptotic costs for existing PQP schemes, with large costs highlighted. Few papers gave these costs explicitly, so the table reflects our best-effort analysis. $|D|$ is domain size, n is the etuple count, C is the result set size, N is a 512–4096 bit number, K_g and K_s are costs of group and symmetric encryption operations, respectively. δ for OPES is small with unknown relation to n. 11
2.2 PQP and Attack Notation 14
2.3 Notation for Node y of PQ-Tree T 23

4.1 Algorithm Notation 98

5.1 Notation 119
5.2 Comparison of different bandwidth-efficient protocols given parameters tuned for current mobile devices with at least 64GiB storage. Common parameters: $N = 2^{22}$ data blocks, 2MiB block size, 64GiB total client storage. 141
5.3 Effect of increasing N on client/server storage and bandwidth cost, with $\epsilon = 1.1$ for OS+PIR, $\epsilon = 1.3$ for ObliviStore. 142
5.4 Effects of changing the level configuration K for $N = 2^{28}$ block count, 512TiB capacity, 512GiB total client storage. Product of all level size factors for each K is 2^{16}. 142
5.5 Effects of changing the eviction rate ϵ for OS+PIR with $N = 2^{28}$, 512TiB capacity, given a fixed total client storage of roughly 1TiB. Client space insufficient to support $\epsilon = 1.0$. 143
5.6 Effects of selectively applying/excluding enhancements to/from OS+PIR. Common parameters: $N = 2^{28}$, $\epsilon = 1.3$ base and 1.1 reduced, $K = (4, 64, 32, 4, 2)$, 479–491GiB total client storage. 144

6.1 Comparison of [77] with results based on our proposed Multi-SBT using the ORAM component from [77], with the parameterizations and costs given below and in [77], with 64 KB block size. Multi-SBT average cost is for uniform random queries of length $\ell = 4\sqrt{N}, \lambda = 8$. Max. cost is three times ceiling of ORAM cost. 149
6.2 SBT and TOM Notation 160

6.3 Comparison of our $\lambda - TOM$ protocols, block size B. Numbers are approximate; estimated average costs taken from Figures 6.4–6.6. Smaller C_{MAX} improves privacy/efficiency tradeoff. λ set to log of S_{MAX} to keep δ constant. $\lg \equiv \log_2$ 161

Chapter 1

Introduction

Cloud computing is a popular paradigm that allows a resource-constrained *client* to outsource data storage and management to a paid service provider, which we refer to as the *server*. Such cloud *servers* include Amazon Web Services [3], Google Cloud Platform [4], and Microsoft Azure [5]. These cloud services take advantage of economies of scale to reduce hardware and information technology management costs for their clients.

Since cloud servers are not under the client's full control, privacy concerns and data protection regulations lead many clients to keep sensitive data offline [20]. The server itself is the principal threat to privacy, since it controls all client data, queries, responses, and processing, so we treat the server as the primary adversary. We assume that servers are *honest but curious*, meaning that they correctly follow the prescribed protocol, but may still try to infer information about client data.

In this work, we investigate novel and existing protocols for protecting the privacy of outsourced data. We expose fundamental weaknesses in several existing protocols, and advocate the use of protocols that provide stronger privacy. We then propose

and evaluate three privacy-preserving data outsourcing protocols that offer quantifiable privacy guarantees.

1.1 Simple Encryption and Access Pattern Leakage

A clear first step in protecting the privacy of outsourced data is to encrypt all data using keys known only to the client before storing it on the server. We may then treat the server as a simple block store, decrypting data after downloading and encrypting it before uploading. We may also want to allow the server to directly support more interesting query types, such as range queries. In such cases, we must also encrypt the queries and enable the server to satisfy queries *homomorphically* (without decryption) to prevent the server from learning the contents of the query or of the returned data. If the server returns only those encrypted records needed to satisfy each query, the protocol is known as a *Precise Query Protocol* (PQP) [88].

In Chapter 2, we present a general attack against any PQP that supports range queries over a single query attribute, regardless of the specific mechanism used to enable querying. We demonstrate that the access patterns created by the encrypted range queries rapidly leak information about the ordering of encrypted records along the plaintext query attribute. We then show how an attacker can use such ordering information to efficiently reconstruct the possible locations of each record in the true ordering, and thereby infer plaintext attribute values.

In Chapter 3, we investigate a suite of PQPs that use Shamir's Secret Sharing Algorithm [70] to enable range queries. We expose flaws in these protocols' security claims, and show how to efficiently recover all plaintext values when the protocols are

assumed secure. The ordering attack presented in Chapter 2 plays a key role in our attack against the secret sharing protocols.

The attacks from Chapters 2 and 3 illustrate the need for more secure outsourcing protocols that protect access pattern privacy. Various protocols have been proposed that try to mask access patterns by injecting dummy blocks, creating cover searches that retrieve decoy blocks, or mixing and re-encrypting blocks on the client before returning them to the server [30, 61]. Unfortunately, though these schemes offer some bounds on access pattern information leaked during certain classes of attacks, they ignore other attacks and fail to quantify or bound total information leakage. We advocate protocols based on the stronger privacy models of Private Information Retrieval, Oblivious RAM, and Tunably-Oblivious Memory discussed below.

1.2 Oblivious RAM and Private Information Retrieval

Private Information Retrieval (PIR) [19] and *Oblivious RAM* (ORAM) [37] protocols both offer provable access pattern privacy for outsourced data, guaranteeing that observed access patterns leak no information about client data. Each model has its own advantages and disadvantages.

In PIR, the client issues an encrypted query for a particular bit or block of B bits of data stored on the server. The server evaluates each query homomorphically, returning the desired block without learning anything about which block was requested. In order to achieve this degree of security, the server must evaluate each query over all bits in the database, making PIR computationally prohibitive for most full-database applications [74].

In ORAM, the server acts as a simple block store. Data blocks are encrypted by the client before being stored on the server. Informally, the ORAM defines a protocol that dictates how the client should download, permute (shuffle), re-encrypt, and upload blocks in order to prevent the server from learning any information about the client's sequence of plaintext block accesses. ORAM guarantees that the observed access patterns for any two same-length sequences of block requests are computationally indistinguishable to all observers other than the client [77]. Equivalently, the output of a simulator that has no access to any of the secret information (block contents and requested block addresses) should be able to produce a sequence of encrypted block transfers that is indistinguishable from that of the actual ORAM [49].

ORAM requires negligible computation, but may incur substantial bandwidth costs or storage overheads on the client or server. Even the most bandwidth-efficient existing ORAM [76] requires roughly $\log_2 N$ block transfers per block request, where N is the number of blocks. Since most of these transfers are performed before or immediately after each request is satisfied, response times for individual requests can be high, especially when bursts of requests arrive together.

In Chapter 4 we propose Burst ORAM, the first ORAM technique that seeks to minimize response times during a burst of requests. Burst ORAM classifies block transfers as either *online*, those needed before a request can be satisfied, or *offline*, those that can be performed after the request is satisfied. Burst ORAM reduces the number of *online* block transfers from $O(\log N)$ to $O(1)$, and uses available client space to delay *offline* transfers as long as possible, ideally until an idle period. On bursty workloads, Burst ORAM achieves near-optimal response times that are orders of magnitude lower than those of existing protocols, while incurring only slightly higher total bandwidth cost than the most bandwidth-efficient prior ORAM [76].

In Chapter 5, we propose a new ORAM called OS+PIR that achieves substantially lower total bandwidth costs than any existing ORAM. OS+PIR combines PIR with the ObliviStore ORAM from [76], and permits tradeoffs between bandwidth costs, client/server storage, and computation costs. For data block counts (N) ranging from 2^{20} to 2^{30}, OS+PIR achieves a total bandwidth cost of only 11 to 13 blocks transferred per request, down from the 18 to 27 of [76]. As N grows, OS+PIR's bandwidth cost grows more slowly than that of [76], providing enhanced savings for larger databases.

1.3 Tunably-Oblivious Memory

ORAM protocols pay a high price in terms of bandwidth, computation, or client/server storage costs in order to achieve full access pattern indistinguishability. In Chapter 6 we propose Tunably-Oblivious Memory (TOM), a new privacy model that relaxes and generalizes ORAM, enabling controlled privacy/efficiency tradeoffs. Unlike ORAM, TOM permits the lengths of publicly visible access patterns to vary, allowing properties such as locality to be exploited in order to improve efficiency. For each query, a λ-TOM generates an access pattern with one of λ pre-determined lengths, limiting information leaked per query to $\log_2 \lambda$ bits. λ-TOM protocols with large λ leak more information and are more efficient, while those with small λ are more expensive and offer better privacy. 1-TOM leaks no information and is as secure as ORAM.

We also introduce a novel, special-purpose TOM called *Staggered-Bin TOM* (SBT). We prove that SBT achieves bandwidth cost $O(\log N/\log\log N)$ for large queries with blocks chosen uniformly at random, but has worst-case cost $O(\sqrt{N})$. We also propose a read-only SBT variant called Multi-SBT, which combines SBT with an ORAM, storing each block in triplicate. The Multi-SBT achieves bandwidth cost

$O(1)$ for large uniform random block queries, and $O(\log N)$ in the worst case, and leaks at most $O(\log \log \log N)$ bits of information per query. Multi-SBT achieves practical bandwidth costs as low as 6 blocks transferred per request for queries of $4\sqrt{N}$ requests.

Chapter 2

Compromising Privacy in Precise Query Protocols

2.1 Introduction

Cloud computing is a popular paradigm that lets clients outsource data management, but many users still keep sensitive data offline due to privacy concerns [20]. In the *Database As a Service* model [40], clients store, update, and query data on *honest-but-curious* cloud servers. Such servers correctly process queries, but do not respect user data privacy. Data stored on the server are encrypted using keys known only to the client (see Figure 2.1). Queries are likewise encrypted, and may be issued only by the client. The server is the principal threat to privacy, having access to all encrypted records, queries, responses, and processing, so we treat the server as the primary attacker.

Fully secure outsourced databases require a full database scan for each query [47]. To avoid such costs, many schemes [8, 12, 17, 26, 48, 51, 72] adopt the more practical *Precise Query Protocol* (PQP) model defined in [88]. In PQPs, records are individually encrypted and stored on the server as *etuples*. The protocol is *precise* since the server

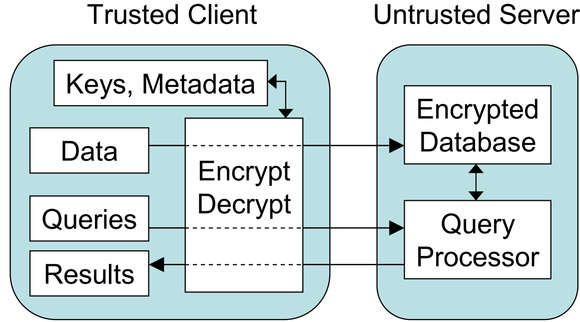

Figure 2.1: *Database As a Service* with encrypted data, queries, and results.

returns the exact set of etuples needed to satisfy each query. PQPs are efficient as they return no spurious etuples, but their precision causes them to leak information that we can use to order the stored etuples.

We propose a novel attack on the privacy of all PQPs that support one-dimensional range queries over a *query-attribute* Q. Our attack identifies a set $\mathcal{P}$ of *permissible etuple permutations*, which are potentially-correct orderings of etuples by plaintext Q value. The permissible permutations in $\mathcal{P}$ define a partial ordering of etuples. We use the precise results returned by successive range queries to exclude permutations from $\mathcal{P}$, refining this partial ordering. Existing attacks that seek to determine such etuple orderings rely on properties specific to particular PQPs [51].

Every permutation $\pi \in \mathcal{P}$ places etuple e at some position $\pi[e]$. The set of positions for e over all $\pi \in \mathcal{P}$ yields the *permissible loci* of e. If we know the permissible loci, we can make inferences about the plaintext Q value in e. As more range queries are run, more permutations are excluded from $\mathcal{P}$, and the number of permissible loci for each etuple drops, improving inferences and further compromising privacy.

Current literature recognizes that revealing etuple order is a privacy threat. The Order-Preserving Encryption Scheme (OPES) [8], an efficient PQP that explicitly reveals etuple order, notes that privacy will be compromised if the distribution of the

query-attribute is known. It is argued in [26] that schemes such as OPES should be avoided, as they allow etuples to be ordered easily. Work in [51] shows how to infer etuple order in PQPs that use prefix-preserving encryption schemes, and claims that privacy is compromised.

As in [43, 51], our goal is to enable the discovery of sensitive information associated with etuples, not necessarily to associate etuples with particular people. PQPs claim to obscure sensitive values, so revealing them clearly defeats PQP privacy guarantees. Information correlated with identity may be inferred through other attacks, or even stored in the clear.

In the example PQP database of Figure 2.2, etuples are employee records and range queries are issued on the salary attribute Q. Initially, we know nothing about which salary matches which etuple, so all 4 loci are permissible for each etuple. However, if we can use query result sets to exclude all permutations that assign Jim's etuple to loci 2 and 3, then only loci 1 and 4 are permissible. Consequently, we know that Jim's salary is either 25000 or 70000.

It is common to assume that attribute distributions are known [8, 16, 26, 43, 51], but our attack does not require exact knowledge of Q values or distributions. Even attribute distributions estimated from other sources, such as public Census Bureau data, suffice once we reduce the number of permissible loci sufficiently. Permissible loci can also be used in a larger attack to recover exact Q values [27].

2.1.1 Our Contributions

We present a novel attack on the privacy of all PQPs that support one-dimensional range queries. Our work is the first to show that etuples can be efficiently

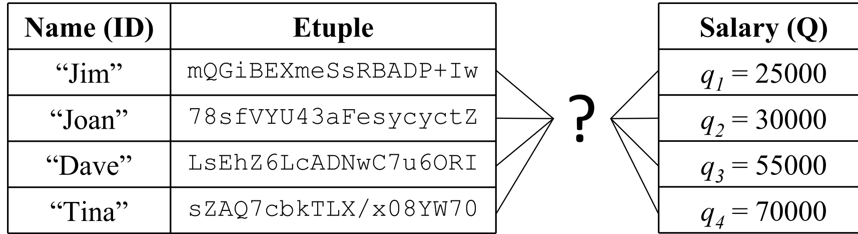

Figure 2.2: Employee records in a PQP. Salaries are known, but not employee-salary relationships.

ordered in *any* such PQP. Existing attacks exploit weaknesses specific to individual PQPs, but the only requirement for our attack is the ability to observe the etuples returned by encrypted queries. In Section 2.3, we outline our attack and show how to use PQ-trees [13] to efficiently maintain the set of permissible permutations. Our core contribution, given in Section 2.4, is a novel algorithm that uses a PQ-tree to identify the permissible loci of each etuple. Query-attribute values can be inferred from these permissible loci. In Section 2.5, we define *equivocation* and *permutation entropy* as metrics for PQP privacy. Section 2.6 gives experimental results on real and synthetic datasets, showing that privacy is compromised quickly and that PQPs are highly insecure against our attack.

2.2 Related Work

Much work exists on applying the Database As a Service model to various query types [8, 26, 40, 43, 51, 52, 60, 72]. See [69] for a survey.

2.2.1 Precise Query Protocol Schemes

PQPs are used for their efficiency. Table 2.1 shows the asymptotic costs of several PQPs. The Order Preserving Encryption Scheme (OPES) of [8] maps plaintexts to ciphertexts while preserving plaintext order and flattening the ciphertext distribution.

Table 2.1: Asymptotic costs for existing PQP schemes, with large costs highlighted. Few papers gave these costs explicitly, so the table reflects our best-effort analysis. $|D|$ is domain size, n is the etuple count, C is the result set size, N is a 512–4096 bit number, K_g and K_s are costs of group and symmetric encryption operations, respectively. δ for OPES is small with unknown relation to n.

PQP Scheme	**Scheme Setup Work**	**Client Query Pre-Process Work**	**Client Storage (Bits)**	**Server Query Work**	**Query Send Size (Bits)**
Order-Pres. Enc. (OPES) [8]	$O((K_s + \delta)n)$	$O(\delta)$	$O(\delta \log \|D\| + \log N)$	$O(\|C\| + \log n)$	$O(\log \|D\|)$
Prefix-Pres. Enc. [51]	$O(K_s n \log \|D\|)$	$O(K_s \log^2 \|D\|)$	$O(\log N)$	$O(\|C\| + \log n \log \|D\|)$	$O(\log^2 \|D\|)$
Encrypted B^+-Tree [26]	$O(K_s n)$	$O(K_s(\|C\| + \log n))$	$O(\log N)$	$O(\|C\| + \log n)$	$O((\|C\| + \log n) \log N)$
1D MRQED [72]	$O(K_g n \log \|D\|)$	$O(K_g \log \|D\|)$	$O(\log \|D\| \cdot \log N)$	$O(K_g n \log \|D\|)$	$O(\log \|D\| \cdot \log N)$
Hidden Vector Enc. (HVE) [12]	$O(K_g n \|D\|)$	$O(K_g)$	$O(\|D\| \log N)$	$O(K_g n)$	$O(\log N)$
HVE with Predicate Enc. [48, 71]	$O(K_g n \|D\|)$	$O(K_g \|D\|)$	$O(\|D\| \log N)$	$O(K_g n \|D\|)$	$O(\|D\| \log N)$

Indexes can easily be created on the ciphertext, keys are small and nearly constant in size, and encryption costs are low. However, OPES fully reveals etuple order, making it highly vulnerable to inferences.

Prefix-preserving encryption [51] encrypts the query-attribute of each record such that if two plaintexts share a k-bit prefix, their ciphertexts share a (distinct) k-bit prefix. The prefix-preserving ciphertexts cause this scheme to leak information about etuple order more rapidly than other PQPs. PQPs such as the encrypted B^+-tree in [26] process queries interactively between client and server, but suffer from heavy communication loads.

Some PQPs, including MRQED [72], RASP [17], and Hidden Vector Encryption (HVE) [12], use novel encryption techniques to improve privacy. Recent work [48, 71] has used inner-product predicate encryption to implement HVE and to initially guarantee privacy of query-attribute values and query ranges. However, when the scheme is used to support one-dimensional range queries, we can still infer etuple order using our attack.

Trusted server-side hardware is used in [47] to process queries and re-encrypt etuples in order to limit the attacker's ability to make inferences. Oblivious index traversal techniques [53] are used to maintain privacy for point queries when PQPs use indexes that are visible to the server.

2.2.2 Imprecise Query Protocols

Many schemes sacrifice query result set precision in favor of improved privacy. In *bucketization* [40, 43] the server returns all etuples in a range of *buckets*, yielding a superset of the query result. Larger buckets improve privacy, but return more spurious etuples, raising client-side costs.

Other schemes rely on data *fragmentation*, assuming that some attributes are only sensitive when paired with others [23]. Such schemes assume non-colluding servers, high client storage capacities, or obscured table relations.

Work in [30] uses an encrypted B^+-tree and incorporates spurious queries, client-side caching, and node content shuffling to provide strong query access pattern privacy. Other schemes with similar goals are based on Oblivious RAM (ORAM) [73] or Private Information Retrieval (PIR) [63]. Such techniques are becoming more efficient, but still require several communication rounds per query. Work in [52] uses hierarchical predicate encryption to achieve access pattern privacy, but depends on non-colluding proxies and only supports restricted query ranges.

2.2.3 Prior Work on Privacy Loss

Even when the privacy of individual records is guaranteed, privacy can be compromised by careful analysis of query access patterns or indexes [82, 88]. Work

in [43] discusses privacy factors of *bucketization* in terms of statistical measures such as variance and entropy, demonstrating a tradeoff between privacy and efficiency.

Using relationships between encrypted records to infer plaintext information is referred to as *inference exposure* in [26]. A common technique is to exploit the fact that identical plaintexts generally produce identical ciphertexts [16, 26]. Other attacks associate frequently requested etuples with significant plaintexts, yielding probabilistic assignments of values to etuples [30].

Authors in [51] propose an attack against PQPs that use prefix-preserving encryption to support range queries. Their attack collects all pairs of etuples known to be adjacent and uses them to infer order. Our attack leads to stronger inferences as it uses everything that can be learned about etuple order from the range query results, not just what can be inferred from adjacent pairs. Further, our attack applies generally to all PQPs that support range queries.

2.3 Attack Model and Outline

Our attack identifies etuple orderings that are consistent with observed range query result sets. We call these orderings *permissible permutations*, and store them using a PQ-tree [13]. In Section 2.4, we use the PQ-tree to identify the permissible loci of each etuple, which we use to make inferences about the etuple's query-attribute values (Section 2.1). Notation is summarized in Table 2.2.

2.3.1 Precise Query Protocols (PQPs)

In a PQP, each plaintext record r is a tuple of attributes. The data owner, or *client*, encrypts each record as a single ciphertext e called an *etuple*. Let r_e denote the

Table 2.2: PQP and Attack Notation

e, E	Encrypted record (etuple), all etuples
r_e	Plaintext record encrypted to form e
$Q, r_e.q$	Query-attribute Q, Q value of record r_e
$[\alpha, \beta]$	Inclusive plaintext range query bounds
$E_{\alpha,\beta}$	Etuples needed to satisfy query range $[\alpha, \beta]$
π_c	Correct ordering of etuples according to Q
C	Cluster of etuples, defined by a result $E_{\alpha,\beta}$
$\mathcal{P}$	Set of permissible permutations of E
Λ_e	Permissible loci of e

plaintext record that produced etuple e. The set E of all etuples is then stored on a semi-trusted, honest-but-curious server (see Section 2.1).

The client generates range queries over a *query-attribute* Q, encrypting the plaintext range $[\alpha, \beta]$ in each query. We let $r_e.q$ denote the Q value in record r_e. The server, without decrypting the query or any of the etuples, finds and returns the exact result set $E_{\alpha,\beta}$ satisfying the query, where:

$$E_{\alpha,\beta} = \{e \in E \mid \alpha \leq r_e.q \leq \beta\}$$

We can realize such server-oblivious querying protocols using specialized encryption techniques, ranging from order-preserving encryption to predicate encryption. We do not present details here, but see [8, 12, 17, 26, 48, 51, 72] for examples of PQPs supporting range queries.

Queries are encrypted, so they can only be generated by trusted clients, and an attacker cannot craft his own queries. Instead, he can mount the attack by observing responses to the queries issued by the client, without knowing the plaintext query ranges. Etuples may be inserted/deleted, so we let E be the subset of etuples that persist in the database across the set of issued queries. We exclude inserted/deleted etuples before mounting the attack.

2.3.2 Permissible Permutations and Loci

Definition 1. A *correct ordering* of E by attribute Q is a permutation $\pi = e_1 e_2 \ldots$ of E with $r_{e_1}.q \leq r_{e_2}.q \leq \cdots$.

If several etuples have the same Q value, there are multiple correct orderings. Our attack handles this case, and both our experimental datasets include repeated values. However, for ease of presentation, we introduce the attack as though only one correct ordering, π_c, exists. We can use *clusters* to learn which permutations might be π_c.

Definition 2. A *cluster* $C \subseteq E$ is a subset of etuples that are contiguous in π_c. $\mathcal{C}$ denotes a set of clusters.

Let $E_{\alpha,\beta} \subseteq E$ be the set of etuples returned by a query on any range $[\alpha, \beta]$. Since $E_{\alpha,\beta}$ contains precisely those $e \in E$ for which $\alpha \leq r_e.q \leq \beta$, etuples in $E_{\alpha,\beta}$ are contiguous in π_c. Thus, each query result set $E_{\alpha,\beta}$ is a cluster.

Definition 3. A cluster C *excludes* a permutation π of E if an etuple $e_i \notin C$ appears in π between two $e_j, e_k \in C$. Once π has been excluded, we know that $\pi \neq \pi_c$.

Definition 4. The set $\mathcal{P} = \{\pi_1, \pi_2, \ldots\}$ of *permissible permutations* consists of all permutations of E not excluded by any cluster. A permutation π is permissible if and only if for every cluster C, the etuples in C are contiguous in π.

Each permutation in $\mathcal{P}$ is potentially the correct ordering, given the observed clusters, so $\mathcal{P}$ defines a partial ordering on E. Initially, every permutation is in $\mathcal{P}$. As queries arrive, we can identify clusters and use them to exclude from $\mathcal{P}$ any permutations in which clustered etuples are not contiguous, thereby refining the partial ordering.

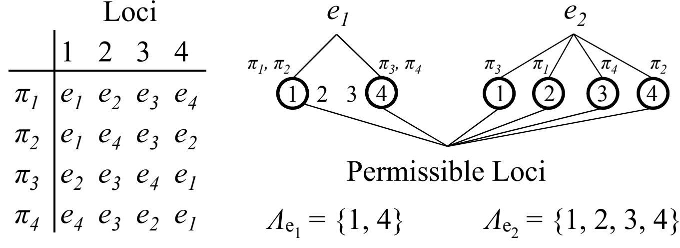

Figure 2.3: *Permissible loci* of etuples e_1 and e_2, given permissible permutations $\mathcal{P} = \{\pi_1, \pi_2, \pi_3, \pi_4\}$.

Consider the etuple set $E = \{I, J, K\}$. Initially, all six permutations of E are permissible. A range query returning J and K defines a cluster $C = \{J, K\}$. Permutations JIK and KIJ are excluded by C, since $I \notin C$ appears between $J, K \in C$. Only four permutations remain permissible.

Definition 5. The *locus* $\ell = \pi[e]$ of etuple e in permutation π is the position of e in π, such that e is the ℓth etuple in π. The *permissible loci* Λ_e of e are the set of loci of e across all permissible permutations $\pi \in \mathcal{P}$ (see Figure 2.3):

$$\Lambda_e = \{\pi[e] \mid \pi \in \mathcal{P}\}$$

Λ_e gives possible positions of e in the correct ordering π_c. Etuples in π_c are ordered by query-attribute value, so if we have even partial knowledge of the query-attribute distribution, we can use Λ_e to make inferences about the value of $r_e.q$. As more clusters are observed, more permutations are excluded from $\mathcal{P}$, lowering $|\Lambda_e|$ and reducing uncertainty about $r_e.q$.

A cluster that excludes any permutation π must also exclude its reverse $\overline{\pi}$. Thus, given enough distinct clusters, we can exclude permutations until $\mathcal{P} = \{\pi_c, \overline{\pi_c}\}$, but no further. Therefore, we always have $|\mathcal{P}| \geq 2$.

2.3.3 Using PQ-Trees to Maintain $\mathcal{P}$

A *PQ-tree* [13] is a rooted, ordered tree capable of compactly representing the set of permissible permutations $\mathcal{P}$ derived from cluster set $\mathcal{C}$. In fact, PQ-trees were designed specifically to represent such permutations. Non-leaf nodes in PQ-trees are of type P or Q. Leaves represent etuples from E, so the terms *etuple* and *leaf* will be used interchangeably. The sequence of leaves reached by a pre-order traversal of a PQ-tree T forms its *frontier* $F(T)$, which defines a permutation of E.

Every P-node and Q-node must have at least two children. Children of a P-node can be arbitrarily permuted, while children of a Q-node may only be reversed. Each combination of rearranged children in T produces an *equivalent* tree $T' \equiv T$. The set $\mathcal{P}_T$ of permutations consistent with T is $\mathcal{P}_T = \{F(T') \mid T' \equiv T\}$.

All PQ-tree leaves start as children of a single P-node, forming a *universal* tree consistent with all $|E|!$ possible permutations. Using a cluster C, we can transform T into a new PQ-tree T^* through a *reduction* operation. The permutations consistent with T^* are those that are consistent with T *and* in which all etuples in C are contiguous. The reduction algorithm runs in time $O(|C|)$ [13].

After successively reducing T using each $C \in \mathcal{C}$, T precisely represents $\mathcal{P}$, giving $\mathcal{P}_T = \mathcal{P}$. With enough distinct clusters, we can reduce T until all its leaves are children of a single Q-node, at which point $\mathcal{P}_T = \{\pi_c, \overline{\pi_c}\}$.

Let $n = |E|$ be the number of leaves (etuples) in T. Since every non-leaf node in T has at least two children, the number of nodes m in the tree is at most $2n - 1$. The height of T ranges from 1 when all etuples are children of a single P or Q-node, to $n - 1$, such as when T is left-deep.

2.3.4 Characteristic Examples

We now work through a few simple examples to demonstrate our attack. In each example, we apply a set of clusters $\mathcal{C}$ to the etuple set $E = \{I, J, K, L, M, N\}$, letting $\pi_c = IJKLMN$. We then describe the set of permissible permutations $\mathcal{P}$ consistent with $\mathcal{C}$, show the corresponding PQ-tree, and identify the permissible loci of several etuples. PQ-tree diagrams represent P-nodes using circles and Q-nodes using rectangles. Etuples are represented by their labels $I \cdots N$.

By Definition 4, a permutation π is in $\mathcal{P}$ if and only if for every cluster $C \in \mathcal{C}$, etuples in C are contiguous in π. It is helpful to think of each etuple as a point on a line, where point e has value $r_e.q$. Etuples in a cluster may appear in any order, as long as they are together on the line and have no other etuples between them.

As more clusters are added, the PQ-tree's structure can become quite complex, so we cannot provide examples for all cases here. For a more thorough demonstration of PQ-trees, see [13]. If $\mathcal{C} = \emptyset$, all permutations are permissible. In this case, all etuples are children of a single P node, and every etuple has all 6 possible permissible loci.

Example 1. *(Disjoint Clusters)* Let $C_1 = \{I, J, K\}$, and let $C_2 = \{L, M\}$, as in Figure 2.4. Permutations in $\mathcal{P}$ must not intersperse etuples in C_1, C_2, and $\{N\}$. However, the sets themselves and the elements within each set may appear in any relative order. There are 3! ways to permute $\{C_1, C_2, \{N\}\}$, 3! ways to permute C_1, 2 for C_2, and 1 for $\{N\}$. The PQ-tree representing $\mathcal{P}$ is given in Figure 2.5. For each $e \in E$, we can still find a $\pi \in \mathcal{P}$ that assigns e to any one of the 6 loci, so all etuples have all 6 permissible loci.

Figure 2.4: Permutation π_1 is permitted but π_2 is excluded as C_1 is split by $N \notin C_1$ in π_2.

Figure 2.5: PQ-tree with frontier $LMNKIJ$ reduced using clusters $\{I, J, K\}$, $\{L, M\}$.

Figure 2.6: Permutation π_1 is permitted but π_2 is excluded as C_2 is split by $I, J \notin C_2$ in π_2.

Figure 2.7: PQ-tree with frontier $NMKLJI$ reduced using $\{I, J, K, L\}$, $\{K, L, M, N\}$.

Example 2. *(Cluster Overlap)* Let $C_1 = \{I, J, K, L\}$ and $C_2 = \{K, L, M, N\}$ as in Figure 2.6. We define sets $S_1 = \{I, J\}$, $S_2 = \{K, L\} = C_1 \cap C_2$, and $S_3 = \{M, N\}$. Considering etuples as points on a line, we know that etuples in C_1 appear together, as do etuples in C_2. To meet both of these conditions, etuples in $C_1 \cap C_2$ must together fall between the remaining etuples. That is, etuples in S_2 must be together, and must separate the etuples in S_1 from those in S_3.

Thus, S_1, S_2, and S_3 form clusters of their own, and they appear in order $S_1S_2S_3$ or $S_3S_2S_1$. We have 2 ways to permute elements within each of S_1, S_2, and S_3. The PQ-tree is given in Figure 2.7, where we use a Q node to represent the fact that the sets S_1, S_2, S_3 can be in one of two orders. Etuples I, J, M, N are restricted to permissible loci $\Lambda_e = \{1, 2, 5, 6\}$, while K and L are restricted to $\Lambda_e = \{3, 4\}$.

Example 3. *(All 2-Clusters)* Figure 2.8 shows all possible size-two clusters. Applying arguments from Example 2 to each pair of intersecting clusters, we get $\mathcal{P} = \{\pi_c, \overline{\pi_c}\}$.

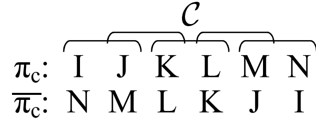

Figure 2.8: The clusters in $\mathcal{C}$ permit *only* π_c and $\overline{\pi_c}$.

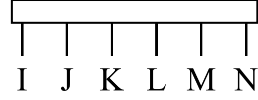

Figure 2.9: PQ-tree fully reduced using all size-2 clusters.

The PQ-tree is given in Figure 2.9. Each etuple has 2 permissible loci. I and N each have $\Lambda_e = \{1, 6\}$, J and M have $\Lambda_e = \{2, 5\}$, and K and N have $\Lambda_e = \{3, 4\}$.

2.4 Identifying Permissible Loci

We give a novel algorithm for identifying permissible loci Λ_e of every etuple $e \in E$ given a PQ-tree T. The algorithm runs in time $O(n^2 \log n)$ and requires $O(n \log n)$ space, where $n = |E|$, and includes a dynamic programming solution for a series of related subset sum problems that we must solve for each P node. Partial results for each solution are cached in a depth-first manner to exploit problem similarities. We also give a variation called *κ-pruning*, which finds Λ_e for etuples with $|\Lambda_e| < \kappa$ in $O(n\kappa \log \kappa)$ time and $O(\kappa \log n)$ space. Key notation is summarized in Table 2.3.

2.4.1 Algorithm Outline and Terminology

Let y be a node in PQ-tree T. Our goal is to identify the permissible loci of each etuple (leaf) in T.

Definition 6. The *etuple descendants* Δ_y of y are the etuples (leaves) descended from y in T. If y is a leaf, $\Delta_y = \{y\}$.

Definition 7. The *spread* σ_y is the number of etuple descendants of y, $\sigma_y = |\Delta_y|$. We can pre-compute spreads for all nodes in time $O(n)$.

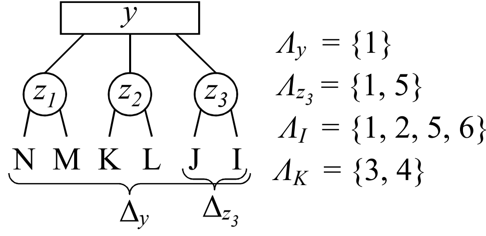

Figure 2.10: Labeled PQ-tree (Example 4).

Definition 8. The symbol η represents an *offset*, which denotes the total number of etuples descended from all siblings that precede a given node in a PQ-tree.

Definition 9. The *locus* of a node y in PQ-tree T' is the position in the frontier $F(T')$ at which etuples in Δ_y begin to appear. The *permissible loci* Λ_y of y are the set of all such loci of y across all $T' \equiv T$. If y is the root, $\Lambda_y = \{1\}$. Since each such frontier is a permissible permutation of etuples (Section 2.3.3), this definition generalizes Definition 5 from etuples to any node. For brevity, we often refer to Λ_y as simply the *loci* of y.

Let $z_1, \ldots, z_c$ be the $c \geq 0$ children of y. The locus of z_i in T' is offset from the locus of y by the spreads of all children of y that precede z_i in T'. The subsets of children that may precede z_i in any tree $T' \equiv T$ are determined by y's node type (P or Q). Thus, given Λ_y, y's type, and the spreads $\sigma_{z_1}, \ldots, \sigma_{z_c}$ of each of y's children, we can identify $\Lambda_{z_1}, \ldots, \Lambda_{z_c}$. We apply this technique recursively to identify the permissible loci of all nodes in T, including its leaves.

Example 4. In Figure 2.10, y is the root of a PQ-tree, so no leaves can precede Δ_y, and $\Lambda_y = \{1\}$. Since y is a Q node, its children can be reversed, so exactly 0 or 4 of the leaves in Δ_y must precede Δ_{z_3}. Thus, the permissible loci Λ_{z_3} are offset from Λ_y by $\eta_1 = 0$ and $\eta_2 = 4$, giving $\Lambda_{z_3} = \{1, 5\}$. Similarly, $\eta_3 = 0$ or $\eta_4 = 1$ of the leaves in Δ_{z_3} precede Δ_I for each locus in Λ_{z_3}, so $\Lambda_I = \{1, 2, 5, 6\}$.

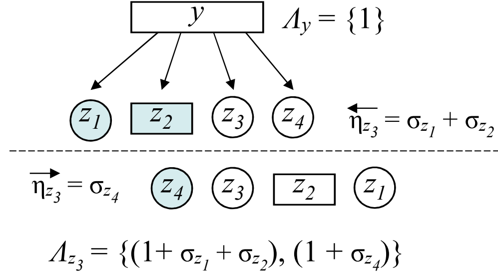

Figure 2.11: The loci Λ_{z_3} are offset from Λ_y by $\overleftarrow{\eta_{z_3}}$ when y's children are ordered left-to-right, and by $\overrightarrow{\eta_{z_3}}$ when ordered right-to-left.

2.4.2 Identifying Loci for Children of Q-Nodes

Let y be a Q-node in T. We let $\overleftarrow{\eta_{z_i}}$ be the number of etuples in Δ_y that precede Δ_{z_i} in T, and $\overrightarrow{\eta_{z_i}}$ the number that precede Δ_{z_i} in the equivalent tree where y's children are reversed. Thus, $\overleftarrow{\eta_{z_i}} = \sum_{j=1}^{i-1} \sigma_{z_j}$ and $\overrightarrow{\eta_{z_i}} = \sum_{j=i+1}^{c} \sigma_{z_j}$. Λ_{z_i} are offset from Λ_y by either $\overleftarrow{\eta_{z_i}}$ or $\overrightarrow{\eta_{z_i}}$, as in Figure 2.11:

$$\Lambda_{z_i} = \{\ell + \overleftarrow{\eta_{z_i}} \mid \ell \in \Lambda_y\} \cup \{\ell + \overrightarrow{\eta_{z_i}} \mid \ell \in \Lambda_y\} \tag{2.1}$$

We call this addition/union operation an *expansion* of Λ_y to Λ_{z_i}, since $|\Lambda_{z_i}| \geq |\Lambda_y|$. Since $|\Lambda_y| \in O(n)$ for all $y \in T$, each expansion takes $O(n)$ time. We perform one such expansion per child, so the per-child cost is $O(n)$. Pseudocode is given in Algorithm 1, Lines 12–23. For space efficiency, we perform the expansion for the child with the largest spread last (see Section 2.4.4).

2.4.3 Identifying Loci for Children of P-Nodes

If y is a P-node in T, any permutation of y's children $z_1, \ldots, z_c$ yields an equivalent PQ-tree. Since a child z_i may be preceded by any subset of the other $c-1$ children, the number of possible offsets for the loci Λ_{z_i} from Λ_y may be large. In contrast,

y $\Lambda_y = \{1\}$

z_1 z_2 z_3 z_4 $\eta_1 = \sigma_{z_1} + \sigma_{z_2}$

$\eta_2 = 0$ z_3 z_1 z_2 z_4

$\eta_3 = \sigma_{z_1}$ z_1 z_3 z_2 z_4

$\vdots$

z_2 z_4 z_3 z_1 $\eta_7 = \sigma_{z_2} + \sigma_{z_4}$

z_1 z_2 z_4 z_3 $\eta_8 = \sigma_{z_1} + \sigma_{z_2} + \sigma_{z_4}$

$$\Lambda_{z_3} = \{(1+\sigma_{z_1}+\sigma_{z_2}), (1), (1+\sigma_{z_1}), \ldots, (1+\sigma_{z_2}+\sigma_{z_4}), (1+\sigma_{z_1}+\sigma_{z_2}+\sigma_{z_4})\}$$

Figure 2.12: For P-node y, each subset of children to the left of z_3 yield an offset η of Λ_{z_3} from Λ_y.

Table 2.3: Notation for Node y of PQ-Tree T

E, n	Set/number of etuples (leaves) in T
m	Total number of nodes (P, Q, and leaf) in T
$\mathcal{P}_T$	Set of permutations of E consistent with T
C	*Cluster* of etuples defined by a query result
Δ_y	The set of etuples descended from y
σ_y	The *spread* of y, $\sigma_y \equiv \lvert\Delta_y\rvert$
Λ_y	The *permissible loci* of y (starting loci of Δ_y)
Φ_y	A set of *intermediate loci* used in computation
η	An *offset*; a number of etuples preceding a node

Q-node children may only be reversed, so there are at most two offsets for each child. Thus, computing loci for P-node children is more challenging.

As Figure 2.12 shows, the sum of spreads of every possible subset of the other $c-1$ children is a valid offset η of Λ_{z_i} from Λ_y. To test whether a given value of η is an offset, we must solve a *subset-sum* problem, where the target sum is η and the list of integers is the multiset $\{\sigma_{z_j} \mid j \neq i\}$ of the spreads of the $c-1$ children.

2.4.3.1 Each Child Considered Separately

The general subset-sum problem is NP-complete, but we know that the sum of all child spreads never exceeds n. Using this fact, $\sum_{j=1}^{c} \sigma_{z_j} \leq n$, we can compute all the offsets of any Λ_{z_i} from Λ_y in time $O(nc)$ using the standard pseudo-polynomial-time dynamic programming algorithm for enumerating subset sums [32]. The dynamic program is based on the insight that we can choose each child $z_j, j \neq i$ to precede or succeed z_i *independently.* Thus, when computing subset sums, we can independently add in or not add in each spread $\sigma_{z_j}, j \neq i$, in any order.

Once we compute the possible offsets, we must add them to the loci in Λ_y as we did for Q-nodes in Equation 2.1. We can combine both steps by initializing the algorithm with Λ_y, yielding the following recurrence for Λ_{z_i}, where Φ_j represents the intermediate loci set obtained after considering the first j children:

$$\Lambda_{z_i} = \Phi_c, \text{ where} \qquad (2.2)$$
$$\Phi_j = \begin{cases} \Lambda_y, & j = 0 \\ \Phi_{j-1}, & j = i \\ \Phi_{j-1} \cup \{\ell + \sigma_{z_j} \mid \ell \in \Phi_{j-1}\}, & \textit{otherwise} \end{cases}$$

We make no change to the intermediate loci when $j = i$, since z_i cannot precede itself. As in Equation 2.1, we call each addition/union operation an *expansion* with $O(n)$ cost. Identifying Λ_{z_i} for a single child z_i using Equation 2.2 requires $c - 1$ expansions, so the per-child cost is $O(nc)$.

Example 5. Let y be a P-node with $\Lambda_y = \{1\}$ and 4 children as in Figure 2.12. Let $\sigma_{z_1} = 2, \sigma_{z_2} = 3, \sigma_{z_4} = 2$. We show how to find Λ_{z_3} using Equation 2.2 with $i = 3$. First, we have $\Phi_0 = \Lambda_y = \{1\}$. We expand with σ_{z_1} to get $\Phi_1 = \{1\} \cup \{1 + 2\} = \{1, 3\}$, then expand with σ_{z_2} to get $\Phi_2 = \{1, 3\} \cup \{1 + 3, 3 + 3\} = \{1, 3, 4, 6\}$. $\Phi_3 = \Phi_2$ since we skip

over σ_{z_3}, and we expand with σ_{z_4} to get $\Phi_4 = \{1,3,4,6\} \cup \{1+2, 3+2, 4+2, 6+2\} = \{1,3,4,5,6,8\}$.

2.4.3.2 All Children Considered Together

If z_i, z_k are children of y, a direct application of Equation 2.2 identifies Λ_{z_i} and Λ_{z_k} by successively expanding Λ_y with each spread in $\{\sigma_{z_j} \mid j \neq i\}$ and $\{\sigma_{z_j} \mid j \neq k\}$, respectively. Thus, both Λ_{z_i} and Λ_{z_k} require expansions that use the *shared* spreads $\{\sigma_{z_j} \mid j \neq i, k\}$. We can reduce the per-child cost of our algorithm from $O(nc)$ to $O(n \log c)$ by limiting the number of expansions performed with shared spreads.

Since y is a P-node, all child orders are legal. Thus, when we identify any Λ_{z_i} using Equation 2.2, we can change the order in which we consider spreads for expansion, as long as we skip over σ_{z_i}. By manipulating the spread order, we can avoid unnecessarily repeating expansions with shared spreads when identifying both Λ_{z_i} and Λ_{z_k}, $k \neq i$.

For example, we can first expand Λ_y using all the shared spreads $\{\sigma_{z_j} \mid j \neq i, k\}$ to get the intermediate loci set Φ_{z_i,z_k}. We then expand Φ_{z_i,z_k} using σ_{z_k} to get Λ_{z_i}, and expand Φ_{z_i,z_k} using σ_{z_i} to get Λ_{z_k}, reducing the number of expansions from $2c-2$ to c. We can apply this principle recursively to efficiently identify the loci of every child of y.

We use a depth-first divide-and-conquer approach. Expansions using the shared spreads from the second half of the children, given by $\sigma_{z_{(c/2)+1}}, \ldots, \sigma_{z_c}$, are common to identifying loci for each of the first half of the children $\Lambda_{z_1}, \ldots, \Lambda_{z_{c/2}}$, and vice-versa. Identifying loci for each fourth of the children we use spreads from the other three fourths, etc.

Example 6. Let y be a PNode with children $z_1, \ldots, z_8$, as in Figure 2.13. Let $\Phi_{z_i,\ldots,z_k}$, $i \leq k$, represent the intermediate loci after expanding Λ_y using all shared spreads com-

Algorithm 1 Identifying Λ_e for all $e \in E$ descended from node y. Children of y are $z_1, \ldots, z_c$.

1: **procedure** TraverseNode(y, Λ_y)
2:     **if** $y \in E$ **then**
3:         report $\Lambda_e = \Lambda_y$ for etuple $e = y$
4:     **else if** y is a Q-node **then**
5:         QNode(y, Λ_y)
6:     **else**
7:         sort y's children s.t. $\sigma_{z_1} \leq \sigma_{z_2} \leq \cdots \leq \sigma_{z_c}$
8:         PNode($y, \Lambda_y, [1, \ldots, c]$)
9:     **end if**
10: **end procedure**
11:
12: **procedure** QNode(y, Λ_y)
13:     $max \leftarrow$ index of child z_{max} of y with max $\sigma_{z_{max}}$
14:     **for** $i \leftarrow 1 \ldots c$ **do**
15:         **if** $i \neq max$ **then**
16:             $\Lambda_{z_i} \leftarrow$ Expand($\Lambda_y, \sum_{j=1}^{i-1} \sigma_{z_j}, \sum_{j=i+1}^{c} \sigma_{z_j}$)
17:             TraverseNode(z_i, Λ_{z_i})
18:         **end if**
19:     **end for**
20:     $\Lambda_{z_{max}} \leftarrow$ Expand($\Lambda_y, \sum_{j=1}^{max-1} \sigma_{z_j}, \sum_{j=max+1}^{c} \sigma_{z_j}$)
21:     Destroy(Λ_y)
22:     TraverseNode($z_{max}, \Lambda_{z_{max}}$)
23: **end procedure**
24:
25: **procedure** PNode(y, Φ, S)
26:     **if** $|S| = 1$ **then** TraverseNode($z_{S(1)}, \Phi$)
27:     **else**
28:         $\Phi' \leftarrow$ copy of Φ
29:         $mid \leftarrow \lfloor |S|/2 \rfloor$
30:         **for** $i \leftarrow (mid + 1) \ldots |S|$ **do**
31:             $\Phi' \leftarrow$ Expand($\Phi', 0, \sigma_{z_{S(i)}}$)
32:         **end for**
33:         PNode($y, \Phi', [S(1), \ldots, S(mid)]$)
34:         Destroy(Φ')
35:         **for** $i \leftarrow 1 \ldots mid$ **do**
36:             $\Phi \leftarrow$ Expand($\Phi, 0, \sigma_{z_{S(i)}}$)
37:         **end for**
38:         PNode($y, \Phi, [S(mid + 1), \ldots, S(|S|)]$)
39:     **end if**
40: **end procedure**
41:
42: **function** Expand(Φ, η_1, η_2)
43:     **return** $\{\ell + \eta_1 \mid \ell \in \Phi\} \cup \{\ell + \eta_2 \mid \ell \in \Phi\}$
44: **end function**

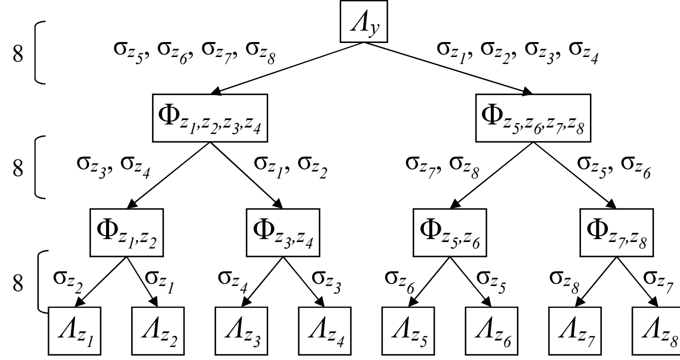

Figure 2.13: Identifying permissible loci for children of P-node y. Each arrow indicates expansions using the labeled spreads. Eight expansions are performed at each of three levels. (Example 6)

mon to identifying $\Lambda_{z_i}, \ldots, \Lambda_{z_k}$. Our goal is to identify all loci $\Lambda_{z_1}, \ldots, \Lambda_{z_8}$. We first identify and cache Φ_{z_1,z_2,z_3,z_4} by expanding Λ_y using spreads $\sigma_{z_5}, \sigma_{z_6}, \sigma_{z_7}, \sigma_{z_8}$. We then identify and cache Φ_{z_1,z_2} by expanding Φ_{z_1,z_2,z_3,z_4} using $\sigma_{z_3}, \sigma_{z_4}$. Finally, we identify Λ_{z_1} by expanding Φ_{z_1,z_2} using σ_{z_2}. We then backtrack and again expand the cached Φ_{z_1,z_2}, this time with σ_{z_1}, to get Λ_{z_2}. We backtrack again to identify Φ_{z_3,z_4} by expanding Φ_{z_1,z_2,z_3,z_4} using $\sigma_{z_1}, \sigma_{z_2}$, and so on until we identify Λ_{z_8}. In all, we perform only $8 \log_2 8 = 24$ expansions, instead of $8 \cdot (8-1) = 56$, as we would if we identified loci for each child separately.

Pseudocode is given in Algorithm 1, Lines 7, 8, and 25–40. The children are initially sorted by spread for space efficiency (see Section 2.4.4). S is a sequence of child indexes, initialized to $[1, \ldots, c]$ (Line 8). If $|S| = 1$, the intermediate loci Φ are the loci of a particular child (Line 26), and we can recursively find the loci of that child's children. Otherwise, we split S in half, and save a copy of Φ. We then expand the copy of Φ using the *second* half of the spreads $\sigma_{z_{S(\lfloor |S|/2 \rfloor+1)}}, \ldots, \sigma_{z_{S(|S|)}}$, and recursively call PNode with the expanded Φ and the first half of the spreads, given by $\sigma_{z_{S(1)}}, \ldots, \sigma_{z_{S(\lfloor |S|/2 \rfloor)}}$ (Lines 28–

33). We then repeat the process, expanding Φ using the *first* half of the spreads, and recursively calling PNode using the second half of the spreads (Lines 35–38).

The recursion has $O(\log c)$ levels, and we expand using each spread at most once for each level, so the total cost is $O(nc \log c)$, with a per-child cost of only $O(n \log c)$, down from $O(nc)$ when the loci of each child are identified separately (Section 2.4.3.1).

2.4.4 Analysis and Space-Time Tradeoff

Algorithm 1 combines our methods for P and Q-node children using a depth-first approach. Once we identify the permissible loci for a node in T, we immediately start identifying the permissible loci of its children. Algorithm 1 identifies loci for $O(n)$ nodes and generates $O(n)$ sets of intermediate loci via recursive calls to PNode, with $O(n)$ loci in each set.

There is a tradeoff between improved speed if we cache more loci and reduced space if we cache fewer. One possible extreme is to store all $O(n)$ loci sets, using $O(n^2)$ space. However, since we can easily have $n \geq 10^6$, storing $O(n^2)$ items in memory is unacceptable. (Even storing the permissible loci for every etuple takes $O(n^2)$ space, but we assume that the attacker logs any loci of long-term interest to secondary storage.)

Another extreme is to store only one set of loci at a time, and apply Equations 2.1 and 2.2 directly. Since we do not cache any intermediate loci, we must start from the root every time we want to identify Λ_e for a different e. For each node y on the path from the root to e, this algorithm takes time $O(nc)$, where y has c children. Thus, identifying Λ_e requires $O(n^2)$ time for each etuple, but only $O(n)$ total space. Repeating for all n etuples takes time $O(n^3)$.

Algorithm 1 is a compromise between these extremes. It caches intermediate loci, but discards them as soon as possible (Lines 21 and 34). By carefully structuring

our traversal of T, we can find all Λ_e in time $O(n^2 \log n)$, using only $O(n \log n)$ space. For example, we can discard the loci of a Q-node as soon as we identify the loci of its last child. Since T can have height $O(n)$, a naïve traversal still requires $O(n^2)$ space, so we must carefully select the order in which we perform expansions.

Theorem 1. *Algorithm 1 takes time $O(n^2 \log n)$.*

Proof. Per-child time is $O(n)$ for children of Q-nodes (Section 2.4.2) and $O(n \log c)$ for children of P-nodes (Section 2.4.3.2). The per-child time for sorting children of P-nodes (Line 7) is only $O(\log c)$. Thus, the worst-case per-child time in Algorithm 1 is $O(n \log c) \subset O(n \log n)$. Since T has n leaves, and each non-leaf node has at least two children, there are $O(n)$ nodes, and thus $O(n)$ children, in T. With $O(n)$ children, and time $O(n \log n)$ per child, the total time for Algorithm 1 is in $O(n^2 \log n)$. □

The parameters in Algorithm 1 are pass-by-reference. Before we prove that Algorithm 1 requires $O(n \log n)$ space, we must introduce the following definitions.

Definition 10. A *caching call* is a procedure call to QNode or PNode for which we are caching a set of loci (Λ_y in QNode, Φ' in PNode). Each call is a caching call until its cached loci are destroyed (Lines 21, 34), except for PNode calls with $|S| = 1$, which are not caching calls.

Definition 11. The *coverage* of a call is the number of etuples for which Λ_e is reported (Line 3) by the call and all its descendant calls. Thus, the coverage of a call is the sum of coverages of its sub-calls. A TraverseNode or QNode call has coverage σ_y, and a PNode call has coverage $\sum_{i=1}^{|S|} \sigma_{z_{S(i)}}$.

Lemma 2. *Calls to* QNode *and* PNode *in Algorithm 1 always make the sub-call with the largest coverage last.*

Proof. For calls to QNODE, all sub-calls are calls to TRAVERSENODE on the children $z_1, \ldots, z_c$. Thus, by Definition 11, the coverage of a sub-call for child z_i is the spread σ_{z_i}. Since Algorithm 1 explicitly makes the call for the child with the largest spread last, it makes the sub-call with the largest coverage last.

For calls to PNODE, if $|S| = 1$, the sub-call with largest coverage is the last and only sub-call. Otherwise, PNODE makes two sub-calls to PNODE. The first sub-call has coverage given by the sum $\sum_{j=1}^{\lfloor |S|/2 \rfloor} \sigma_{z_{S(j)}}$, and the second by $\sum_{j=\lfloor |S|/2 \rfloor+1}^{|S|} \sigma_{z_{S(j)}}$. The second sub-call's coverage sums at least as many spreads as the first, since $\lfloor |S|/2 \rfloor \leq |S|/2$. Further, the indexes in S are always in increasing order, and the children are sorted such that for any two children z_i, z_k, with $i < k$, we have that $\sigma_{z_i} \leq \sigma_{z_k}$ (Line 7). Thus, $\sum_{j=1}^{\lfloor |S|/2 \rfloor} \sigma_{z_{S(j)}} \leq \sum_{j=\lfloor |S|/2 \rfloor+1}^{|S|} \sigma_{z_{S(j)}}$, so the last (second) sub-call has the largest coverage. □

Lemma 3. *The coverage of each caching call is at least twice that of its active sub-call.*

Proof. Let ψ be a caching call with active sub-call λ and last sub-call ω. By Lemma 2, ω is the sub-call with the largest coverage. By Definition 10, ψ must be a call to PNODE or QNODE, and $\lambda \neq \omega$, since cached loci are destroyed before sub-call ω is made. The coverage of ψ is at least that of λ and ω combined, and since the coverage of ω is at least that of λ, the coverage of ψ is at least twice that of λ. □

Theorem 4. *Algorithm 1 requires $O(n \log n)$ space.*

Proof. Let e be a leaf for which we are reporting Λ_e (Line 3), and let χ be the number of currently cached loci sets. Since loci are only cached by a caching call, χ is also the number of caching calls in the current call stack.

The coverage of the root's TRAVERSENODE call is n. By Definition 11, the coverage of any call is at least as large as the coverage of each of its sub-calls. By

Lemma 3, the coverage of the ith deepest caching call is at least twice that of the $(i+1)$st deepest caching call. Thus, the coverage of the deepest call is at most $n/2^\chi$. The deepest call is the TRAVERSENODE call reporting Λ_e, which has coverage $\sigma_e = 1$ according to Definition 11. Thus, we have:

$$\frac{n}{2^\chi} \geq 1 \;\rightarrow\; 2^\chi \leq n \;\rightarrow\; \chi \leq \log_2 n \tag{2.3}$$

Each of the χ cached loci sets consumes $O(n)$ space, so Algorithm 1 requires $O(n \log n)$ space. □

2.4.5 The κ-Pruning Variant

Etuples with fewer permissible loci (smaller $|\Lambda_e|$) yield more information (Section 2.3.2). Thus, we may want to identify Λ_e for only those etuples with $|\Lambda_e| < \kappa$, for some threshold κ. We can prune the call tree in Algorithm 1 to find such loci in $O(\kappa \log n)$ space and $O(n\kappa \log \kappa)$ time. We refer to the resulting algorithm as the *κ-pruning* variant.

If z_i is a child of y, we expand Λ_y to get Λ_{z_i}, so $|\Lambda_{z_i}| \geq |\Lambda_y|$. Thus, if $|\Lambda_y| \geq \kappa$, all etuple descendants of y will have $|\Lambda_e| \geq \kappa$, and we can *prune* y, skipping the TRAVERSENODE calls for y and all its descendants. Since we need only traverse nodes with $|\Lambda_y| < \kappa$, we need only store loci sets with at most $O(\kappa)$ loci, so κ-pruning has space complexity $O(\kappa \log n)$. When y is a P-node, we may be able to use the following theorem to prune y even if $|\Lambda_y| < \kappa$.

Theorem 5. *If y is a P-node with children $z_1, \ldots, z_c$, then for every child z_i, $|\Lambda_{z_i}| \geq |\Lambda_y| + c - 1$.*

Proof. Let m_j be the maximum value in Φ_j in Equation 2.2. For each expansion $(j \neq i)$, σ_{z_j} is added to each element in Φ_{j-1}, including m_{j-1}, and the results are placed in Φ_j.

Thus, $m_j \geq \sigma_{z_j} + m_{j-1}$. Since $\sigma_{z_j} \geq 1$, $m_j > m_{j-1}$ and thus $m_j \notin \Phi_{j-1}$. Since $\Phi_{j-1} \subset \Phi_j$, and Φ_j has at least one element (m_j) that is not in Φ_{j-1}, we know that $|\Phi_j| \geq |\Phi_{j-1}| + 1$. We perform $c - 1$ expansions going from Φ_0 to Φ_c, so $|\Lambda_{z_i}| = |\Phi_c| \geq |\Phi_0| + (c-1) = |\Lambda_y| + (c-1)$, and thus $|\Lambda_{z_i}| \geq |\Lambda_y| + c - 1$. □

By Theorem 5, we know that if y is a P-node, and if $|\Lambda_y| + c - 1 \geq \kappa$, then for every child z_i of y, $|\Lambda_{z_i}| \geq \kappa$. Thus, we can prune y if it has at least $c \geq \kappa - |\Lambda_y| + 1$ children. Since we always have $|\Lambda_y| \geq 1$, we can always prune y if $c \geq \kappa$.

Algorithm 1 requires $O(1)$ expansions per child for a Q-node, and $O(\log c)$ per child for a P-node. Since we need only traverse P-nodes with $c < \kappa$, we need at most $O(\log \kappa)$ expansions per child. Loci sets now contain at most $O(\kappa)$ loci, so each expansion takes time $O(\kappa)$. In pathological cases, we still traverse $O(n)$ children, so the total time for κ-pruning is $O(n\kappa \log \kappa)$. In practice, κ-pruning runs much faster than this asymptotic bound.

The following theorem will be used in Section 2.6.

Theorem 6. *If at least $\kappa - 1$ etuples appear in none of the clusters in $\mathcal{C}$, then every etuple has $|\Lambda_e| \geq \kappa$, for $1 < \kappa < n$.*

Proof. Let y be the root of T, with $|\Lambda_y| = 1$. Recall that y starts out as a P-node with all etuples as its children. If $\kappa > 1$ and at least $\kappa - 1$ etuples do not appear in any cluster, then those $\kappa - 1$ etuples must still be children of y, and y must still be a P-node with at least κ children. Therefore, by Theorem 5, every child z_i of y has $|\Lambda_{z_i}| \geq |\Lambda_y| + \kappa - 1 = \kappa$. Thus, since all etuples are descendants of y, every etuple also has $|\Lambda_e| \geq \kappa$. □

2.5 Measuring Privacy Loss

As the number of permissible loci $|\Lambda_e|$ becomes smaller, more information is revealed about the query-attribute of e, reducing privacy. *Equivocation* captures this measure of progress toward compromising PQP privacy.

Definition 12. Etuple e has *equivocation* $\varepsilon_e = |\Lambda_e|$.

We can compute ε_e using Algorithm 1. Since $\overline{\pi_c} \in \mathcal{P}$ if $\pi_c \in \mathcal{P}$, we can have $\varepsilon_e = 1$ only if e is the center etuple in π_c. Otherwise, $\varepsilon_e \geq 2$. When $\varepsilon_e \leq 2$, we have learned all that we can about e using clusters, and e's privacy has clearly been compromised. Most clients will not accept the privacy compromise of *any* of their etuples, so we can state:

Definition 13. The *privacy of a PQP is compromised* if $\varepsilon_e \leq 2$ for any $e \in E$.

Having $\varepsilon_e \leq 2$ for some $e \in E$ is sufficient, but not necessary, to compromise privacy. In requiring all etuples to have equivocation at least 3, we propose a notion of privacy similar to ℓ-diversity [56], where each entity must be associated with at least ℓ sensitive values. Here, these values are loci. In Section 2.6, we demonstrate that PQPs are insecure by showing that at least one etuple's equivocation quickly drops below 3.

2.5.1 Alternate and Related Metrics

We could also measure progress toward compromising privacy in terms of uncertainty about which permutation is the correct ordering π_c. Each $\pi \in \mathcal{P}$ has equal likelihood of being π_c, and each $\pi \notin \mathcal{P}$ has likelihood zero, so we can measure this uncertainty using *permutation entropy* [9].

Definition 14. The *permutation entropy* of E is $\log_2 |\mathcal{P}|$.

Permutation entropy is straightforward to compute using a PQ-tree T. Since every tree $T' \equiv T$ represents a unique permutation $\pi \in \mathcal{P}_T$, there are $|\mathcal{P}_T|$ trees equivalent to T. Let $f(y_i)$ be the number of ways to rearrange the children of node y_i in T. Every combination of valid child arrangements yields an equivalent tree, so $|\mathcal{P}_T| = f(y_1) \cdots f(y_m)$. Thus $\log_2 |\mathcal{P}_T| = \log_2 f(y_1) + \cdots + \log_2 f(y_m)$ gives the permutation entropy, which we can compute in time $O(m) = O(n)$.

If y_i is a leaf node, $f(y_i) = 1$. If y_i is a Q-node, $f(y_i) = 2$, as y_i's children can only be in forward or reverse order. If y_i is a P-node with c_i children, then $f(y_i) = c_i!$, as the children can be arbitrarily permuted.

We give experimental results measuring permutation entropy in Section 2.6. Permutation entropy adequately measures uncertainty about π_c, but it fails to capture the idea that $\mathcal{P}$ may give more information about some etuples than others. Thus, equivocation is generally preferable.

Another alternative is to extend Algorithm 1 to count the number of permissible permutations that assign each etuple to each of its permissible loci. Intuitively, loci deemed permissible by more permutations are more likely to be correct. We could then merge this information with knowledge of the query-attribute distribution to obtain a precise metric for the uncertainty about the query-attribute value of each etuple. Unfortunately, this modification to Algorithm 1 raises its costs to $O(n^3/\log n)$ space, and $O(n^4)$ time, leaving equivocation as a better choice when n is large. Due to space constraints, we omit details for this modification of Algorithm 1.

Work in [26] uses a metric akin to permutation entropy to analyze attacks based on repeated ciphertexts. Averaging equivocations also resembles work in [26], and counting etuples with $\varepsilon_e < \kappa$ relates to *confidential intervals* in [82].

2.6 Experiments and Evaluation

We conducted experiments to study how quickly the privacy of Precise Query Protocols (PQPs) is compromised. In each experiment, we generate a set E of n etuples, and create an initial PQ-tree T with all n etuples as leaves of a single P node. We then generate a series of random range queries, obtain the cluster C of etuples returned by each query, and use each C to *reduce* T (see Section 2.3.3). We use the algorithms described in Section 2.4 to identify permissible loci and compute equivocations when needed. When averages are reported, they are computed by averaging results from 10 sets of queries issued on a single dataset.

We ran experiments using two datasets. In the *Random* dataset, each etuple is given a query-attribute value sampled uniformly with replacement from the domain $D = \mathbb{Z}_{10^8}$. For our real-world *Salary* dataset, we used a set of 162591 federal employee salaries [1] with values from $D = \mathbb{Z}_{373071}$. Only a tenth of the salaries are distinct. Over 600 have minimum value 0, while only one has maximum value. In practice, this property could be used to distinguish low and high-salary etuples, and thus to distinguish the correct ordering from its reverse. The most frequent salary appears over 7000 times.

The integer center of each query range is sampled from the uniform distribution $U(0, |D|)$. Integer query widths are sampled either from $U(0, |D|)$ or from a Gaussian distribution. We refer to such queries as *Uniform* and *Gaussian*, respectively. The Gaussian distribution is given by $N(10^5, 5 \times 10^4)$ for the Random dataset, and $N(2 \times 10^4, 10^4)$ for the Salary dataset. Uniform query widths tend to be large, while Gaussian widths are smaller. The Gaussian queries used for the Salary dataset are larger, relative to $|D|$, than for the Random dataset, to ensure that at least one query spans each pair of adjacent etuples. In the Salary dataset, the maximum separation between subse-

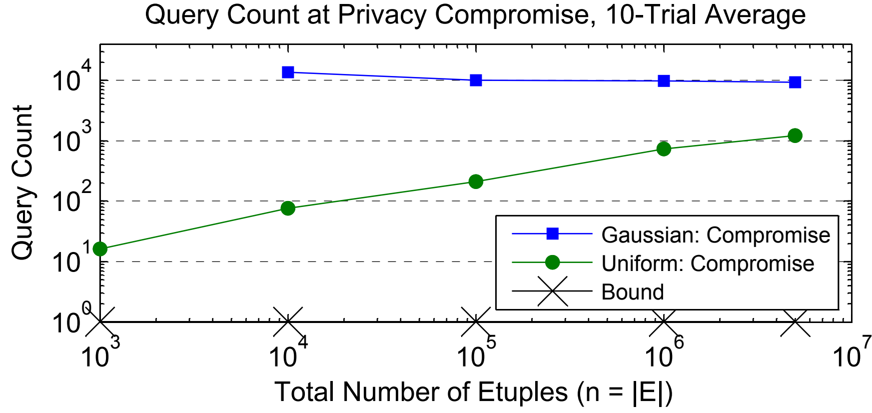

Figure 2.14: *Query Count* before *Privacy Compromise*, Random dataset.

quent query-attribute values is nearly 25000, despite its small domain size. We use the following terms:

- *Query Count*: The number of queries issued so far.
- *Total Return Count*: The total number of etuples returned by queries issued so far.
- *Distinct Return Count*: The number of distinct etuples returned so far.
- *Privacy Compromise*: The event of at least one etuple reaching equivocation $\varepsilon_e \leq 2$, as in Definition 13. Recall that this condition is sufficient, but not necessary, for PQP privacy to be compromised.

2.6.1 Progress Before Privacy Compromise

Query Count and *Total Return Count* both measure query processing work done by the server. We primarily use *Total Return Count*, as it is less sensitive to the distribution of query widths. Figures 2.14 and 2.15 show equivalent results under both metrics for the Random dataset.

Figure 2.14 gives the average *Query Count* before *Privacy Compromise* occurs. These numbers are strikingly low. Privacy is compromised sooner for the larger, Uniform

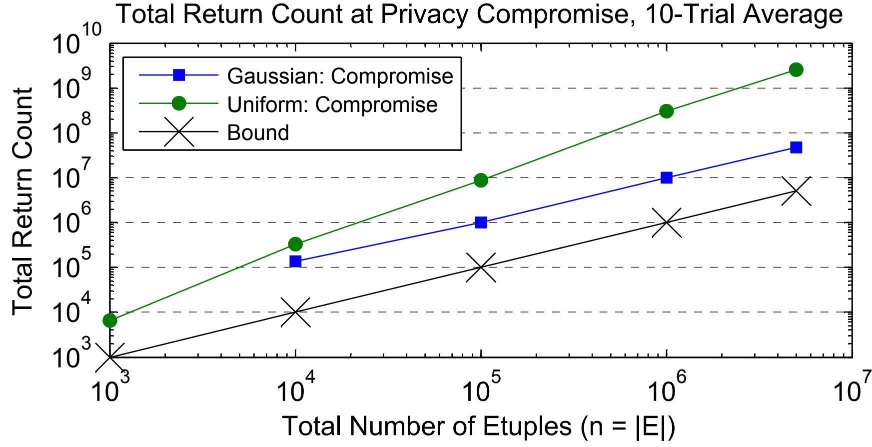

Figure 2.15: *Total Return Count* before *Privacy Compromise*, Random dataset.

queries since large queries are more likely to intersect with others, and thus exclude more permutations.

In most cases, for any etuple to have $\varepsilon_e \leq 2$, the root of T must be a Q node, which will not happen until all etuples are linked together or *covered* by overlapping clusters. Further, the clusters must be dense enough that the intersection of at least one pair of clusters contains only one etuple. For a given query width distribution, reaching sufficient coverage requires a constant number of queries, while reaching sufficient density requires a number of queries that increases when etuples are more closely spaced along the domain (larger n).

The larger, Uniform queries reach sufficient coverage before sufficient density, so as n increases, we need more of them in order to reach *Privacy Compromise*. However, the Gaussian queries reach sufficient density before sufficient coverage, so we need nearly the same number of them for all n. In fact, as n increases, the number of Gaussian queries needed drops slightly, as the regions where query ranges overlap are more likely to contain etuples, and thus to yield intersecting clusters.

Figure 2.14 shows that issuing even 100 queries is risky. In principle, even a single query can cause *Privacy Compromise* if it returns all but one etuple, as captured

by curve *Bound*. All etuples returned by such a query are contiguous, so the remaining etuple e must be assigned to locus 1 or n in the correct ordering. That is, e has $\Lambda_e = \{1, n\}$, and $\varepsilon_e = 2$.

Figure 2.15 measures against *Total Return Count* instead of *Query Count*. It shows that for small queries, on average, *Privacy Compromise* occurs after the *Total Return Count* reaches roughly 10 times the database size ($10n$). *Privacy Compromise* is just as quick for the Salary dataset ($n = 162591$), averaging 13.2 queries or 1.017×10^6 etuples returned (Uniform), and averaging 100.9 queries or 1.019×10^6 etuples (Gaussian).

While larger queries exclude more permutations, they exclude fewer permutations than several small queries with the same total size. Thus, privacy is compromised sooner for the smaller, Gaussian queries in terms of the *Total Return Count*, since the Gaussian queries exclude more permutations *per etuple returned*. This contrasts with the fact that privacy is compromised sooner for Uniform queries in terms of *Query Count* (Figure 2.14).

2.6.2 Higher Thresholds and Larger κ

If value is distributed across many records, we may permit many etuples to have small equivocations. In other cases, it is unacceptable for any etuple to have even a moderate equivocation, such as when an attacker is guessing a passcode and can afford several attempts. Figures 2.16-2.19 illustrate the rate of privacy loss in such cases by plotting the fraction of etuples with equivocation $\varepsilon_e < \kappa$, for various κ.

We explain the threshold phenomenon in Figure 2.16 by considering the *Distinct Return Count*, which is the number of distinct etuples returned so far. If the

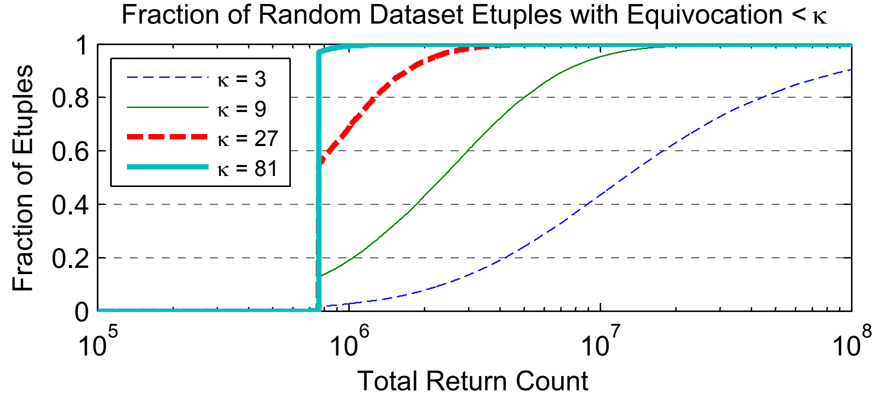

Figure 2.16: Random dataset, Gaussian widths: $N(10^5, 5 \times 10^4)$.

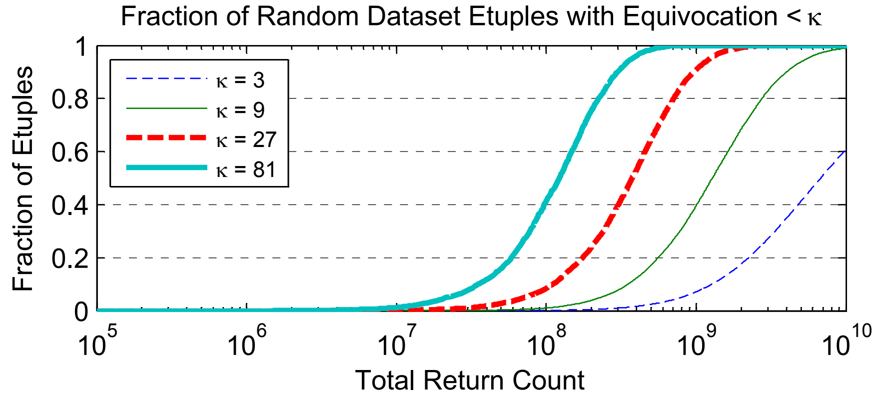

Figure 2.17: Random dataset, Uniform widths: $U(0, 10^8)$.

Distinct Return Count is at most $(n - \kappa + 1)$, then at least $\kappa - 1$ etuples have not been returned, and thus do not appear in any cluster. By Theorem 6 (Section 2.4.5), every etuple $e \in E$ then has equivocation $\varepsilon_e \geq \kappa$, for $1 < \kappa < n$. Thus, no etuples have $\varepsilon_e < \kappa$ until the *Distinct Return Count* exceeds $n - k + 1$.

This threshold is reached with only a few of the large, Uniform queries, but requires many of the small, Gaussian queries. The numerous Gaussian queries exclude many permutations, such that many etuples have equivocations only slightly larger than κ. When the threshold is reached, many such etuples drop to $\varepsilon_e < \kappa$ together, yielding the phenomenon in Figure 2.16. With the larger, Uniform queries, fewer permutations are excluded before the threshold is reached, so the trend is more gradual (Figure 2.17).

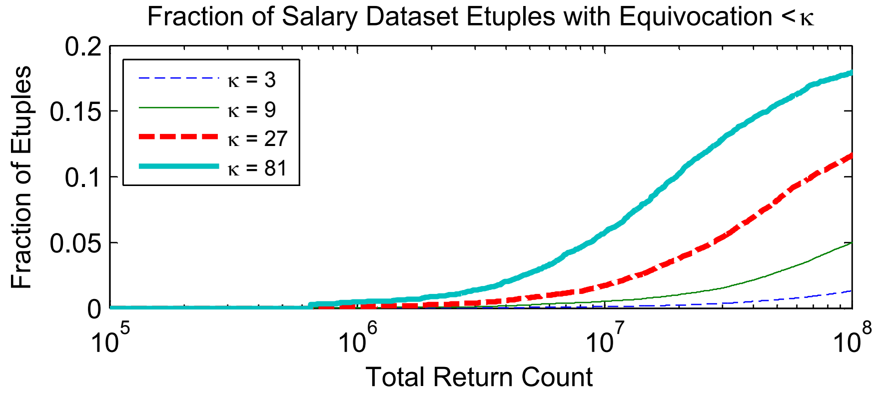

Figure 2.18: Salary dataset, Gaussian widths: $N(2 \times 10^4, 10^4)$.

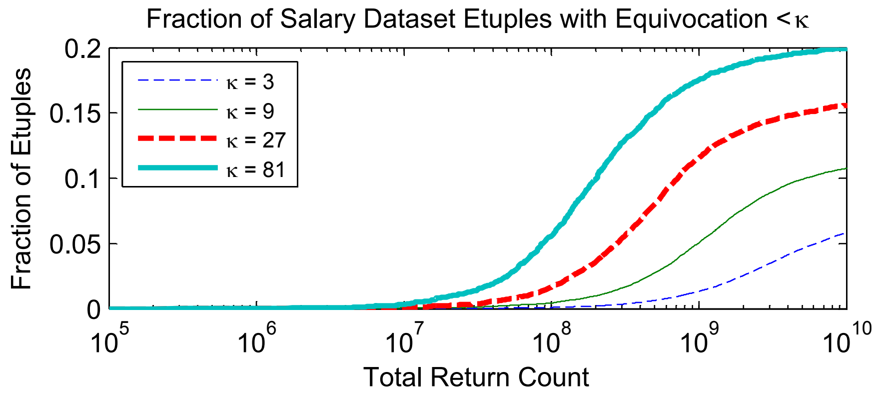

Figure 2.19: Salary dataset, Uniform widths: $U(0, 373071)$.

This same threshold phenomenon appears for Gaussian queries in the Salary dataset (Figure 2.18), though it is almost undetectable since the queries are larger relative to the domain. The Salary dataset contains many indistinguishable etuples (etuples with duplicate values), so only a few etuples can ever reach low equivocations (Figures 2.18-2.19).

2.6.3 Permutation Entropy

Figure 2.20 plots permutation entropy against the *Total Return Count* for the experiments in Figures 2.16–2.19. Permutation entropy is independent of κ and captures overall progress toward identifying the correct ordering. In these experiments,

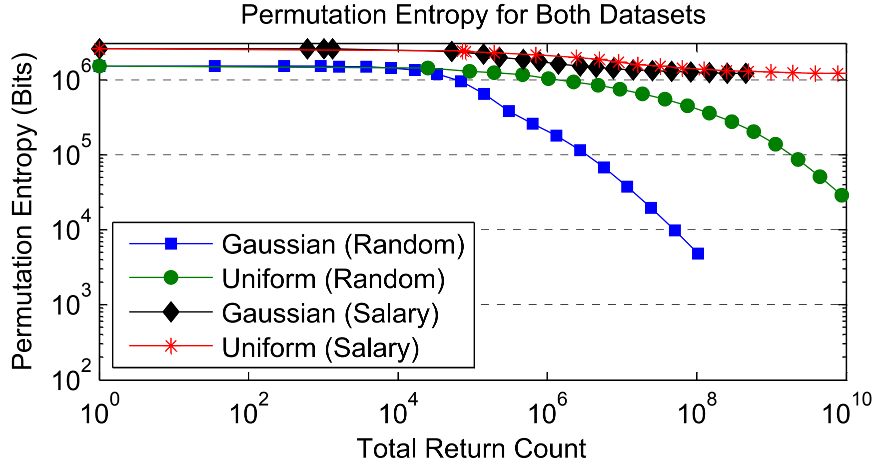

Figure 2.20: Drop rates of permutation entropy for Gaussian and Uniform query width distributions on the Random and Salary datasets.

permutation entropy remains high even after equivocation drops below 3 for several etuples, at which point privacy is compromised. Thus it appears that equivocation is a more reliable privacy metric. Permutation entropy stays higher for the Salary dataset because of the large number of indistinguishable etuples with duplicate query-attribute values.

Permutation entropy is useful when the query distribution is so skewed that many etuples are never returned and all equivocations remain large. Privacy may still be eroding, as etuples that *are* returned become relatively ordered, allowing the attacker to make limited inferences. In such cases, permutation entropy is an effective and efficient privacy metric, whereas equivocation is expensive for large κ and gives no indication of privacy loss for small κ.

2.6.4 Effects of Indexes on PQP Privacy

We have shown that the privacy of any PQP can be compromised quickly as queries are issued, regardless of the PQP's encryption and querying mechanisms. In

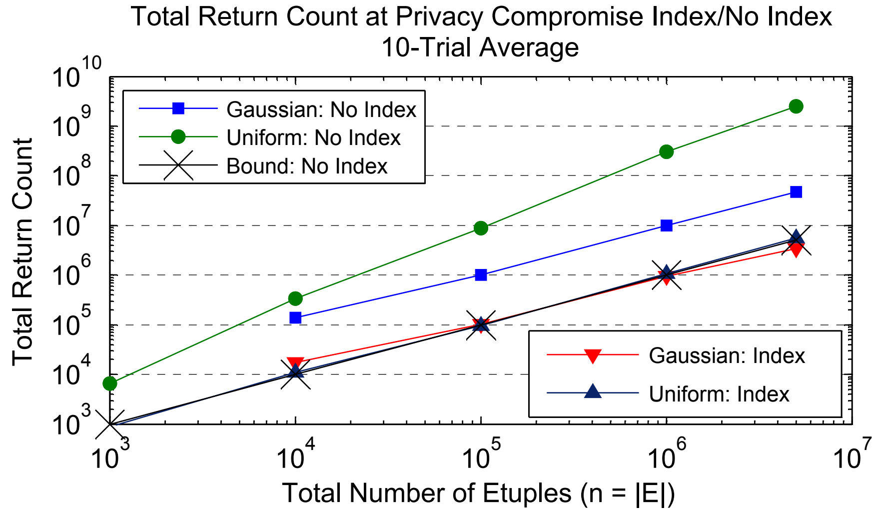

Figure 2.21: Effects of a binary range tree index on the average *Total Return Count* reached before *Privacy Compromise*.

practice, *Privacy Compromise* occurs even sooner, as most PQPs use a server-side index that leaks additional information.

For example, consider a balanced, binary range tree, with nodes that hold encrypted query-attribute ranges. The children of a single parent node hold disjoint ranges covering the parent's range. Each leaf represents a single query-attribute value, and points to all etuples with that value. The server receives encrypted query range tokens from the client and uses them to search the tree by homomorphically checking whether query and node ranges overlap, all without learning the plaintext ranges. Similar indexes can be found in [17] and [51].

In such an index, etuples descended from each node form a cluster. These clusters, along with those obtained from query result sets, can be used to reduce a PQ-tree. In Figure 2.21, we see that including such an index dramatically reduces the average *Total Return Count* reached before *Privacy Compromise*.

2.6.5 Consequences and Alternatives

The Figures above show that in a PQP that supports range queries, *Privacy Compromise* can occur quickly. Faced with this reality, clients have two options: abandon PQPs, or assume that query-attribute distributions are hidden.

2.6.5.1 Assuming Attribute Distributions are Hidden

The client may choose to make the dangerous assumption that the attacker will never learn the query-attribute distribution. In this case, our work shows that the client should operate under the assumption that the attacker knows the correct etuple ordering. PQPs like OPES [8] are designed for this scenario, and claim decent privacy properties [10] and excellent efficiency. Thus it is hard to see much advantage to using a more complex PQP, which will likely be less efficient (see Table 2.1).

2.6.5.2 Abandoning PQPs Altogether

If the distribution is known, the client must find alternatives to PQPs. One option is to use partitioning schemes [40, 43], which make privacy guarantees in terms of entropy and indistinguishability, at the cost of spurious results.

An alternative is to periodically re-encrypt etuples using a new key. The attacker cannot correlate old and re-encrypted etuples, so he must then restart his attack. As Figure 2.15 shows, we must re-encrypt *at least* one etuple for every 10–100 returned to retain even a basic level of privacy.

Re-encryption, also called *node swapping* or *shuffling*, is already used to support privacy-preserving point queries in work on oblivious index traversal techniques [30, 53], and Oblivious RAM [73]. Oblivious index traversals generally use spurious queries and a tunable constant number of re-encryptions per etuple returned in order to

achieve probabilistic privacy guarantees. Existing Oblivious RAM schemes require at least $\log n$ re-encryptions per etuple returned, and achieve provable access pattern indistinguishability by requesting spurious items and frequently re-encrypting recently requested items. Such techniques are promising, but have not yet been optimized for range queries.

2.7 Conclusion

We have presented an attack that can be used to infer attribute values of encrypted records in any Precise Query Protocol (PQP) that supports one-dimensional range queries. We mounted the attack using PQ-trees and a novel algorithm for identifying permissible loci. Experimental results demonstrate that our attack requires us to observe only 10^4 queries to compromise privacy for a database of over 10^6 records, indicating that PQPs are highly insecure when query-attribute distributions are known. Future research on privacy-preserving range queries should investigate efficient alternatives to PQPs.

We will explore additional privacy metrics for PQPs in future work. We are also interested in finding an analog to our attack for multi-dimensional queries, where returned etuples are contiguous along multiple attributes at once.

Chapter 3

Security Limitations of Using Secret Sharing for Data Outsourcing

3.1 Introduction

As cloud computing grows in popularity, huge amounts of data are being outsourced to cloud-based database service providers for storage and query management. However, some customers are unwilling or unable to entrust their raw sensitive data to cloud providers. As a result, privacy-preserving data outsourcing solutions have been developed around an *honest-but-curious* database server model. In this model, the server is trusted to correctly process queries and manage data, but may try to use the data it manages for its own nefarious purposes.

Private outsourcing schemes keep raw data hidden while allowing the server to correctly process queries. Queries can be issued only by a trusted client, who has

insufficient resources to manage the database locally. Most such schemes use specialized encryption or complex mechanisms, ranging from order-preserving encryption [8], which has limited security and high efficiency, to oblivious RAM [77], which has provable access pattern indistinguishability but poor performance.

Three recent works [6, 41, 80] propose outsourcing schemes based on Shamir's secret sharing algorithm [70] instead of encryption. We refer to these works by their authors' initials, HJ [41], AAEMW [6], and TSWZ [80], and in aggregate as the HAT schemes. The AAEMW scheme was also published in [7]. In secret sharing, each sensitive data element, called a *secret*, is split into n *shares*, which are distributed to n data servers. To recover the secret, the client must combine shares from at least k servers. Secret sharing has perfect information-theoretic security when at most $k-1$ of the n servers *collude* (exchange shares) [70].

Since secret sharing requires only k multiplications to reconstruct a secret, proponents argue that HAT schemes are faster than encryption based schemes. Other benefits of the HAT schemes include built-in redundancy, as only k of the n servers are needed, and additive homomorphism, which allows SUM queries to be securely processed by the server and returned to the client as a single value. Each of the HAT schemes makes two security claims:

Claim 1. The scheme achieves perfect information-theoretic security when at most $k-1$ servers collude.

Claim 2. When k or more servers collude, the scheme still achieves adequate security as long as certain information used by the secret sharing algorithm, namely a prime p and a vector $\vec{X}$, are kept private, known only to the client.

It is doubtful that the HAT schemes truly fulfill Claim 1, as they sort shares by secret value, which certainly reveals some information about the data [47]. Further, the AAEMW scheme [6] uses correlated coefficients in the secret sharing algorithm instead of random ones, which also contradicts this claim. Nevertheless, for the purposes of this work, we assume that Claim 1 holds.

We are primarily concerned with evaluating Claim 2, which asserts that even k or more colluding servers cannot easily recover secrets. Claim 2 is stated in Section 4.5 of [41], Section 3 of [6], Sects. 2.2 and 6.1 of [80], and Section 3 of [7].

3.1.1 Our Contribution

Our contribution is to demonstrate that all three HAT schemes [6, 41, 80] fail to fulfill Claim 2. We give a practical attack that can reconstruct all secrets in the database when k servers collude, even when p and $\vec{X}$ are kept private. Our attack assumes that the servers know, or can discover, at least $k + 2$ secrets. To limit data storage costs, k is kept small ($k \approx 10$), so discovering $k + 2$ secrets is feasible (see Section 3.4.2). All three HAT schemes argue that they fulfill Claim 2, so our result provides a much-needed clarification of their security limitations.

The TSWZ scheme [80] argues that if p is known, secrets could be recovered. However, it provides no attack description, and argues that large primes are prohibitively expensive to recover. Our attack recovers 8192-bit primes in less time than [80] needed to recover 32-bit primes. In fact, we can generally recover large primes in less time than the client takes to generate them (Section 3.5.1).

In Section 3.2 we review Shamir's secret sharing algorithm and how it is used for private outsourcing. In Section 3.3 we give assumptions and details of our attack, and we show how to align shares and discover secrets in Section 3.4. We give experimental

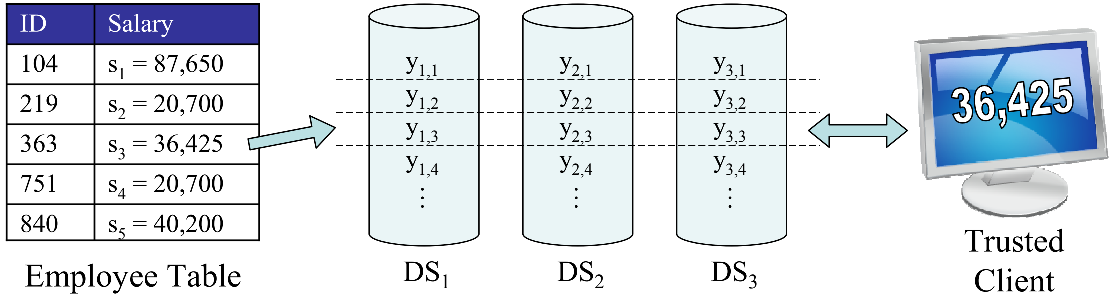

Figure 3.1: Secret (salary) data from an employee table is split into shares and distributed to multiple data servers. A trusted client queries shares from the data servers and combines them to recover the secrets.

runtime results in Section 3.5, and discuss possible attack mitigations in Section 3.6. We discuss related work in Section 3.7 and conclude with Section 3.8.

3.2 Data Outsourcing Using Secret Sharing

We now review Shamir's secret sharing scheme and show how it is used for private data outsourcing in the HAT schemes [6, 41, 80]. We use an employee table with m records as our driving example, where queries are issued over the salary attribute. Each salary $s_1, \ldots, s_m$ is a secret that is shared among the n data servers (see Figure 3.1).

3.2.1 Shamir's Secret Sharing

Shamir's secret sharing scheme [70] is designed to share a single secret value s_j among n servers such that shares must be obtained from any k servers in order to reconstruct s_j. The scheme's security rests on the fact that at least k points are needed to uniquely reconstruct a polynomial of degree $k-1$. Theoretically, the points and coefficients used in Shamir's scheme can be taken from any field $\mathbb{F}$. However, to use the scheme on finite-precision machines, we require $\mathbb{F}$ to be the finite field $\mathbb{F}_p$, where p is a prime.

To share s_j, we choose a prime $p > s_j$, and $k-1$ coefficients $a_{1,j}, \ldots, a_{k-1,j}$ selected randomly from $\mathbb{F}_p$. We then construct the following polynomial:

$$q_j(x) = s_j + \sum_{h=1}^{k-1} a_{h,j} x^h \mod p \tag{3.1}$$

We then generate a vector $\vec{X} = (x_1, \ldots, x_n)$ of distinct elements in $\mathbb{F}_p$, and for each data server DS_i, we compute the *share* $y_{i,j} = q_j(x_i)$. Together, x_i and $y_{i,j}$ form a point $(x_i, y_{i,j})$ through which polynomial $q_j(x)$ passes.

Given any k such points $(x_1, y_{1,j}), \ldots, (x_k, y_{k,j})$, we can reconstruct the polynomial $q_j(x)$ using Lagrange interpolation:

$$q_j(x) = \sum_{i=1}^{k} y_{i,j} \ell_i(x) \mod p \tag{3.2}$$

where $\ell_i(x)$ is the Lagrange basis polynomial:

$$\ell_i(x) = \prod_{1 \le j \le k, j \ne i} (x - x_j)(x_i - x_j)^{-1} \mod p \tag{3.3}$$

and $(x_i - x_j)^{-1}$ is the multiplicative inverse of $(x_i - x_j)$ modulo p.

The secret s_j is the polynomial q_j evaluated at $x = 0$, so we get:

$$s_j = \sum_{i=1}^{k} y_{i,j} \ell_i(0) \mod p \tag{3.4}$$

Given only $k' < k$ shares, and thus only k' points, we cannot learn anything about s, since for any value of s, we could construct a polynomial of degree $k-1$ that passes through all k' points. Thus Shamir's scheme offers perfect, information-theoretic security against recovering s_j from fewer than k shares [70].

3.2.2 Data Outsourcing via Secret Sharing

We now describe the mechanism used by all three HAT schemes to support private outsourcing via secret sharing. We first choose a single prime p and a vector $\vec{X}$, which are the same for all secrets and will be stored locally by the client. For each secret

s_j, we generate coefficients $a_{1,j}, \ldots, a_{k-1,j}$, and produce a polynomial $q_j(x)$ as in (3.1). We then use q_j to split s_j into n shares $y_{1,j}, \ldots, y_{n,j}$, where $y_{i,j} = q_j(x_i)$, and distribute each share $y_{i,j}$ to server DS_i, as in Figure 3.1.

An important distinction is that the AAEMW scheme performs secret sharing over the real number field $\mathbb{R}$, so there is no p to choose. However, since the scheme must run on finite precision hardware, any implementation will suffer from roundoff error. Our attack works over $\mathbb{R}$, and is efficient because the field is already known. However, we expect that in practice, the AAEMW scheme will switch to a finite field $\mathbb{F}_p$, so we do not treat it as a special case.

When the client issues a point query for the salary s_j of a particular employee, he receives a share from each of the n servers. Using any k of these shares, he can recover s_j using the interpolation equation (3.4). Other query types, including range and aggregation queries, are supported by the HAT schemes. We give some relevant details in Section 3.2.4, but the rest can be found in [6, 41, 80].

$\vec{X}$ and p are re-used across secrets for two reasons. First, storing distinct $\vec{X}$ or p on the client for each secret would require at least as much space as storing the secret itself. Second, when the same $\vec{X}$ and p are used, the secret sharing scheme has additive homomorphism. That is, if each server DS_i adds shares $y_{i,1} + y_{i,2}$, and we interpolate using those sums, the recovered value is the sum $s_1 + s_2$. With additive homomorphism, when the client issues a SUM query, the server can sum the relevant shares, and return a single value to the client, instead of returning shares separately and having the client perform the addition. Using a different $\vec{X}$ or p for each secret breaks additive homomorphism.

3.2.3 Security

If $\vec{X}$ and p are public, and k or more servers collude, then the HAT schemes are clearly insecure, as the servers could easily perform the interpolation themselves. On the other hand, if at most $k-1$ servers collude, and coefficients are chosen independently at random from $\mathbb{F}_p$, then the servers learn nothing about a secret by examining its shares, and Claim 1 is fulfilled.

The HAT schemes state that by keeping $\vec{X}$ [6, 41] and p [80] private, they achieve security even when k or more servers collude (Claim 2). Our attack shows that any k colluding servers can recover all secrets in the database, even when $\vec{X}$ and p are unknown (Section 3.3), contradicting Claim 2.

3.2.4 Supporting Range and Aggregation Queries

We can use the mechanisms that support range and SUM queries in the HAT schemes to reveal the order of the shares on each server according to their corresponding secret values. We then use these orders to align corresponding shares across colluding servers, and to discover key secret values (see Section 3.4).

The AAEMW scheme [6] crafts coefficients such that the shares preserve the order of their secrets. The HJ and TSWZ schemes [41, 80] both use a single B^{+} tree to order each server's shares and facilitate range queries. TSWZ assumes the tree is accessible to all servers, while HJ assumes it is on a separate, non-colluding index server. HJ obscures share order from the servers, but we can reconstruct it by observing range queries over time (see Section 3.4.3).

3.3 Attack Description

We now show that the HAT schemes are insecure when k or more servers collude, even if $\vec{X}$ and p are kept private. Our attack efficiently recovers all secret values (salaries in Figure 3.1) stored in the database, and relies on the following assumptions:

1. At least k servers collude, exchanging shares or other information.
2. The number of servers k and the number of bits b in prime p are modest: $k \approx 13$, $b \approx 2^{13}$. None of the HAT schemes give recommended values for k or b, with the exception of a brief comment in [6] alluding to 16-bit primes originally suggested by Shamir. In practice, primes with more than 2^{13} bits take longer for the client to generate than for our attack to recover, and the cost of replicating data to every server keeps k small.
3. $\vec{X}$ and p are unknown, and are the same for each secret (see Section 3.2.2).
4. Each set of k corresponding shares can be *aligned*. That is, the colluding servers know which shares correspond to the same secret, without knowing the secret itself. We can align shares if we know share orders (see Section 3.4.1).
5. At least $k+2$ secrets, and which shares they correspond to, are known or can be discovered. Since k is modest, knowing $k+2$ secrets is reasonable, especially when the number of secrets m is large (see Section 3.4.2).

In Section 3.6, we show that modifying the HAT schemes to violate these assumptions sacrifices performance, functionality, or generality, eroding the schemes' slight advantages over encryption based techniques.

3.3.1 Recovering Secrets when p is Known and $\vec{X}$ is Private

As a stepping stone to our full attack, we show how to recover secrets if p is already known. Without loss of generality, let $s_1, \ldots, s_k$ be known secrets, and let $DS_1, \ldots, DS_k$ be the colluding servers. For each secret s_j, we have shares $y_{1,j}, \ldots y_{k,j}$, generated by evaluating $q_j(x)$ at $x_1, \ldots, x_k$, respectively. We therefore have a system of k^2 equations of the form $y_{i,j} = s_j + \sum_{h=1}^{k-1} a_{h,j} x_i^h \mod p$, as in (3.1). The system has $k(k-1)$ unknown coefficients $a_{h,j}$, and k unknown x_i, giving k^2 equations in k^2 unknowns. Thus, it would seem we can solve for the relevant values of $\vec{X}$, which would allow us to recover the remaining secrets. Unfortunately, the system is non-linear, so naively solving it directly requires expensive techniques such as Groebner basis computation [14].

Instead, we can recover the remaining secrets without solving for $\vec{X}$. Consider the following system of equations obtained by applying the interpolation equation (3.4) to each of the k secrets:

$$\begin{aligned}
y_{1,1}\ell_1(0) + y_{2,1}\ell_2(0) + \cdots + y_{k,1}\ell_k(0) - s_1 &\equiv 0 \pmod p \\
y_{1,2}\ell_1(0) + y_{2,2}\ell_2(0) + \cdots + y_{k,2}\ell_k(0) - s_2 &\equiv 0 \pmod p \\
&\vdots \\
y_{1,k}\ell_1(0) + y_{2,k}\ell_2(0) + \cdots + y_{k,k}\ell_k(0) - s_k &\equiv 0 \pmod p
\end{aligned} \tag{3.5}$$

If we treat each basis polynomial value $\ell_i(0)$ as an unknown, we get k unknowns $\ell_1(0), \ldots, \ell_k(0)$, which we call *bases*, in k linear equations. Since we know p, we can easily solve (3.5) using Gaussian elimination and back-substitution. We can then use the bases to recover the remaining secrets in the database via (3.4).

We can construct (3.5) since we know that all shares from a given server DS_i were obtained from the same x_i, and thus should be multiplied by the same base $\ell_i(0)$.

The client could obscure the correspondence between shares by mixing shares among servers, but would be forced to store i with each share in order to properly reconstruct the secret. To completely hide the correspondence, i itself would need to be padded and encrypted, which is precisely what secret sharing tries to avoid. Further, mixing the shares would break additive homomorphism.

3.3.2 Recovering p when $\vec{X}$ and p are Private

Let b be the number of bits used to represent p. We can easily have $b > 2^6$, so enumerating possible values for p is not practical. However, we can recover p by exploiting known shares and the $k + 2$ known secrets. Our attack identifies two composites δ_1 and δ_2 both divisible by p ($p|\delta_1, p|\delta_2$), such that the remaining factors of δ_1, δ_2 are largely independent. We then take δ' to be the greatest common divisor of δ_1 and δ_2, and factor out small primes from δ', leaving us with $\delta' = p$ with high probability. Once p is known, we can use the attack from Section 3.3.1 to recover the bases and the remaining, unknown secrets.

3.3.2.1 Computing δ_1, δ_2.

Without loss of generality, we let $s_1, \ldots, s_{k+2}$ be the known secrets. To compute δ_γ, $\gamma \in \{1, 2\}$, we consider the system of interpolation equations for secrets $s_\gamma, \ldots, s_{\gamma+k}$ as in (3.5), represented by the following $(k + 1) \times (k + 1)$ matrix:

$$\begin{bmatrix} y_{1,\gamma} & y_{2,\gamma} & \cdots & y_{k,\gamma} & -s_\gamma \\ y_{1,\gamma+1} & y_{2,\gamma+1} & \cdots & y_{k,\gamma+1} & -s_{\gamma+1} \\ \vdots & & \ddots & & \vdots \\ y_{1,\gamma+k-1} & y_{2,\gamma+k-1} & \cdots & y_{k,\gamma+k-1} & -s_{\gamma+k-1} \\ y_{1,\gamma+k} & y_{2,\gamma+k} & \cdots & y_{k,\gamma+k} & -s_{\gamma+k} \end{bmatrix} \tag{3.6}$$

Since p is unknown, we cannot compute inverses modulo p and thus cannot divide as in standard Gaussian elimination. However, we can still convert (3.6) to upper triangular (row echelon) form using only multiplications and subtractions.

We start by eliminating coefficients for $\ell_1(0)$ from all but the first row ($j = \gamma$). To eliminate $\ell_1(0)$ from row $j > \gamma$, we multiply the contents of row γ through by $y_{1,j}$, and of row j by $y_{1,\gamma}$, producing a common coefficient for $\ell_1(0)$ in both rows. We then subtract the multiplied row γ from the multiplied row j, canceling the coefficient for $\ell_1(0)$. Row 1 is left unchanged, but row j now has coefficient 0 for $\ell_1(0)$, and coefficient $(y_{i,j})(y_{1,\gamma}) - (y_{i,\gamma})(y_{1,j})$ for $\ell_i(0), i \geq 2$:

$$\begin{bmatrix} y_{1,\gamma} & y_{2,\gamma} & \cdots & -s_\gamma \\ 0 & (y_{2,\gamma+1})(y_{1,\gamma}) - (y_{2,\gamma})(y_{1,\gamma+1}) & \cdots & (-s_{\gamma+1})(y_{1,\gamma}) - (-s_\gamma)(y_{1,\gamma+1}) \\ \vdots & & \ddots & \vdots \\ 0 & (y_{2,\gamma+k})(y_{1,\gamma}) - (y_{2,\gamma})(y_{1,\gamma+k}) & \cdots & (-s_{\gamma+k})(y_{1,\gamma}) - (-s_\gamma)(y_{1,\gamma+k}) \end{bmatrix}$$

We then repeat the process, eliminating successive coefficients from lower rows, until the matrix is in upper triangular form:

$$\begin{bmatrix} y_{1,\gamma} & y_{2,\gamma} & \cdots & y_{k,\gamma} & -s_\gamma \\ 0 & c_{2,\gamma+1} & \cdots & c_{k,\gamma+1} & c_{k+1,\gamma+1} \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & c_{k,\gamma+k} & c_{k+1,\gamma+k} \\ 0 & 0 & \cdots & 0 & \delta_\gamma \end{bmatrix} \tag{3.7}$$

We use $c_{i,j}$ values to denote constants. In the last row of (3.7), the coefficient for every $\ell_i(0)$ is 0, so the row represents the equation $\delta_\gamma \equiv 0 \pmod{p}$. Thus, $p | \delta_\gamma$.

3.3.2.2 Size of δ_1, δ_2.

As coefficients for successive $\ell_i(0)$ are eliminated, each non-zero cell below the ith row is set to the difference of products of two prior cell values, doubling the number of bits required by the cell. Thus, the number of bits per cell in (3.7) is given by:

$$\begin{bmatrix} b & b & \cdots & b & b \\ 0 & 2^1 b & \cdots & 2^1 b & 2^1 b \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 2^{k-1} b & 2^{k-1} b \\ 0 & 0 & \cdots & 0 & 2^k b \end{bmatrix}$$

As a result, each of δ_1, δ_2 has at most $2^k b$ bits. This is closely related to the result that in the worst case, simple integer Gaussian elimination leads to entries that are exponential in the matrix size [33].

3.3.2.3 Recovering p from δ_1, δ_2.

Since δ_1, δ_2 have $2^k b$ bits, and p has only b bits, it is likely that δ_1, δ_2 both have some prime factors larger than p, so factoring them directly is not feasible. Instead, we take $\delta' = gcd(\delta_1, \delta_2)$, where gcd is the greatest common divisor function, which can be computed using the traditional Euclidean algorithm, or more quickly using Stein's algorithm [79].

Since δ_1 and δ_2 were obtained using different elimination orders and sets of secrets, they rarely share large prime factors besides p, so all other prime factors of δ' should be small. Thus, we can factor δ' by explicitly dividing out all prime factors with at most β bits, leaving behind only p, with high probability. We know that p is larger than all shares, so to avoid dividing out p itself, we never divide out primes that are larger than the largest known share. We have found empirically that the probability that δ_1, δ_2, as computed above, share a factor with more than β bits can be approximated by $\frac{2^{(k-2)/4}}{2^{\beta+1}} k$ for the values of β, k we are interested in (Section 3.5.2). Our attack fails if δ_1, δ_2 share such a factor, but we can increase β to make the failure rate arbitrarily low.

3.3.3 Attack Complexity

Since δ_1 and δ_2 are both $(2^k b)$-bit integers, the time required to find $gcd(\delta_1, \delta_2)$ is in $O(2^{2k}b^2)$ [79]. As k grows, storing δ_1, δ_2 and computing their gcd quickly become the dominant space and time concerns, respectively. Thus, recovering p has space complexity $O(2^k b)$ and time complexity $O(2^{2k}b^2)$.

Recovering the bases, once p is known, has space complexity $O(k^2 b)$ for storing the matrix, and time complexity dominated either by computing $O(k^3)$ b-bit integer multiplications during elimination, or $O(k)$ modular inverses during back-substitution. Clearly, these costs are dominated by the costs of recovering p. Once p and the bases have been recovered, the time spent recovering a secret is the same for the colluding servers as it is for the trusted client.

3.3.4 Example Attack for $k = 2$

We now demonstrate our attack on a simple dataset with $m = 6$ records shared over $n = k = 2$ servers. We choose the 6-bit prime $p = 59$ and select $x_1 = 17, x_2 = 39$. We then generate secrets, coefficients, and shares as follows:

$s_1 = 18$	$a_{1,1} = 18$	$q(x_1, s_1) = 29$	$q(x_2, s_1) = 12$
$s_2 = 36$	$a_{1,2} = 5$	$q(x_1, s_2) = 3$	$q(x_2, s_2) = 54$
$s_3 = 22$	$a_{1,3} = 17$	$q(x_1, s_3) = 16$	$q(x_2, s_3) = 36$
$s_4 = 10$	$a_{1,4} = 28$	$q(x_1, s_4) = 14$	$q(x_2, s_4) = 40$
$s_5 = 39$	$a_{1,5} = 31$	$q(x_1, s_5) = 35$	$q(x_2, s_5) = 9$
$s_6 = 57$	$a_{1,6} = 51$	$q(x_1, s_6) = 39$	$q(x_2, s_6) = 40$

We assume s_1, s_2, s_3, s_4 are known. We first generate the matrix in (3.6) using s_1, s_2, s_3 ($\gamma = 1$), and do the following elimination to get $\delta_1 = 307980$:

$$\begin{bmatrix} 29 & 12 & -18 \\ 3 & 54 & -36 \\ 16 & 36 & -22 \end{bmatrix} \rightarrow \begin{bmatrix} 29 & 12 & -18 \\ 0 & 1530 & -990 \\ 0 & 852 & -350 \end{bmatrix} \rightarrow \begin{bmatrix} 29 & 12 & -18 \\ 0 & 1530 & -990 \\ 0 & 0 & 307980 \end{bmatrix}$$

We do the same with s_2, s_3, s_4 ($\gamma = 2$) to get $\delta_2 = -33984$:

$$\begin{bmatrix} 3 & 54 & -36 \\ 16 & 36 & -22 \\ 14 & 40 & -10 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 54 & -36 \\ 0 & -756 & 510 \\ 0 & -636 & 474 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 54 & -36 \\ 0 & -756 & 510 \\ 0 & 0 & -33984 \end{bmatrix}$$

We then compute $\delta' = gcd(\delta_1, \delta_2) = 2124$, and factor δ' by dividing out the small prime factors $2 \cdot 2 \cdot 3 \cdot 3$, to get $p = 59$, as expected. Now we can recover the bases using the following system of equations as in (3.5):

$$\begin{aligned} 29\ell_1(0) + 12\ell_2(0) - 18 &\equiv 0 \pmod{59} \\ 3\ell_1(0) + 54\ell_2(0) - 36 &\equiv 0 \pmod{59} \end{aligned}$$

We eliminate $\ell_1(0)$ from the second equation, giving $55\ell_2(0) \equiv 46 \pmod{59}$. We then compute the inverse $(55)^{-1} \pmod{59} = 44$, giving $\ell_2(0) = 46 \cdot 44 \mod 59 = 18$. We then back-substitute to get $\ell_1(0) = 42$. To verify, we compute s_5 and s_6 using (3.4), giving $s_5 = 35 \cdot 42 + 9 \cdot 18 \pmod{59} = 39$, and $s_6 = 39 \cdot 42 + 40 \cdot 18 \pmod{59} = 57$, as expected.

3.4 Aligning Shares and Discovering Secrets

In order to mount our attack, we must be able to *align* shares across colluding servers. That is, given the set of shares from each of k servers, we must be able to identify which subsets of k shares, one from each server, were obtained from the same polynomial q_j (3.1), even if we do not know the secret value s_j itself. Further, we must know, or be able to discover, at least $k + 2$ secret values and the subset of k shares

to which they correspond. We now show how we can satisfy these assumptions for the HAT schemes [6, 41, 80] using knowledge of *share order*.

In the AAEMW [6] and TSWZ [80] schemes, the shares on each server are explicitly, totally ordered (Section 3.2.4). The share order sorts the shares in non-decreasing order of their corresponding secrets. If two shares are obtained from distinct polynomials, but the same secret, they have the same relative order on each server. In the HJ scheme [41], shares are totally ordered, but the order is hidden from the data servers. In this case, we can infer a partial share order by observing queries over time.

3.4.1 Aligning Shares

When the total share order on each data server is known, we simply align the jth share from each server. If only a partial order is known, as in the HJ scheme, the alignment of some shares will be ambiguous. To recover secrets for such shares, we must either try multiple alignments, or wait for more queries to arrive, and use them to refine the partial order and eliminate the ambiguity (see Section 3.4.3).

3.4.2 Discovering $k + 2$ Secrets

We have shown that given $k + 2$ secrets and their corresponding shares, our attack can recover all remaining secrets. This weakness is a severe limitation of the HAT schemes, and contradicts Claim 2 (Section 3.1). In practice, $k \ll m$, where m is the number of secrets, so assuming $k + 2$ known secrets is reasonable. Our attack is independent of the mechanism used to discover these secrets.

Simple methods for learning $k + 2$ secrets include a known plaintext attack, where we convince the trusted client to insert $k+2$ known secrets, and a known ciphertext attack, where the client reveals at least $k+2$ secrets retrieved by some small range query.

Since shares are ordered according to their secret values, we can easily identify which subsets of shares from the query go with each secret.

We can also infer secret values using share order. Consider an employee table with secret salaries, as in Figure 3.1. If at least $k + 2$ employees earn a well-known minimum-wage salary, then the share order reveals that the first $k + 2$ shares have this known salary. Alternatively, there may be $k + 2$ employees who anonymously post their salaries. If we can estimate the distribution of salaries in the database, we can guess roughly where the known salaries fall in the order, and run the attack for nearby guesses until we get a solution with a recoverable prime and recovered secrets that fit the expected order.

3.4.3 Inferring Order in the HJ Scheme

If a scheme hides the share order from the data server, share alignment and secret discovery become harder. The HJ scheme [41] stores the share order for each data server on a single index server that ostensibly does not collude with any data servers. The client sends each query to the index server, and the response tells the client which shares to request from each data server.

In the simplest case, we can align shares by observing point queries, which return only one share from each server. If the colluding servers all observe an isolated request for a single share at the same time, they can assume the shares satisfy a point query, and thus that they all correspond to the same secret. Given enough point queries, we can align enough shares to mount our attack. However, if point queries are rare, this technique will take too long to be useful.

More generally, we can order shares on each server by observing overlapping range queries. In the HJ scheme, a range query appears to the data server as set of

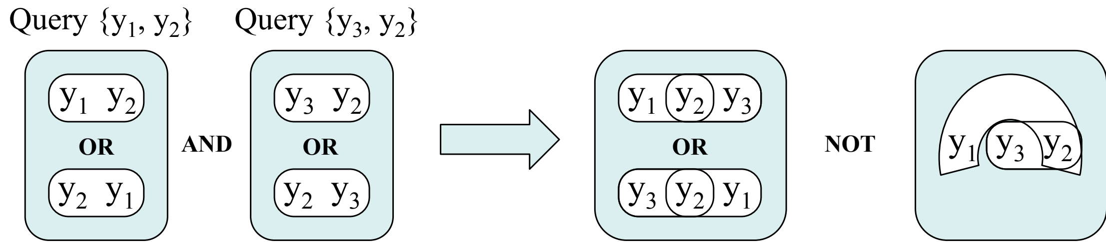

Figure 3.2: Range queries indicate that the secrets of shares y_1, y_2 are contiguous, as are those of y_3, y_2. Thus, the secret of y_2 falls between the secrets of y_1 and y_3, though either y_1 or y_3 may have the smallest secret.

unordered share requests. Since range queries request shares that have secrets inside a given range, we know that secrets of requested shares are contiguous. We use this information to order shares according to their secret values.

Consider an example where a client issues two range queries to the same data server. The first query returns shares $\{y_1, y_2\}$, and the second, shares $\{y_3, y_2\}$. Each query is a range query, so the server knows that no secret can fall between the secrets of y_1 and y_2 or of y_3 and y_2. Since y_2 appears in both queries, the server knows that the secret of y_2 comes between the secrets of y_1 and y_3, but is not sure whether the secret of y_1 or of y_3 is smaller. Thus, the true share order contains either subsequence $y_1y_2y_3$ or $y_3y_2y_1$, and we say that the server knows the share order of $\{y_1, y_2, y_3\}$ up to symmetry (see Figure 3.2).

We can extend this technique to additional range queries of varying sizes. Given enough queries, we can reconstruct the entire share order on each server up to symmetry. The full reconstruction algorithm uses PQ-trees [13] and is discussed in Chapter 2. We can link reconstructed share orders across servers, and thereby align shares, by observing that if a query issued to one data server requests the jth share, then the same query must also request the jth share from every other server. If we use the share order to

discover secrets, we must make twice as many guesses, since we still only know the order up to symmetry.

3.5 Attack Implementation and Experiments

We implemented our attack in Java, and ran each of our attack trials using a single thread on a 2.4GHz Intel® Core™2 Quad CPU. All trials used less than 2GB RAM. We used two datasets. The first consists of $m = 1739$ maximum salaries of Riverside County (California, USA) government employees as of February, 2012 [2]. The second is a set of $m = 10^5$ salaries generated uniformly at random from the integer range $[0, 10^7)$.

3.5.1 Time Measurements

Our first set of experiments measures the time required to run the full attack as described in Section 3.3. Each experiment varies the number of servers k or the number of bits b in the hidden prime p. The total number of servers n has no effect on the attack runtime, so we let $n = k$. All times are averaged over 10 independent trials, and averages are rounded up to a 1ms minimum. In each trial, we divide out primes with at most $\beta = 16$ bits (Section 3.3.2.3), and we successfully recover p, all k bases ($\ell_i(0)$ values), and all m secrets.

Each plot gives the times spent by the client finding a random b-bit prime p and creating k shares for each of the m secrets. We then plot the times spent by the colluding servers recovering p and the k bases. We also give the time spent recovering all m secrets, which is the same for the colluding servers as it is for the client. From Section 3.3.3, we know that the time needed to recover p is in $O(2^{2k}b^2)$. Thus, incrementing k or $\log_2 b$

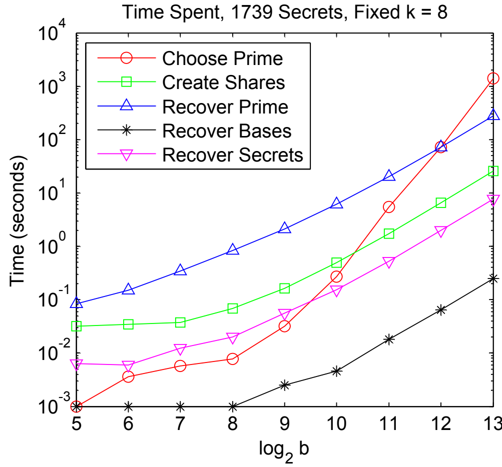

Figure 3.3: Riverside dataset times, varied b

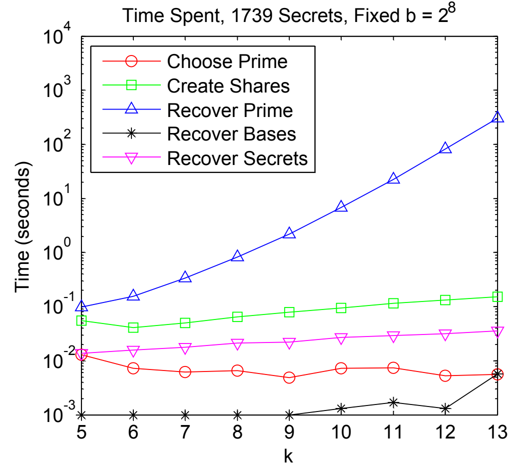

Figure 3.4: Riverside dataset times, varied k

increases prime recovery time by a factor of 4. Since k and $\log_2 b$ have similar effects on prime recovery time, we plot against $\log_2 b$ instead of b on the x axis.

Figures 3.3 and 3.4 plot times using the Riverside dataset with fixed $k = 8$ and $b = 2^8$, respectively. Figures 3.5 and 3.6 give corresponding times for the random dataset. Times to create shares and recover secrets are proportional to m, and so are higher for the larger, random dataset. Times to generate p, recover p, and recover bases depend only on b and k, and so are dataset-independent.

Figures 3.3 and 3.5 show that when k is held constant, increasing b costs the client more than it costs the colluding servers. Both prime recovery and modular multiplication take time proportional to b^2, so prime recovery time is a constant factor of share generation time. Further, the time to choose a random b-bit prime using the Miller-Rabin primality test is in $O(b^3)$ [67], so as b grows past 2^{12}, the cost to generate p quickly outstrips the cost to recover it. Thus it is entirely impractical to thwart our attack by increasing b.

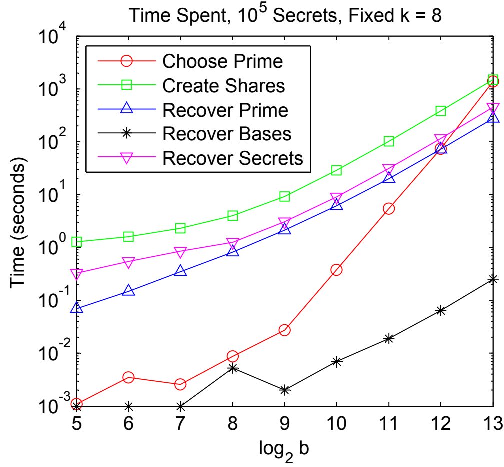

Figure 3.5: Synthetic dataset times, varied b

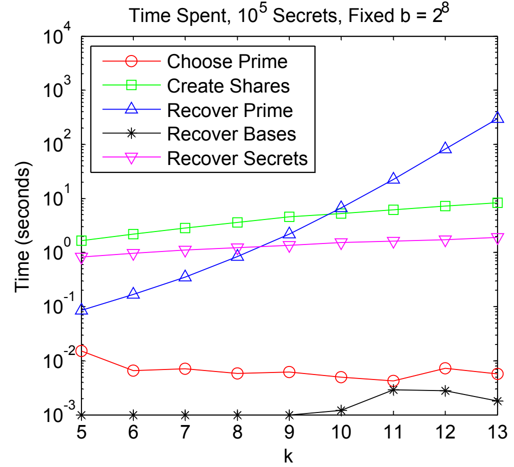

Figure 3.6: Synthetic dataset times, varied k

In the TSWZ scheme [80], the measured time to recover a prime with less than 2^5 bits was over 1500 seconds. In contrast, our method recovers primes with 2^{13} bits in under 500 seconds on comparable hardware, for $k = 8$. As long as $k \ll b$, as is likely in practice, our method is far faster.

Figures 3.4 and 3.6 show that when b is fixed, most times are in $O(k)$, with the exception of prime recovery time, which is in $O(2^{2k})$. Thus, by increasing k, the attack can be made arbitrarily expensive at a relatively small cost to the client. However, as we discuss in Section 3.6, even $k = 10$ may be impractical.

3.5.2 Failure Rate Measurements

Since we only factor out small primes with at most β bits (Section 3.3.2.3), our attack fails if δ_1, δ_2 share any prime factor, other than p, that has more than β bits. Thus, our attack's failure rate r_f is the probability that $\delta_1/p, \delta_2/p$ share a prime with more than β bits. Since δ_1, δ_2 are not independent random numbers, it is difficult to

compute r_f analytically, so we measure it empirically. The results of our experiment are shown in Figure 3.7.

We found that r_f is largely independent of b, but depends heavily on k and β. To measure r_f, we conducted several trials in which we generated a prime p of $b = 32$ bits, and ran our attack using $k + 2$ randomly generated secrets. For $k = 2$, we ran 10^6 trials, and were able to get meaningful failure rates up through $\beta \approx 16$. Trials with larger k were much more expensive, so we only ran 10^3 trials for $k = 6$ and $k = 10$, and the results are accurate only through $\beta \approx 10$.

From our results, we derived the approximate expression $r_f \approx \frac{2^{(k-2)/4}}{2^{\beta+1}}k$. We then plotted this estimated r_f in Figure 3.7, denoted by *est*. The approximation is adequate for the range of β we're considering. The dependence of r_f on $2^{-(\beta+1)}$ is expected, as the probability that a factor of $\beta + 1$ bits found in one random d-bit number is found in another random d-bit number is roughly $\frac{2^{d-(\beta+1)}}{2^d} = 2^{-(\beta+1)}$. The nature of the dependence on k is unclear, but it may be related to the fact that k of the $k + 1$ equations used to compute δ_1 are also used to compute δ_2.

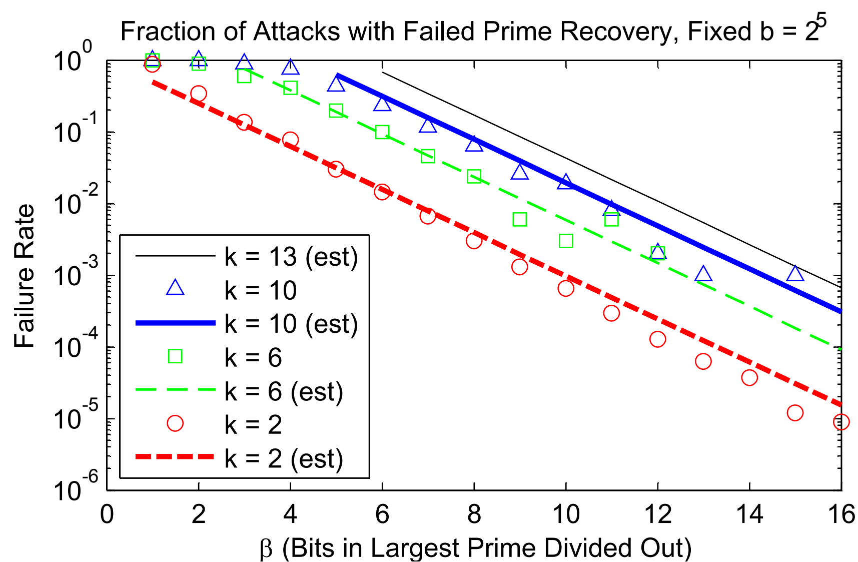

Figure 3.7: Attack failure rates for varied k and β

Using our approximation for r_f, we estimate the worst-case failure rate for our timing experiments, where $\beta = 16$ and $k = 13$, to be $r_f \approx \frac{2^{(13-2)/4}}{2^{16+1}} 13 = 2^{-14.25} 13 \approx 6.67 \times 10^{-4}$. If necessary, we can lower r_f further by increasing β.

3.6 Attack Mitigations

We now discuss possible modifications a client can make to the HAT schemes that may improve security. In order to mitigate our attack, a modification must cause at least one of the attack assumptions listed in Section 3.3 to be violated.

Assumption: At Least k Servers Collude. The simplest way to thwart our attack is to ensure that no more than $k - 1$ servers are able to collude. Only in such cases can secret sharing schemes hope to achieve perfect, information-theoretic security. However, if the number of colluding servers must be limited, secret sharing schemes cannot be applied to the honest-but-curious server threat model commonly used for data outsourcing [8, 40, 47, 60, 77].

Assumption: b, k Modest. In Section 3.5, we showed that increasing b costs the client more than it costs the colluding servers, so a large b is impractical. With limited resources, we successfully mounted attacks for $k = 13$ in under 500 seconds, so k must be substantially larger ($k > 20$) to achieve security in practice. For each server, the client pays a storage cost equal to that of storing his data in plaintext. If $k \geq 10$, the combined storage cost exceeds that of many encryption-based private query techniques [8, 60, 77], so increasing k is also impractical.

Assumption: Same $\vec{X}, p$ for Each Secret. Storing a distinct $\vec{X}$ or prime p on the client for each secret is at least as expensive as storing the secret itself. An alternative is

to use a strong, keyed hash h_j to generate a distinct vector $\vec{X}' = h_j(\vec{X})$ for each secret s_j. Using this method, each secret requires different basis polynomials for interpolation, so mounting an attack would be much harder. Unfortunately, it also eliminates additive homomorphism, removing support for server-side aggregation, which is cited as a reason for adopting secret sharing.

Assumption: Corresponding Shares can be Aligned. Hiding share order from data servers as in [41] can hinder share alignment, but if the scheme supports range or point queries, share alignment can eventually be inferred (Section 3.4.3). Schemes could use re-encryption or shuffling to obscure order as in [77], but the cost of such techniques outweighs the performance advantages of secret sharing.

Assumption: $k + 2$ Known Secrets. It is difficult to keep all secrets hidden from an attacker. Known plaintext/ciphertext attacks for small amounts of data are always a threat, and if we known the real-world distribution of the secrets, we can guess them efficiently (Section 3.4.2). The client could encrypt secrets before sharing, but doing so adds substantial cost and eliminates additive homomorphism.

3.7 Related Work

Privacy-preserving data outsourcing was first formalized in [40] with the introduction of the *Database As a Service* model. Since then, many techniques have been proposed to support private querying [8, 23, 60, 62, 77], most based on specialized encryption techniques. For example, order-preserving encryption [8] supports efficient range queries, while [60] supports server-side aggregation through additively homomor-

phic encryption. Other schemes are based on fragmentation, where only links between sensitive and identifying data are encrypted [23, 62].

As far as we know, the schemes we discuss in this paper [6, 7, 41, 80] are the first to use secret sharing to support private data outsourcing, though secret sharing has been used for related problems, such as cooperative query processing [31]. Prior works, such as [47], have addressed various security issues surrounding data outsourcing schemes, but as far as we know, ours is the first to reveal the specific limitations of schemes based on secret sharing.

3.8 Conclusion

Private data outsourcing schemes based on secret sharing have been advocated because of their slight advantages over existing encryption-based schemes. Such advantages include security, speed, and support for server-side aggregation. All three outsourcing schemes based on secret sharing [6, 41, 80] claim that security is maintained even when k or more servers collude. To the contrary, we have shown that all three schemes are highly insecure when k or more servers collude, regardless of whether $\vec{X}$ and p are kept secret.

We described and implemented an attack that reconstructs all secret data when only $k + 2$ secrets are known initially. In less than 500 seconds, our attack recovers a hidden 256-bit prime for $k \leq 13$ servers, or an 8192-bit prime for $k \leq 8$. We discussed possible modifications to mitigate our attack and improve security, but any such modifications sacrifice generality, performance, or functionality.

We conclude that secret sharing outsourcing schemes are not simultaneously secure and practical in the honest-but-curious server model, where servers are not trusted

to keep data private. Such schemes should only be used when the client is absolutely confident that at most $k-1$ servers can collude.

Chapter 4

Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns

4.1 Introduction

Cloud computing allows customers to outsource the burden of data management and benefit from economy of scale, but privacy concerns hinder its growth [20]. It is well-understood that encryption alone is insufficient to ensure privacy in storage outsourcing applications. Information about the contents of encrypted records may still be leaked, especially via data access patterns. Existing work has shown that access patterns on an encrypted email repository may leak sensitive keyword queries [44], and that access patterns on encrypted database tuples may reveal ordering information [28].

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky [37], is a cryptographic protocol that allows a client to provably hide access patterns from an un-

trusted storage server. Recently, the research community has focused on making ORAM schemes practical for real-world applications [22, 35, 39, 73, 77, 78, 83]. Unfortunately, even with some of the recent improvements [39, 73, 76, 77, 78, 83], ORAMs still incur substantial bandwidth and response time costs.

Most prior works on ORAM focus on minimizing bandwidth consumption. Several recent works on cloud-based ORAMs have shown how to achieve relatively low bandwidth costs given a reasonable amount of client-side storage [39, 76, 77]. Others rely on expensive primitives like PIR [58] or additional assumptions such as trusted hardware [54] or non-colluding servers [75] to reduce client bandwidth costs.

While bandwidth consumption is an important metric for characterizing ORAM costs, to be practical ORAM must also minimize the response time observed by clients for each request. We propose Burst ORAM, a novel ORAM scheme based on the ObliviStore ORAM [76] that dramatically reduces ORAM's response times for realistic workloads with bursty characteristics.

Burst ORAM employs novel techniques, also applicable to other ORAMs, for minimizing the *online* work of serving requests and for delaying the *offline* work of block shuffling until idle periods. Thus we can drastically reduce response times to nearly those of an unprotected block store without ORAM while maintaining security and incurring total bandwidth costs close to those of ObliviStore. Existing works exhibit response times on the order of seconds or higher, due to high bandwidth [39, 76, 78, 87] or computation [58] requirements.

Burst ORAM detects and optimizes for bursts automatically. During long bursts, Burst ORAM's behavior gracefully degrades to be similar to that of ObliviStore. Thus, in a worst-case workload, Burst ORAM's response times and bandwidth costs are competitive with those of existing ORAMs. However, under realistic bursty traffic,

Burst ORAM achieves orders of magnitude shorter response times than existing ORAMs, while incurring only 50% higher overall bandwidth costs.

We use a real-world corporate data access workload (7,500 clients and 15 days) to demonstrate that Burst ORAM can be used practically in a corporate cloud storage environment. We compare against an insecure baseline block store without ORAM and show that when baseline response times are low, Burst ORAM response times are also low. In a 32TB ORAM with 50ms network latency and sufficient bandwidth capacity to ensure 90% of requests have baseline response times under 53ms, 90% of Burst ORAM requests have response times under 63ms. Similarly, with sufficient bandwidths to ensure 99.9% of requests have baseline responses under 70ms, 99.9% of Burst ORAM requests have response times under 76ms. More results are presented in Section 4.9.5. To our knowledge, our work is the first to evaluate ORAM response times on a realistic, bursty workload.

Non-goals As in previous ORAM schemes, we do not seek to hide the timing of data requests. Thus, we assume that block request start times and durations are known.

Maintaining security To ensure security in Burst ORAM, we do not allow the IO scheduler to make use of the data access sequence, or any other sensitive information. In this way, the security of Burst ORAM reduces to that the underlying ORAM ObliviStore [76].

4.1.1 Burst ORAM Contributions

Burst ORAM introduces several techniques for reducing response times and keeping bandwidth costs low that distinguish it from ObliviStore and other predecessors.

Novel scheduling policies Burst ORAM prioritizes the *online* work that must be complete before requests are satisfied. If possible, our scheduler delays shuffle work until off-peak times. Delaying shuffle work consumes client-side storage, so if a burst is sufficiently long, client space will fill, forcing shuffling to resume. By this time, there are typically multiple shuffle jobs pending. We could schedule jobs in order to minimize overall bandwidth consumption or to minimize response times.

We use a greedy strategy that prioritizes those jobs that free up the most client-side space per unit of shuffling bandwidth consumed. This strategy allows us to sustain lower response times for longer during an extended burst, and incurs under 50% additional bandwidth in practice.

Reducing online bandwidth cost We propose a new *XOR technique* that reduces the online bandwidth cost from $O(\log N)$ blocks per request in ObliviStore to nearly 1, where N is the outsourced block count. The XOR technique can also be applied to other ORAM implementations such as SR-ORAM [86] (see Section 4.10).

Level caching We propose a new technique for using additional available client space to store small levels from each partition. By caching these levels on the client, we are able to reduce total bandwidth cost substantially.

Outline Section 4.3 defines terminology and reviews ObliviStore. Section 4.4 provides an overview of our techniques. Section 4.5 discusses prioritizing and reducing online IO, and Section 4.6 covers scheduling and techniques for reducing shuffle IO. Section 4.7 details our system design, and Section 4.9 presents our experimental results.

4.2 Related Work

Oblivious RAM was first proposed in a seminal work by Goldreich and Ostrovsky [37]. Since then, a fair amount of theoretic work has focused on improving its asymptotic performance [11, 25, 38, 38, 39, 49, 64, 65, 73, 77, 83, 84, 85]. Recently, there has been much work toward designing and optimizing ORAM for a cloud-based storage outsourcing setting, as mentioned below. Different ORAMs provide varying tradeoffs between bandwidth cost, client/server storage, round complexity, and in some cases server computation.

ORAM has been shown to be feasible for secure (co-) processor prototypes which prevent information leakage due to physical tampering [34, 54, 55, 68]. In this context, since on-chip trusted cache is expensive, an ORAM scheme with constant or logarithmic client-side storage is needed, such as the binary-tree ORAM [73] and its variants [21, 35, 58, 78].

In cloud-based ORAMs, the client typically has more space, capable of storing $O(\sqrt{N})$ blocks or a small amount of per-block metadata [38, 76, 77, 87] that can be used to further reduce ORAM bandwidth requirements. Burst ORAM also makes such client space assumptions.

Online and offline costs for ORAM were first made explicit by Boneh et al. [11] They propose a construction that has $O(1)$ online but $O(\sqrt{N})$ overall bandwidth cost. The recent Path-PIR work by Mayberry et al. [58] mixes ORAM and PIR to achieve $O(1)$ online bandwidth cost with a modest overall bandwidth cost of $O(\log^2 N)$ with constant client memory. Unfortunately, the PIR is still computationally expensive, so their scheme requires more than 40 seconds for a read from a terabyte-sized database

[58]. Burst ORAM has $O(1)$ online and $O(\log N)$ overall bandwidth cost, without the added overhead of PIR.

Other ORAMs that do not rely on trusted hardware or non-colluding servers have $\Omega(\log n)$ online bandwidth cost, including works by Williams, Sion, and others [83, 85, 87]; by Goodrich, Mitzenmacher, Ohrimenko, and Tamassia [38, 39]; by Kushilevitz et al. [49]; and by Stefonov, Shi, and others [73, 76, 77, 78]. In comparison, Burst ORAM handles bursts much better by reducing the online cost to nearly 1 block transfer per block requested during a burst, substantially reducing response times.

4.3 Preliminaries

4.3.1 Bandwidth Costs

Bandwidth consumption is the primary cost in many modern ORAMs, so it is important to define how we measure different aspects of bandwidth cost. We say that each block transferred between the client and the server is a single unit of IO. We assume that blocks are large in practice (at least 4KB), so in most cases the meta-data exchanged (e.g. block IDs) have negligible size.

Definition 15. The *bandwidth cost* of a storage scheme is given by the average number of blocks transferred in order to read or write a single block.

We identify bandwidth costs by appending X to the number. A bandwidth cost of 2X indicates two blocks fetched per block requested, double that required by an unprotected scheme. We consider *online*, *offline*, *effective*, and *overall* IO and bandwidth costs, where each cost is given by the average amount of the corresponding type of IO.

Online IO consists of the block transfers needed before a request can be safely marked as satisfied, assuming the scheme starts with no pending IO. The *online band-*

width cost of a storage scheme without ORAM is just 1X — the IO cost of downloading the desired block. In ORAMs it may be higher, as additional blocks may be downloaded to hide the requested block's identity.

Offline IO consists of transfers needed to prepare for subsequent requests, but which may be performed after the request is satisfied. Without ORAM, the *offline bandwidth cost* is 0X. In ORAMs it is generally higher, as additional *shuffle IO* is needed to obliviously permute blocks on the server in order to guarantee access pattern privacy for future requests.

Overall IO / bandwidth cost is just the sum of the online and offline IO / bandwidth costs, respectively.

Effective IO consists of all online IO plus any pending offline IO from previous requests that must be issued before this request can be satisfied. Without ORAM, the effective IO and online IO are equal. In traditional ORAMs, offline IO is issued immediately after each request's online IO, so effective and overall IO are equal. In Burst ORAM, we delay some of the offline IO, reducing the effective IO of each request, as illustrated in Figure 4.1. Smaller effective costs mean less IO between requests, and ultimately shorter response times.

In ORAM, by definition, read and write operations are indistinguishable, so bandwidth costs of an ORAM read are the same as those of an ORAM write.

4.3.2 Response Time

The *response time* of a block request (ORAM read/write operation) is defined as the lapse of wall-clock time between when the request is first issued by the client and when the client receives a response. The minimum response time is just the time needed

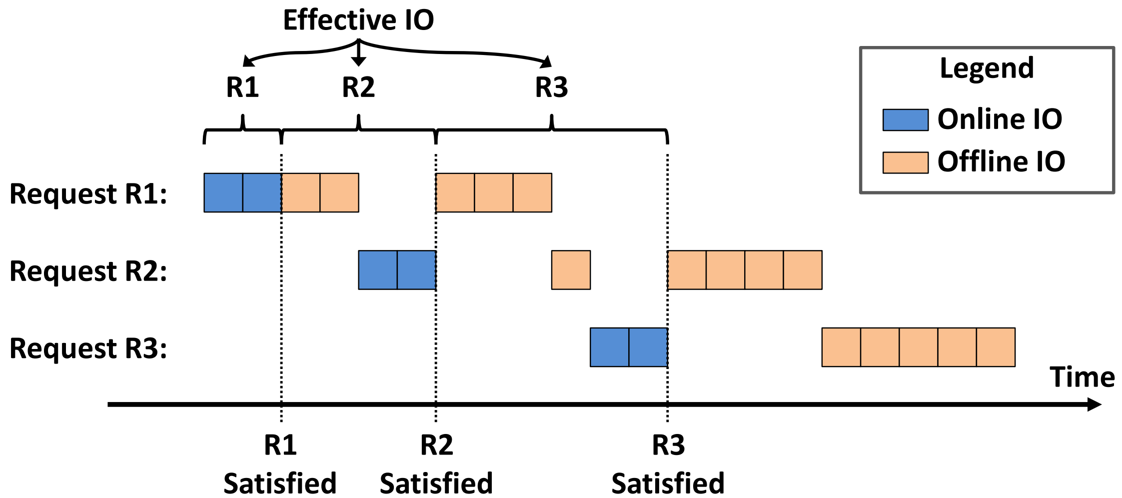

Figure 4.1: **Effective IO.** Simplified scheme with sequential IO and contrived capacity for delaying offline IO. Three requests require same online IO (2), offline IO (5), and overall IO (7). Online IO for R1 can be handled immediately, so R1's effective IO is only 2. R2 must wait for 2 units of offline IO from R1, so its effective IO is 4. R3 must wait for the rest of R1's offline IO, plus one unit of R2's offline IO, so its effective IO is 6.

to perform all online IO for the request. Response times may increase when offline IO is performed between requests, increasing each request's effective IO, or when requests are issued rapidly in a burst, delaying later requests.

4.3.3 ObliviStore ORAM

At IEEE S&P 2013, Stefanov and Shi introduced a cloud-based oblivious storage system named ObliviStore [76]. Burst ORAM builds on ObliviStore, so we give an overview of the scheme here. A full description of the ObliviStore system and its ORAM algorithm spans about 55 pages [76, 77], so we describe it at a high level, focusing only on components relevant to Burst ORAM.

Partitions and levels ObliviStore stores N logical data blocks. Each block is encrypted using a standard symmetric key encryption scheme before it is stored on the

server. Every time a block is uploaded by the client, it is re-encrypted using a new nonce to prevent linking.

ObliviStore securely splits blocks into $O(\sqrt{N})$ *partitions* of $O(\sqrt{N})$ blocks each. Each partition is itself a fully functioning ORAM consisting of $O(\log N)$ *levels* with $\{2, 4, 8, 16, 32, \ldots, O(\sqrt{N})\}$ blocks each. When a level is created, it is filled half with real encrypted blocks and half with dummy encrypted blocks, randomly permuted so that real and dummy blocks are indistinguishable to the server. Each level is occupied only half the time on average. The client has space to store $O(\sqrt{N})$ blocks and the locations of all N blocks.

Requests When the client makes a block request via the ORAM, whether a read or write, the block must first be downloaded from the appropriate server partition. To maintain obliviousness, ObliviStore must fetch one block from every non-empty level in the target partition ($O(\log N)$ blocks of *online* IO). Only one fetched block is real, and the remaining are dummy blocks, except in the case of early shuffle reads described below. Once a dummy block is fetched, it is discarded, and new dummies are created later as necessary. ObliviStore securely processes multiple requests in parallel, enabling full utilization of available bandwidth capacity.

Eviction Once the real block is fetched from the server, it is updated or returned to the client as necessary, then assigned to a new random partition p. The block is not immediately uploaded to the server. Instead, it is scheduled for *eviction* to p and stored in a client-side *data cache*. An independent eviction process later obliviously evicts the block from the cache to p such that the server does not know which blocks were evicted

to which partitions. The eviction triggers a write operation on p's ORAM, which creates or enlarges a *shuffling job* for p.

Shuffling Jobs Each partition p has at most one pending *shuffle job*. A job consists of downloading up to $O(\sqrt{N})$ blocks from p, shuffling them locally, and uploading them back to the server along with recently evicted blocks and new dummy blocks. Shuffle jobs incur *offline* IO, and vary in *size* (amount of IO) from $O(1)$ to $O(\sqrt{N})$. Intuitively, to guarantee that each non-empty level has at least one dummy block remaining, we must re-shuffle a level once half its blocks have been removed. Since larger levels need shuffling less often, larger jobs occur less frequently than small ones, keeping the offline bandwidth cost at $O(\log N)$ on average.

Shuffle IO scheduling In ObliviStore, a fixed amount of $O(\log N)$ shuffle IO is performed after each request in order to amortize the work required for large shuffle jobs. The IO for shuffle jobs from multiple partitions may be executed in parallel: while waiting on reads to complete for one partition, we may issue reads or writes for another. Jobs are chosen in the order they are created, regardless of size.

Early shuffle reads *Early shuffle reads*, referred to as *early cache-ins* or *real cache-ins* in ObliviStore, occur when a request needs to fetch a block from a level, but at least half the level's original blocks have already been removed. Since half the blocks have been removed, we cannot guarantee that additional dummies are present. Thus, early shuffle reads must be treated as real blocks and stored separately by the client until they are returned to the server as part of a shuffle job. We call such reads *early shuffle reads* as the blocks would have eventually been read during a shuffle job. Early shuffle

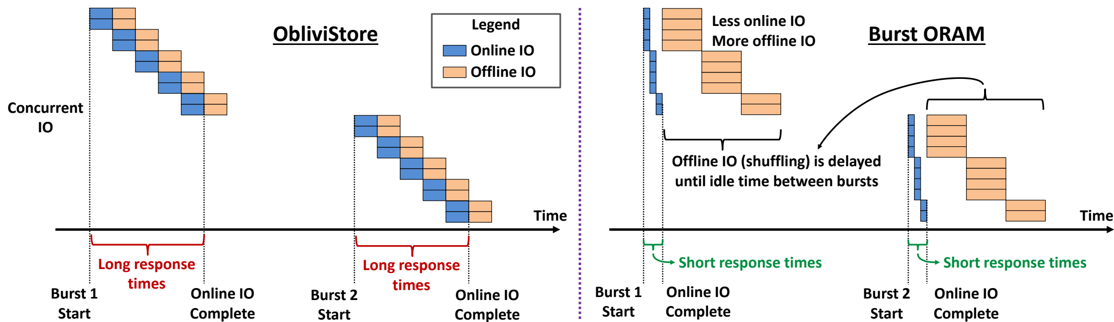

Figure 4.2: **Reducing response time.** Because Burst ORAM (right) does much less online IO than ObliviStore (left) and delays offline IO, it is able to respond to ORAM requests much faster. In this (overfly simplified) illustration, the bandwidth capacity is enough to transfer 4 blocks concurrently. Both ORAM systems do the same amount of IO.

reads are possible since ObliviStore performs requests while shuffling is in progress, but fortunately they are infrequent.

Level compression ObliviStore uses a technique called *level compression* [77] to compress blocks uploaded during shuffling. Level compression allows the client to upload k real and k dummy blocks to the server using only k blocks of bandwidth without revealing which k are dummies. Level compression reduces only the offline shuffling cost, while the online cost remains the same.

4.4 Overview of our Approach

Traditional ORAMs, such as ObliviStore, focus on reducing average and worst-case *overall bandwidth costs*. That is, they seek to minimize the overall IO needed to satisfy each ORAM request. However, even the most bandwidth-efficient schemes still suffer from a 20X to 35X bandwidth cost [76, 77].

In this paper, we take a different approach. We focus on reducing *effective* IO by reducing online IO and delaying offline IO. We can then satisfy bursts of ORAM

requests quickly, delaying most IO until idle times between bursts. Figure 4.2 illustrates this concept.

Our approach is so successful that many bursts can be satisfied with an effective bandwidth cost of nearly 1X, using little more IO than a storage scheme without ORAM. That is, during the burst, we transfer just over one block on average for every block requested. After the burst we do extra IO to catch up on shuffling and prepare for future requests. Our approach maintains an overall bandwidth cost competitive with [76, 77], less than 50% higher in practice (see Figure 4.12 in Section 4.9).

Bursts Intuitively, a burst is a period of frequent block requests from the client preceded and followed by relatively idle periods. Many real-world workloads exhibit bursty patterns (e.g. [18, 50]). Often, bursts are not discrete events, such as when multiple users of a network file system are operating concurrently. Thus we treat periods of time as more or less bursty: the more requests issued at a given time, the more Burst ORAM tries to delay offline IO until idle periods.

Challenges We are faced with two key challenges when building a burst-friendly ORAM system that reduces online IO and delays offline IO. The first is ensuring that we maintain security. A naive approach to reducing online IO may mark requests as satisfied before enough blocks are read from the server, leaking information about the requested block's identity.

The second challenge is ensuring that we maximally utilize client memory and available bandwidth while avoiding deadlock. An excessively aggressive strategy, that delays too much offline IO too long, may use so much client space that we run out of room to shuffle. It may also under-utilize available bandwidth, causing increased

response times. On the other hand, an overly conservative strategy may under-utilize client space or perform shuffling too early, delaying online IO and causing unnecessary increases in response times.

Techniques and Outline In Burst ORAM, we address the challenges described above by combining several novel techniques. In Section 4.5 we introduce our XOR technique for reducing the online bandwidth to nearly 1X. We also describe our techniques for prioritizing online IO and delaying offline/shuffle IO until client memory is nearly full. In Section 4.6 we show how Burst ORAM prioritizes efficient shuffle jobs in order to delay the bulk of the shuffle IO even further, ensuring that we minimize effective IO during long bursts. We then introduce a technique for using available client space to cache small levels locally in order to reduce shuffle IO in both Burst ORAM and ObliviStore.

In Section 4.7 we discuss the system-level techniques required to construct Burst ORAM, and present its design in more detail. In Section 4.9, we evaluate Burst ORAM's performance through micro-benchmarks and extensive simulations on a large corporate file system trace.

4.5 Prioritizing and Reducing Online IO

Existing ORAM schemes tend to require high online and offline bandwidth costs in order to obscure access patterns. For example, to satisfy each request ObliviStore must fetch one block from every level in a partition (see Section 4.3.3), requiring $O(\log N)$ online IO per request. The left side of Figure 4.3 illustrates this behavior. After each request, ObliviStore also requires $O(\log N)$ offline/shuffle IO to prepare for future requests. Since ObliviStore must issue online and offline IO before satisfying the next request, its effective IO is high, leading to large response times during long bursts.

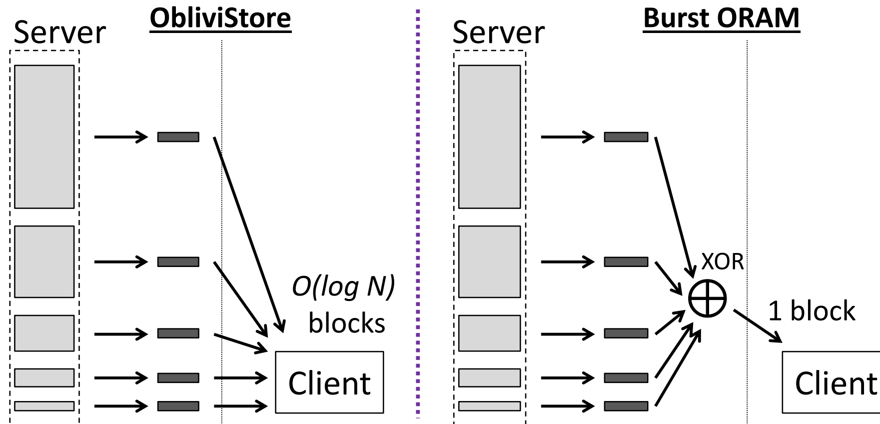

Figure 4.3: **Reducing online cost.** In ObliviStore (left) the online bandwidth cost is $O(\log N)$ blocks of IO on average. In Burst ORAM (right), we reduce online IO to only one block, improving handling of bursty traffic.

Other ORAM schemes work differently, such as Path ORAM [78] which organizes data as a tree instead of a level hierarchy, but still have high effective bandwidth costs.

We now show how Burst ORAM achieves lower effective bandwidth costs and lower response times than ObliviStore without compromising security.

4.5.1 Prioritizing Online IO

One way we achieve low response times in Burst ORAM is by prioritizing online IO over shuffle IO. That is, we suppress shuffle IO during bursts, delaying it until the idle time between bursts. Requests are satisfied once their online IO finishes,[1] so prioritizing online IO allows us to satisfy all requests before any shuffle IO starts, keeping response times low even for later requests during a burst. Figure 4.2 illustrates this behavior.

During the burst, we continue processing requests by fetching blocks from the server, but since shuffling is suppressed, no blocks are stored back to the server. Thus, we must resume shuffling once client storage fills. Section 4.6.2 discusses how we can delay the bulk of the shuffle IO even further. Prioritizing online IO requires fundamental

[1] Each client write also incurs a read, so writes still incur online IO.

changes from the ObliviStore design in order to avoid deadlock while fully utilizing client space. Details are given in Section 4.7.

When available bandwidths are large and bursts are short, the response time saved by prioritizing online IO is limited, as most IO needed for the burst can be issued in parallel. However, when bandwidth is limited or bursts are long, the savings can be substantial. With shuffle IO delayed until idle times, online IO dominates the effective IO, becoming the bottleneck during bursts. Thus we can further reduce response times by reducing online IO.

4.5.2 XOR Technique: Reducing Online IO

We introduce a new mechanism called the *XOR technique* that allows the Burst ORAM server to combine the $O(\log N)$ blocks fetched during a request into a single block that is returned to the client (Figure 4.3 right), reducing the online bandwidth cost to $O(1)$. If we fetched only the desired block, we would reveal its identity to the server. Instead, we XOR all the blocks together and return the result. Since there is at most one real block among the $O(\log N)$ returned, the client can locally derive the dummy block values and XOR them with the returned block to recover the encrypted real block. Of course, we must ensure that the dummy block contents are entirely predictable by the client. The basic steps of the XOR technique are shown in Figure 4.4.

4.5.2.1 XOR Technique Details

In Burst ORAM, as in ObliviStore, each request needs to retrieve a block from a single *partition*, which is a simplified hierarchical ORAM resembling those originally proposed by Goldreich and Ostrovsky [37]. The hierarchy contains $L \approx \frac{1}{2}\log_2 N$ levels of capacity $1, 2, 4, 8, \ldots, 2^{L-1}$ respectively.

1. Client issues block requests to server, one from each level
2. Server, to satisfy request
   (a) Retrieves all blocks from disk and returns early shuffle reads
   (b) XORs remaining blocks together into single *compressed* block and return it
3. Client, while waiting for response
   (a) Looks up level-specific nonce for each (non-early-shuffle-read) dummy block
   (b) Hashes each dummy block's nonce with its level position to reconstruct contents
   (c) XORs all dummy blocks together into *subtraction* block
4. Client receives compressed block from client and XORs with subtraction block to get requested block
5. Client decrypts requested block

Figure 4.4: XOR Technique Steps

To retrieve a desired block from the partition, the client needs to fetch exactly one block at a pseudorandom location from each of the L levels. To use the XOR technique, we must ensure that the client can reconstruct dummy blocks, while ensuring that dummy blocks are indistinguishable from real blocks to the server. We achieve this property by encrypting a real block b residing in partition p, level ℓ, and offset *off* as $\mathsf{AES}_{sk_{p,\ell}}(\mathit{off}||B)$. We encrypt a dummy block residing in partition p, level ℓ, and offset *off* as $\mathsf{AES}_{sk_{p,\ell}}(\mathit{off})$. The key $sk_{p,\ell}$ is specific to partition p and level ℓ, and is randomized every time ℓ is rebuilt.

For simplicity, we start by considering the case without early shuffle reads. In this case, exactly one of the L blocks requested is the encryption of a real block, and the rest are encryptions of dummy blocks. The server XORs all L encrypted blocks together into a single block X_Q that it returns to the client. The client knows which blocks are dummies, and knows p, ℓ, off for each block, so it reconstructs all the encrypted dummy blocks and XORs them with X_Q to obtain the encrypted requested/real block.

4.5.2.2 Handling early shuffle reads

An early shuffle read occurs when we need to read from a level with no more than half its original blocks remaining. Since such early shuffle reads may be real blocks, they cannot be included in the XOR. Fortunately, the number of blocks in a level is public, so the server already knows which levels will cause early shuffle reads. Thus, the server simply returns early shuffle reads individually, then XORs the remaining blocks together and returns them to the client, and no information is leaked about the access sequence.

Since each early shuffle read block must be transferred individually, early shuffle reads increase online IO. Fortunately, early shuffle reads are relatively uncommon, even while shuffling is suppressed during bursts, so the online bandwidth cost stays under 2X and close to 1X in most cases (see Figure 4.7 in Section 4.9).

4.5.2.3 Comparison with ObliviStore

ObliviStore uses *level compression* to reduce shuffle IO. When the client uploads a level to the server during shuffling, it first compresses the level down to just over the combined size of the level's real blocks. Since half the blocks are dummies, nearly half of the upload shuffle IO is eliminated. For details on level compression, including an explanation for why it is secure, see [76].

Unfortunately, Burst ORAM's XOR technique is incompatible with level compression due to discrepancies in the ways dummy blocks must be formed. The XOR technique requires that the client be able to reconstruct dummy blocks locally in order to XOR them out of the returned block, so in Burst ORAM, each dummy block's position determines its contents. In level compression, each level's dummy block contents

must be a function of the level's real block contents. Since the client cannot know the contents of all real blocks in the level, it cannot reconstruct the dummy blocks locally.

Level compression and the XOR technique yield comparable overall IO reductions, though level compression performs slightly better. For example, the experiment in Figure 4.8 incurs roughly 23X and 26X overall bandwidth cost using level compression and the XOR technique respectively. However, the XOR technique reduces *online* IO, while level compression reduces *offline* IO. In Burst ORAM, online IO is the bottleneck, since offline IO can be delayed, so the XOR technique is far more effective at reducing response times during bursts.

4.6 Scheduling and Reducing Shuffle IO

In Section 4.5, we showed how Burst ORAM prioritizes and reduces online IO, keeping effective bandwidth costs low during short bursts by delaying shuffling. Once client space fills, we must start shuffling in order to return blocks to the server and continue the burst. However, if we are not careful about shuffle IO scheduling, we may immediately start doing large amounts of IO, dramatically increasing response times.

In this section, we show how Burst ORAM schedules shuffle IO such that jobs that free the most client space using the least shuffle IO are prioritized. Thus, at all times, Burst ORAM issues only the minimum amount of effective IO needed to continue the burst, keeping response times lower for longer. We also show how to reduce overall IO by locally caching the smallest levels from each partition. We start with a definition of *shuffle jobs*.

4.6.1 Shuffle Jobs

In Burst ORAM, as in ObliviStore, shuffle IO is divided into per-partition *shuffle jobs*. Each job represents the work needed to shuffle a partition and return, from client to server, all blocks previously evicted to that partition. A shuffle job is defined by five entities:

- A partition p
- Blocks evicted to but not yet returned to p
- Levels to read blocks from
- Levels to write blocks to
- Blocks already read from p (early shuffle reads)

Each shuffle job moves through three phases:

Creation A shuffle job for p is created when a block is evicted to p after some request. Every shuffle job starts out *inactive*, meaning Burst ORAM is not yet working on it. When another block is evicted to p, we update the sets of eviction blocks and read/write levels in p's inactive shuffle job.

When Burst ORAM *activates* a shuffle job, it marks the job *active* and moves it to the *Read Phase*, freezing the eviction blocks and read/write levels. Subsequent evictions to p will create a new *inactive* shuffle job. At any time, there is at most one active and one inactive shuffle job for each partition.

Read Phase Once a shuffle job is activated and moved to the *Read Phase*, Burst ORAM begins fetching all blocks still on the server that need to be shuffled. That is, all previously unread blocks from all the job's read levels. Once all such blocks are fetched, they are *shuffled* with all blocks evicted to p and any early shuffle reads from the read

levels. Shuffling consists of adding/removing dummies, pseudo-randomly permuting the blocks, and then re-encrypting each block. Once shuffling completes, the job moves to the *Write Phase*.

Write Phase Once a job is shuffled and moved to the *Write Phase*, Burst ORAM begins storing all shuffled blocks to the job's write levels on the server. Once all the writes finish, the job is complete, and Burst ORAM is free to activate p's inactive job, if any.

4.6.2 Prioritizing Efficient Jobs

In Burst ORAM, when client space fills, we are forced to start shuffling in order to continue handling requests. Since executing shuffle IO delays the online IO needed to satisfy requests, we can reduce response times by always doing as little shuffling as is needed to free up space. The hope is that we can delay the bulk of the shuffling until an idle period, so that it does not interfere with pending requests.

By the time client space fills, there will be many partitions with inactive shuffle jobs. Since we can choose jobs in any order, we can minimize the up-front shuffling work by prioritizing the most *efficient* shuffle jobs: those that free up the most client space per unit of shuffle IO. The space freed by completing a job for partition p is the number of blocks evicted to p plus the number of early shuffle reads from the job's read levels. Thus, we can define shuffle job efficiency as follows:

$$\text{Job Efficiency} = \frac{\#\text{ Evictions} + \#\text{ Early Shuffle Reads}}{\#\text{ Blocks to Read} + \#\text{ Blocks to Write}}$$

Job efficiencies vary substantially. For example, most jobs start out with 1 eviction and 0 early shuffle reads, so their relative efficiencies are determined strictly by IO, which is in turn determined by the sizes of the job's read and write levels. If the

partition's bottom level is empty, no levels need be read, and only the bottom must be written, for an overall IO of 2 an an efficiency of 0.5. If instead the bottom 4 levels are occupied, all 4 levels must be read, and the 5th level written, for an total of roughly 15 reads and 32 writes, yielding a much lower efficiency of just over 0.02. Both jobs free equal amounts of space, but the higher-efficiency job uses less IO.

Since small levels are written more often than large ones, efficient jobs are common. Further, by delaying an unusually inefficient job, we give it time to accumulate more evictions. While such a job will also accumulate more IO, the added write levels are generally small, so the job's efficiency tends to improve with time. Thus, by prioritizing efficient jobs, we reduce the shuffle IO needed during the burst, thereby reducing effective IO and ultimately response times.

Unlike Burst ORAM, ObliviStore does not use client space to delay shuffling, so there are fewer shuffle jobs to choose from at any one time. Thus, job scheduling is less important and jobs are chosen in creation order. Since ObliviStore is concerned with throughput, not response times, it has no incentive to prioritize efficient jobs.

4.6.3 Reducing Shuffle IO via Level Caching

We have shown how Burst ORAM uses client space to delay shuffling during bursts, reducing response times. We now show how we could instead use that same space to reduce shuffle IO.

Since small, efficient shuffle jobs are common, Burst ORAM spends a lot of time accessing small levels. If we use client space to locally cache the smallest levels of each partition, we can eliminate the shuffle IO associated with those levels entirely. Since levels are shuffled with a frequency inversely proportional to their size, each is responsible for roughly the same fraction of shuffle IO. Thus, even if we can cache only

a few levels from each partition, shuffle IO savings can be substantial. Further, since caching a level eliminates its early shuffle reads, and early shuffle reads are common for small levels, caching the smallest levels can also reduce online IO.

We are therefore faced with a tradeoff between using space to store requested blocks, which reduces response times for short bursts, and using it for local level caching, which reduces overall bandwidth cost.

4.6.3.1 Level Caching in Burst ORAM

In Burst ORAM, we take a conservative approach, and cache only as many levels as are guaranteed to fit in the worst case. More precisely, we identify the maximum number λ such that the client could store all real blocks from the smallest λ levels of every partition even if all were full simultaneously. We cache levels by only updating an inactive job when the number of evictions is such that all the job's write levels have index at least λ.

Since each level is only occupied half the time, caching λ levels consumes at most half of the client's space on average, leaving the other half for storing blocks downloaded by online requests. As we show experimentally in Section 4.9, level caching substantially reduces overall bandwidth cost. Further, in most cases, response times are actually lower with level caching than without it, due to fewer early shuffle reads.

4.7 Detailed Burst ORAM Design

The Burst ORAM design is based on ObliviStore, but incorporates many fundamental functional and system-level changes. For example, Burst ORAM replaces or revises all the semaphores used in ObliviStore to achieve our distinct goal of online

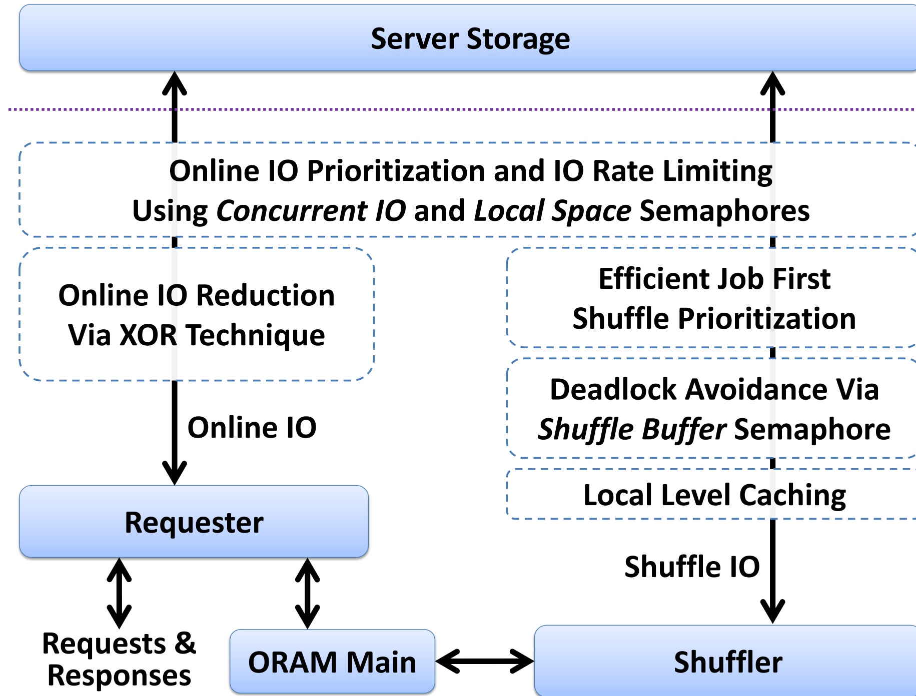

Figure 4.5: **Burst ORAM Architecture.** Solid boxes represent key system components, while dashed boxes represent functionality and the effects of the system on IO.

IO prioritization while maintaining security and avoiding deadlock. Burst ORAM also maximizes client space utilization, implements the XOR technique to reduce online IO, revises the shuffler to schedule efficient jobs first, and implements level caching to reduce overall IO. We describe the Burst ORAM design in detail below, comparing and contrasting with ObliviStore where appropriate.

4.7.1 Overall Architecture

Figure 4.5 presents the basic architecture of Burst ORAM, highlighting key components and functionality. Burst ORAM consists of two primary components, the online *Requester* and the offline *Shuffler*, which are controlled by the main event loop *ORAM Main*. Client-side memory allocation is shown in Figure 4.6.

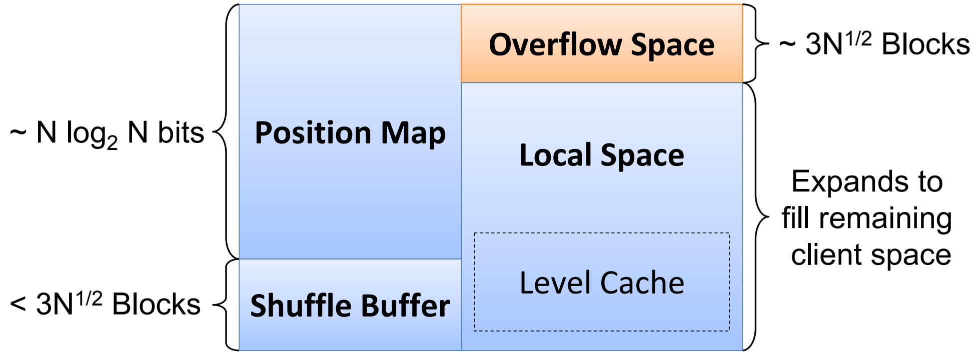

Figure 4.6: **Burst ORAM Client Space Allocation.** Fixed client space is reserved for the position map and shuffle buffer. A small amount of overflow space is needed for blocks assigned but not yet evicted to partitions (*data cache* in [77]). Remaining client space is managed by *Local Space* semaphore and contains evictions, early shuffle reads, and the level cache.

ORAM Main accepts new block requests (reads and writes) from the user, and adds them to a *Request Queue*. On each iteration, ORAM Main tries to advance the Requester first, only advancing the Shuffler if the Requester has no IO to perform, thereby prioritizing online IO over shuffle IO. The Requester and Shuffler utilize shared *semaphores* (Section 4.7.2) that regulate access to network bandwidth and client space.

The *Requester* reads each request from the Request Queue, identifies the desired block's partition, and fetches it along with any necessary dummy blocks: one block from each level in the partition. To ensure oblivious behavior, the Requester must wait until all dummy blocks have been fetched before marking the request satisfied. It then updates the desired block (for writes) or returns it to the user (for reads). All IO initiated by the requester is considered online IO.

The *Shuffler* ensures that each block fetched by the Requester is re-encrypted, shuffled with other blocks from its target partition, and returned to the server. The Shuffler is responsible for managing shuffle jobs, including prioritizing efficient jobs and implementing level caching. All IO initiated by the shuffler is considered *offline* or *shuffle* IO.

4.7.2 Semaphores

Resources in Burst ORAM are managed via *semaphores*, as in ObliviStore. Semaphores are updated using only server-visible information, so ORAM can safely base its behavior on semaphores without revealing new information. Since Burst ORAM gives online IO strict priority over shuffle IO, our use of semaphores is substantially different than ObliviStore's, which tries to issue the same amount of IO after each request. ObliviStore uses four semaphores: *Shuffling Buffer*, *Early Cache-ins*, *Eviction*, and *Shuffling IO*. In Burst ORAM, we use three:

- *Shuffle Buffer* manages the client space reserved for blocks from active shuffle jobs, and differs from ObliviStore's *Shuffling Buffer* semaphore only in initial value.
- *Local Space* manages all remaining space, combining ObliviStore's *Early Cache-in* and *Eviction* semaphores.
- *Concurrent IO* manages concurrent block transfers based on network link capacity, preventing the Shuffler from starving the Requester. It differs fundamentally from ObliviStore's *Shuffling IO* semaphore, which manages per-request shuffle IO.

***Shuffle Buffer* semaphore** The *Shuffle Buffer* semaphore gives the number of blocks that may be added to the client's shuffle buffer. We initialize it to double the maximum partition size (under $2.4\sqrt{N}$ total for $N > 2^{10}$), to ensure that the shuffle buffer is large enough to store at least two in-progress shuffle jobs. When *Shuffle Buffer* reaches 0, the Shuffler may not issue additional reads.

***Local Space* semaphore** The *Local Space* semaphore gives the number of blocks that may still be stored in remaining client space (space not reserved for the position map or shuffle buffer). If *Local Space* is 0, the Requester may not fetch more blocks. Blocks

fetched by the Requester count toward *Local Space* until their partition's shuffle job is activated and they are absorbed into *Shuffle Buffer*. Once a block moves from *Local Space* to *Shuffle Buffer*, it is considered *free* from the client, and more requests may be issued. The more client space, the higher *Local Space*'s initial value, and the better burst and long-term performance.

***Concurrent IO* semaphore** The *Concurrent IO* semaphore is initialized to the network link's block capacity. Every time a block transfer is queued, *Concurrent IO* is decremented. Every time a transfer completes, *Concurrent IO* is incremented. The Shuffler may only initiate a transfer if *Concurrent IO* > 0. However, the Requester may continue to initiate transfers and decrement *Concurrent IO* even if it is negative. This mechanism ensures that no new shuffle IO can start while there is a sufficient amount of online IO to keep the link fully utilized. If no new online IO is started, *Concurrent IO* eventually becomes positive, and shuffle IO resumes, ensuring full link utilization.

4.7.3 Detailed System Behavior

We now describe the interaction between ORAM Main, the Requester, the Shuffler, and the semaphores in detail. Accompanying pseudocode can be found in Section 4.8.

ORAM Main (Algorithm 2) Incoming read and write requests are asynchronously added to the Request Queue. During each iteration, ORAM Main first tries to advance the Requester, which attempts to satisfy the next request from the Request Queue. If the queue is empty, or *Local Space* too low, ORAM Main advances the Shuffler instead. This mechanism suppresses new shuffle IO during a new burst of requests until the Requester has fetched as many blocks as possible.

For each request, we evict ϵ blocks to randomly chosen partitions, where ϵ is the *eviction rate*, set to 1.3 as in ObliviStore [76]. When evicting, if the Requester has previously assigned a block to be evicted to partition p, then we evict that block. If there are no assigned blocks, then to maintain obliviousness we evict a new dummy block instead. Eviction does not send a block to the server immediately. It merely informs the Shuffler that the block is ready to be shuffled into p.

Requester (Algorithm 3) To service a request, the Requester first identifies the partition and level containing the desired block. It then determines which levels require early shuffle reads, and which need only standard reads. If the *Local Space* semaphore is large enough to accommodate the retrieved blocks, the requester issues an asynchronous request for the necessary blocks from each level. Else, control returns to ORAM Main, giving the Shuffler a chance to free space.

The server asynchronously returns the early shuffle read blocks and a single *subtraction* block obtained from all standard-read blocks using the XOR technique (Section 4.5). The Requester extracts the desired block from the subtraction block or from an early shuffle read block, then updates the block (write) or returns it to the client (read). The Requester then assigns the desired block for eviction to a randomly chosen partition.

Shuffler (Algorithm 4) The Shuffler may only proceed if *Concurrent IO* > 0. Otherwise, there is pending online IO, which takes priority over shuffle IO, so control returns to ORAM Main without any shuffling.

The Shuffler places shuffle jobs into three queues based on phase. The *New Job Queue* holds inactive jobs, prioritized by efficiency. The *Read Job Queue* holds active

jobs for which some reads have been issued, but not all reads are complete. The *Write Job Queue* holds active jobs for which all reads, not writes, are complete.

If all reads have been issued for all jobs in the *Read Job Queue*, the Shuffler *activates* the most efficient job from the *New Job Queue*, if any. *Activating* a job moves it to the *Read Job Queue* and freezes its read/write levels, preventing it from being updated by subsequent evictions. It also moves the job's eviction and early shuffle read blocks from *Local Space* to *Shuffle Buffer*, freeing up *Local Space* to handle online requests. By ensuring that all reads for all active jobs are issued before activating new jobs, we prevent the Scheduler from hastily activating inefficient jobs.

The shuffler then tries to decrement *Shuffle Buffer* to determine whether a shuffle read may be issued. If so, the Shuffler asynchronously fetches a block for a job in the *Read Job Queue*. If not, the Shuffler asynchronously writes a block from a job in the *Write Job Queue* instead. Unlike reads, writes do not require *Shuffle Buffer* space, so they can always be issued. The Shuffler prioritizes reads since they are critical prerequisites to activating new jobs and freeing up *Local Space*. The equally costly writes can be delayed until *Shuffle Buffer* space runs out.

Once all reads for a job complete, the job is *shuffled*: dummy blocks are added as necessary, then all blocks are re-encrypted and permuted. The shuffler then moves the job to the *Write Job Queue*. When all writes for a job finish, the job is marked complete and removed from the *Write Job Queue*.

Table 4.1: Algorithm Notation

ϵ	Eviction rate: blocks evicted per request
λ	Number of levels cached locally
p	A partition
V_p	# blocks evicted to p since last shuffle of p finished
C_p	State of p after last shuffle (total shuffled evictions)
b	Block ID
$D(b)$	Plaintext contents of b
$E(b)$	Encrypted contents of b
$S(b)$	Server address/ID of b
$P(b)$	Partition containing b, or random partition if none
$L(b)$	Level containing b, or $\perp$ if none
Q	IDs of standard-read blocks to fetch
C	IDs of early shuffle read blocks to fetch
X_Q	Subtraction block (XOR of blocks in Q)
J_p	Shuffle job for p
V_{J_p}	Number of evicted blocks J_p will shuffle
E_{J_p}	Efficiency of J_p
A_{J_p}	Number of early shuffle reads for J_p
R_{J_p}	Total blocks remaining to be read for J_p
W_{J_p}	Total blocks to write for J_p
NJQ	New Job Queue
RJQ	Read Job Queue
WJQ	Write Job Queue

4.8 Pseudocode

Algorithms 2–5 give pseudocode for Burst ORAM, using the notation summarized in Table 4.1. The algorithms are described in detail in Section 4.7, but we clarify some of the code and notation below.

The efficiency of shuffle job J_p is given by:

$$E_{J_p} = \frac{V_{J_p} + A_{J_p}}{R_{J_p} + W_{J_p}}. \tag{4.1}$$

C_p represents the state of partition p at the time p's last shuffle job completed, and determines the current set of occupied levels in p. V_p represents the number of blocks that have been evicted to p, since p's last shuffle job completed. $C_p + V_p$ determines which levels would be occupied if p were to be completely shuffled.

Algorithm 2 Pseudocode for Client and ORAM Main

```
 1: function ClientRead(b)
 2:     Append b to RequestQueue
 3:     On RequestCallback(D(b)), return D(b)
 4: procedure Write(b, d)
 5:     Append b to RequestQueue
 6:     On RequestCallback(D(b)), write d to D(b)
 7: procedure ORAM Main
 8:     RequestMade ← false
 9:     if RequestQueue ≠ ∅ then
10:         b ← Peek(RequestQueue)
11:         if Fetch(b) then                          ▷ Request Successfully Issued
12:             RequestMade ← true
13:             Pop(RequestQueue)
14:             MakeEvictions()
15:     if RequestMade = false then
16:         TryShuffleWork()
17: procedure MakeEvictions
18:     PendingEvictions = PendingEvictions + ϵ
19:     while PendingEvictions ≥ 1 do
20:         p ← random partition
21:         Evict new dummy or assigned real block to p
22:         V_p = V_p + 1
23:         if shuffling p would only write levels ≥ λ then
24:             J_p ← p's inactive job                  ▷ Create if necessary
25:             V_{J_p} ← V_p
26:             if p has no active job then
27:                 NJQ = NJQ ∪ J_p
28:         PendingEvictions = PendingEvictions − 1
```

V_{J_p} represents the number of evicted blocks that will be shuffled into p by job J_p. Thus, C_p and V_{J_p} together determine the levels that will be read and written when J_p is shuffled.

If J_p is inactive, it is updated whenever V_p changes, setting $V_{J_p} \leftarrow V_p$ (Algorithm 2, Line 25). However, we implement level caching by skipping those updates to J_p that would cause J_p to write to levels with indexes less than λ (Algorithm 2, Line 23). Once J_p is activated, V_{J_p} is no longer updated. When J_p completes, p's state is updated to reflect the blocks shuffled in by J_p, setting $C_p \leftarrow C_p + V_{J_p}$ (Algorithm 4, Line 37).

If p has no inactive shuffle job, the job is created after the first eviction to p that permits updating (Algorithm 2, Line 24). If p has no active job, the inactive job moves to the *New Job Queue* (NJQ) as soon as the job is created (Algorithm 2, Line 27), where it stays until the job is activated. If p does have an active shuffle job, the inactive job is not added to NJQ until the active job completes (Algorithm 4, Line 38).

Thus, NJQ contains only inactive shuffle jobs for those partitions with no active job, ensuring that any job in NJQ may be activated. NJQ is a priority queue serving the most efficient jobs first. Job efficiency may change while the job is in NJQ, since V_{J_p} can still be updated.

4.9 Evaluation

We ran simulations to compare Burst ORAM's response times and bandwidth costs with those of ObliviStore and an insecure baseline without ORAM using real and synthetic workloads.

4.9.1 Methodology

4.9.1.1 Baselines

We compare Burst ORAM and its variants against two baselines. The first is the ObliviStore ORAM described in [76], including its level compression optimization. For fairness, we allow ObliviStore to use extra client space to locally cache the smallest levels in each partition.

The second baseline is an insecure scheme without ORAM in which blocks are encrypted, but access patterns are not hidden. This scheme transfers exactly one

Algorithm 3 Pseudocode for Requester

```
1:  function FETCH(b)
2:      P(b), L(b) ← position map lookup on b
3:      Q = ∅, C = ∅
4:      for level ℓ ∈ P(b) do
5:          if ℓ is non-empty then
6:              b_ℓ ← b if ℓ = L(b)
7:              b_ℓ ← ID of next dummy in ℓ if ℓ ≠ L(b)
8:              if ℓ more than half full then
9:                  Q ← Q ∪ S(b_ℓ)                          ▷ Standard read
10:             else
11:                 C ← C ∪ S(b_ℓ)                ▷ Early shuffle read (rare)
12:     Ret ← |C|+ MAX(|Q|, 1)                     ▷ Num. blocks to return
13:     if Not TRYDEC(Local Space, Ret) then
14:         return false                  ▷ Not enough space for blocks
15:     DEC(Concurrent IO, Ret)
16:     Issue asynch. request for (C, Q) to server
17:     When done, server calls:
18:         FETCHCALLBACK(early shuffle reads, XOR block)
19:     return true
20: procedure FETCHCALLBACK({E(c_i)}, X_Q)
21:     INC(Concurrent IO, 1)
22:     if b ∈ Q then
23:         X'_Q ← ⊕{E(q_i) | S(q_i) ∈ Q, q_i ≠ b}
24:                                  ▷ Subtraction block, computed locally
25:         E(b) ← X_Q ⊕ X'_Q
26:     if b ∈ C then
27:         E(b) ← E(c_i) where c_i = b
28:     D(b) ← decrypt E(b)
29:     Assign b for eviction to random partition
30:     REQUESTCALLBACK(D(b))
```

block per request. We did not include results from Path-PIR [58] because it requires substantially larger block sizes to be efficient, and its response times are dominated by the orthogonal consideration of PIR computation. Path-PIR reports response times in the 40–50 second range for comparably-sized databases.

4.9.1.2 Metrics

We evaluate Burst ORAM and our baselines using *response time* and *bandwidth cost* as metrics (see Section 4.3). We measure average, maximum, and p-percentile

Algorithm 4 Pseudocode for Shuffler

```
1: procedure TRYSHUFFLEWORK
2:     if Not TRYDEC(Concurrent IO, 1) then
3:         return
4:     ReadIssued, WriteIssued ← false
5:     if All reads for all jobs in RJQ have been issued then
6:         TRYACTIVATE()                                    ▷ Try to add job to RJQ
7:     if J_p ∈ RJQ has not issued read b_R then
8:         if TRYDEC(Shuffle Buffer, 1) then
9:             Issue asynch. request for S(b_R) from server
10:            When done, call READCALLBACK(E(b_R))
11:            ReadIssued ← true
12:    if !ReadIssued and J_p ∈ WJQ has write b_W then
13:        Write E(b_W) to server
14:        When done, call WRITECALLBACK(S(b_W))
15:        WriteIssued ← true
16:    if Not ReadIssued and Not WriteIssued then
17:        INC(Concurrent IO, 1)                            ▷ No shuffle work needed
18: procedure TRYACTIVATE
19:    if NJQ ≠ ∅ then
20:        J_p ← PEEK(NJQ)                                  ▷ Most efficient job
21:        if TRYDEC(Shuffle Buffer, V_{J_p} + A_{J_p}) then
22:            Mark J_p active                              ▷ V_{J_p} will no longer change
23:            INC(Local Space, V_{J_p} + A_{J_p})
24:            Move J_p from NJQ to RJQ
25: procedure READCALLBACK(E(b_R))
26:    INC(Concurrent IO, 1)
27:    Decrypt E(b_R), place D(b_R) in Shuffle Buffer
28:    if all reads in J_p have finished then
29:        Create dummy blocks to get W_{J_p} blocks total
30:        Permute and re-encrypt the blocks
31:        Move J_p from RJQ to WJQ
32: procedure WRITECALLBACK(S(b_W))
33:    INC(Concurrent IO, 1)
34:    if all writes in J_p have finished then
35:        Mark J_p complete
36:        Remove J_p from WJQ
37:        Update C_p ← C_p + V_{J_p}, V_p ← V_p − V_{J_p}
38:        Add p's inactive job, if any, to NJQ
```

response times for various p. A p-percentile response time of t indicates that p-percent of the requests have response times under t seconds.

We explicitly measure the online, effective, and overall bandwidth costs. In the insecure baseline, all three costs are 1X, so response times are minimal. However, if

Algorithm 5 Pseudocode for semaphores

```
1: procedure DEC(Semaphore, Quantity)
2:     Semaphore ← Semaphore − Quantity
3: procedure INC(Semaphore, Quantity)
4:     Semaphore ← Semaphore + Quantity
5: function TRYDEC(Semaphore, Quantity)
6:     if Semaphore ≥ Quantity then
7:         DEC(Semaphore, Quantity)
8:         return true
9:     return false
```

a burst has high enough frequency to saturate available bandwidth, requests may still pile up, leading to response times longer than the round-trip latency.

4.9.1.3 Workloads

We use three workloads. The first consists of an endless burst of requests all issued at once, and compares changes in bandwidth costs of each scheme as a function of burst length. The second consists of two identical bursts with equally-spaced requests, separated by an idle period. It shows how response times change in each scheme before and after the idle period.

The third workload is based on the NetApp Dataset [18, 50], a corporate workload containing file system accesses from over 5000 corporate clients and 2500 engineering clients during 100 days. The file system uses 22TB of its 31TB of available space. More details about the workload are provided in the work by Leung et al. [50].

Our NetApp workload uses a 15 day period (Sept. 25 through Oct. 9) during which corporate and engineering clients were active. Requested chunk sizes range from a few bits to 64KB, with most at least 4KB [50]. Thus, we chose a 4KB block size. In total, 312GB of data were requested using $8.8 \cdot 10^7$ 4KB queries.

We configure the NetApp workload ORAM with a 32TB capacity, and allow 100GB of client space, for a usable storage increase of 328 times. For Burst ORAM and

ObliviStore, at least 33GB is consumed by the position map, and only 64GB is used for local block storage. The total block count is $N = 2^{33}$. Blocks are divided into $\lfloor 2^{17}/3 \rfloor$ partitions to maximize server space utilization, each with an upper-bound partition size of 2^{18} blocks.

4.9.2 Simulator

We evaluated Burst ORAM's bandwidth costs and response times using a detailed simulator written in Java. The simulator creates an asynchronous event for each block to be transferred. We calculate the transfer's expected end time from the network latency, the network bandwidth, and the number of pending transfers.

Our simulator also measures results for ObliviStore and the insecure baseline. In all schemes, block requests are time-stamped as soon as they arrive, and serviced as soon as possible. Requests pile up indefinitely if they arrive more frequently than the scheme can handle them.

Burst ORAM's behavior is driven by semaphores and appears data-independent to the server. Each request reads from a partition that appears to be chosen uniformly at random, so bandwidth costs and response times depend only on request arrival times, not on requested block IDs or contents. Thus, the simulator need only store counters representing the number of remaining blocks in each level of each partition, and can avoid storing block IDs and contents explicitly.

Since the simulator need not represent blocks individually, it does not measure the costs of encryption/decryption, permuting blocks, looking up block IDs, or performing disk reads for blocks. Thus, measured bandwidth costs and response times depend entirely on network latency, available bandwidth, request arrival times, and the scheme itself. While most of these ignored costs are negligible, server-side disk accesses do have

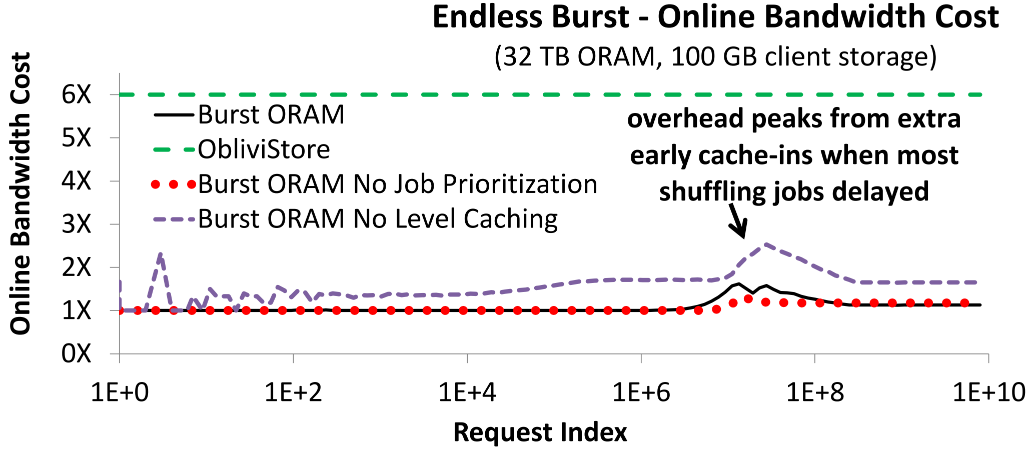

Figure 4.7: Online bandwidth costs as a burst lengthens. Burst ORAM maintains low online cost regardless of burst length, unlike ObliviStore.

the potential to become a substantial cost, especially on slow networks [76]. For now, we assume that blocks are distributed across enough fast disks that block transfers over the network are the bottleneck.

4.9.3 Endless Burst Experiments

For the endless burst experiments, we use a 32TB ORAM with $N = 2^{33}$ 4KB blocks and 100GB client space. We issue 2^{33} requests at once, then start satisfying requests in order using each scheme. We record the bandwidth costs of each request, averaged over requests with similar indexes and over three trials. Figures 4.7 and 4.8 show online and effective costs, respectively. The insecure baseline is not shown, since its online, effective, and overall bandwidth costs are all 1.

Figure 4.7 shows that Burst ORAM maintains 5X–6X lower online cost than ObliviStore for bursts of all lengths. When Burst ORAM starts to delay shuffling, it incurs more early shuffle reads, increasing online cost, but it still stays well under 2X on average.

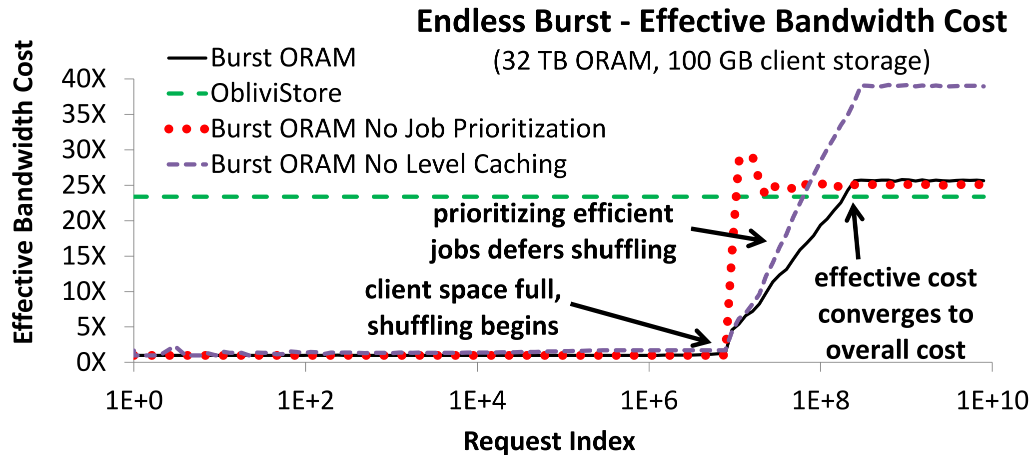

Figure 4.8: Effective bandwidth costs as a burst lengthens. Burst ORAM can handle most bursts with ∼1X effective cost. Each scheme's effective cost converges to its overall cost for long bursts.

Burst ORAM defers shuffling, so its effective cost stays close to its online cost until client space fills, while ObliviStore starts shuffling immediately, so its effective cost stays constant (Figure 4.8). Thus, response times for short bursts will be substantially lower in Burst ORAM than in ObliviStore.

Eventually, client space fills completely, and even Burst ORAM must shuffle continuously to keep up with incoming requests. This behavior is seen at the far right of Figure 4.8, where each scheme's effective cost converges to its overall cost. Burst ORAM's XOR technique results in slightly higher overall cost than ObliviStore's level compression, so Burst ORAM is slightly less efficient for very long bursts. Without local level caching, Burst ORAM spends much more time shuffling the smallest levels, yielding the poor performance of *Burst ORAM No Level Caching*.

If shuffle jobs are started in arbitrary order, as for *Burst ORAM No Prioritization*, the amount of shuffling per request quickly increases, pushing effective cost toward overall cost. However, by prioritizing efficient shuffle jobs as in *Burst ORAM* proper, more shuffling can be deferred, keeping effective costs lower for longer, and maintaining shorter response times.

4.9.4 Two-Burst Experiments

Our Two-Burst experiments show how each scheme responds to idle time between bursts. We show that Burst ORAM uses the idle time effectively, freeing up as much client space as possible. The longer the gap between bursts, the longer Burst ORAM maintains low effective costs during Burst 2.

Figure 4.9 shows response times during two closely-spaced bursts, each of $\sim 2^{27}$ requests spread evenly over 72 seconds. The ORAM holds $N = 2^{25}$ blocks, and the client has space for 2^{18} blocks. Since we must also store early shuffle reads and reserve space for the shuffle buffer, the client space is not quite enough to accommodate a single burst entirely. We simulate a 100Mbps network connection with 50ms latency.

All ORAMs start with low response times during Burst 1. ObliviStore response times quickly increase due to fixed shuffle work between successive requests. Burst ORAMs delay shuffle work, so response times stay low until client space fills. Without level caching, additional early shuffle reads cause early shuffling and thus pre-mature spikes in response times.

When Burst 1 ends, the ORAMs continue working, satisfying pending requests and catching up on shuffling during the idle period. Longer idle times allow more shuffling and lower response times at the start of Burst 2. None of the ORAMs have time to fully catch up, so response times increase sooner during Burst 2. ObliviStore cannot even satisfy all Burst 1 requests before Burst 2 starts, so response times start high on Burst 2. Burst ORAM does satisfy all Burst 1 requests, so it uses freed client space to efficiently handle early Burst 2 requests.

Clearly, Burst ORAM performs better with shuffle prioritization, as it allows more shuffling to be delayed to the idle period, satisfying more requests quickly in both

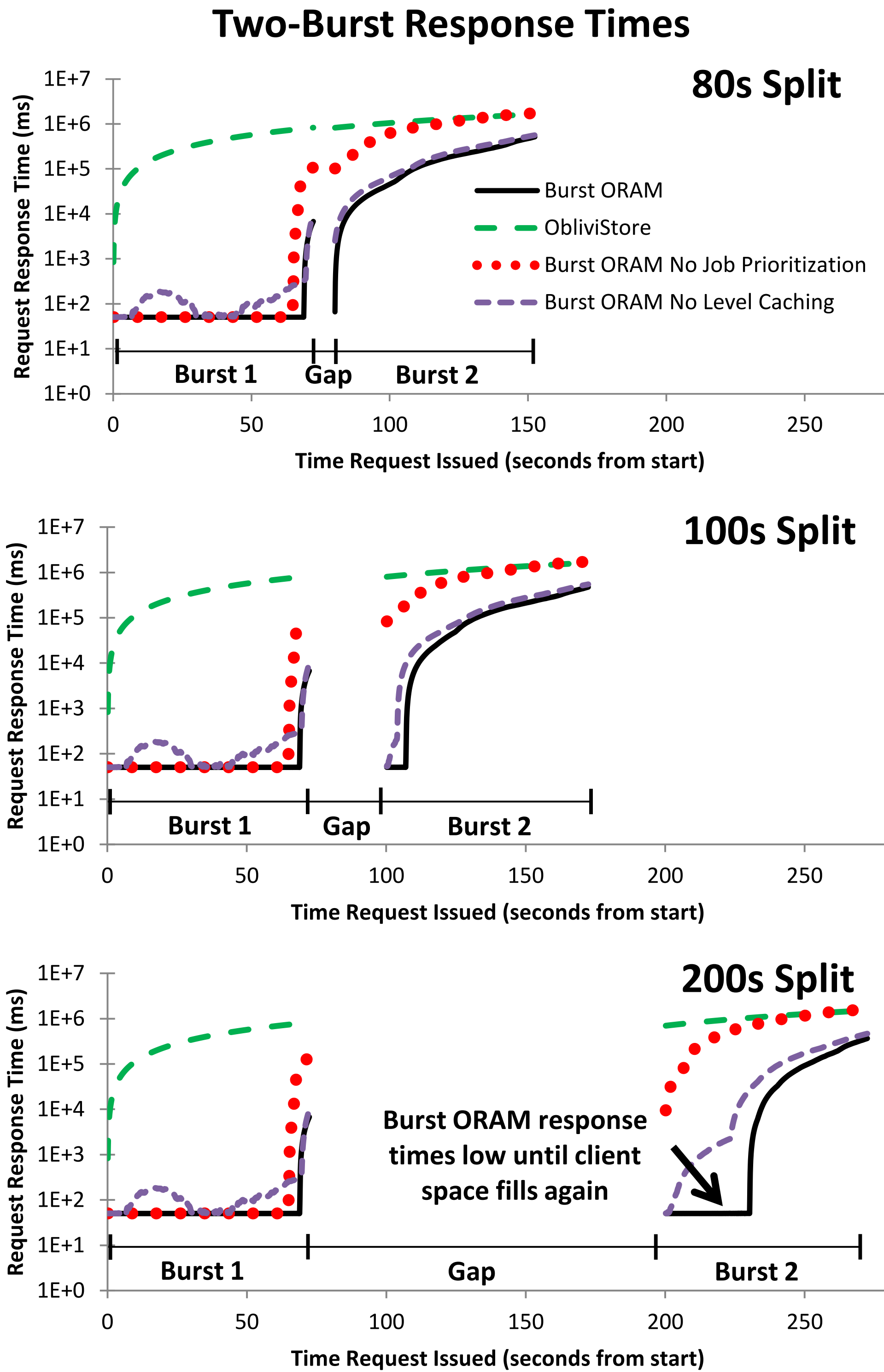

Figure 4.9: Response times during two same-size bursts of just over 2^{17} requests spread evenly over 72 seconds. Client has space for at most 2^{18} blocks. No level caching causes early spikes due to extra early shuffle reads.

bursts. Burst ORAM also does better with local level caching. Without level caching, we start with more available client space, but the extra server levels yield more early shuffle reads to store, filling client space sooner.

4.9.5 NetApp Workload Experiments

The NetApp experiments show how each scheme performs on a realistic, bursty workload. Burst ORAM exploits the bursty request patterns, minimizing online IO and delaying shuffle IO to achieve near-optimal response times far lower than ObliviStore's. Level caching keeps Burst ORAM's overall bandwidth costs low.

Figure 4.10 shows 99.9-percentile response times for several schemes running the 15-day NetApp workload for varying bandwidths. All experiments assume a 50ms network latency. For most bandwidths, Burst ORAM response times are orders of magnitude lower than those of ObliviStore and comparable to those of the insecure baseline. Shuffle prioritization and level caching noticeably reduce response times for bandwidths under 1Gbps.

Figure 4.11 compares p-percentile response times for p values of 90%, 99%, and 99.9%. It gives absolute p-percentile response times for the insecure baseline, and differences between the insecure baseline and Burst ORAM p-percentile response times (Burst ORAM *overhead*). When baseline response times are low, Burst ORAM response times are also low across multiple p.

The NetApp dataset descriptions [18, 50] do not specify the total available network bandwidth, but since it was likely sufficient to allow decent performance, we expect from Figure 4.10 that it was at least between 200Mbps and 400Mbps. Figure 4.12 compares the overall bandwidth costs incurred by each scheme running the NetApp

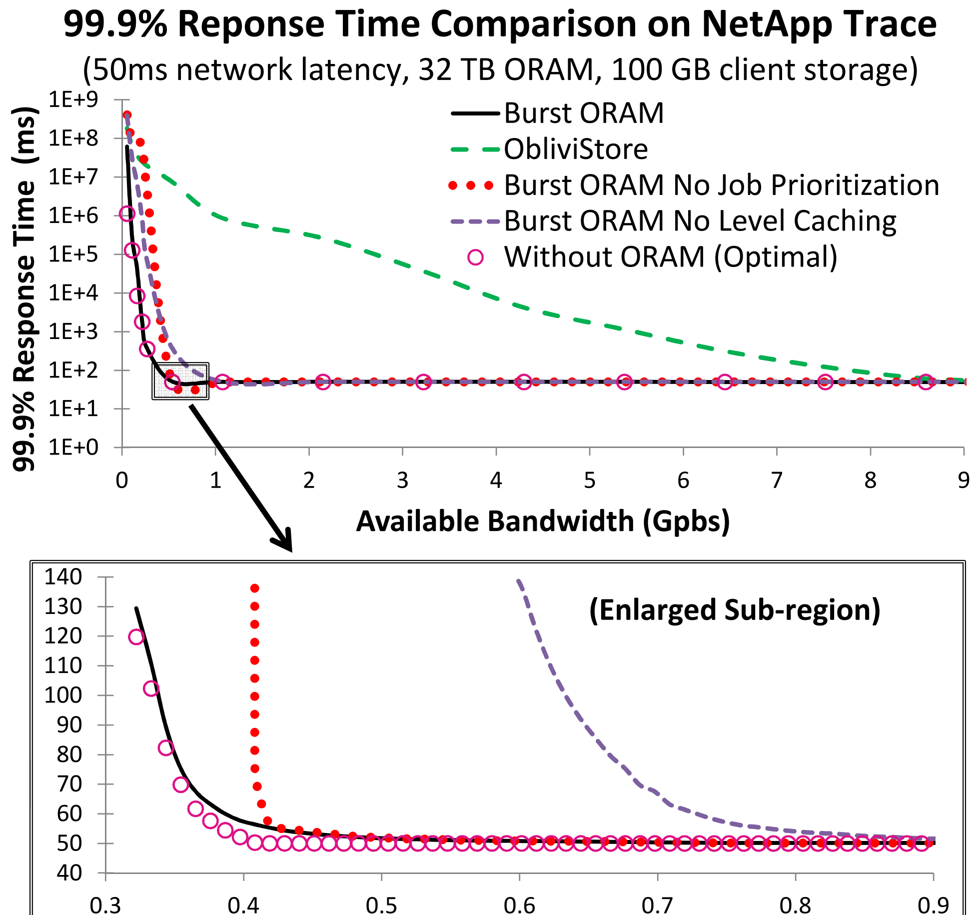

Figure 4.10: (Top) Burst ORAM achieves short response times in bandwidth-constrained settings. Since ObliviStore has high effective cost, it requires more available client-server bandwidth to achieve short response times. (Bottom) Burst ORAM response times are comparable to those of the insecure (without ORAM) scheme.

workload at 400Mbps. Costs for other bandwidths are similar. Burst ORAM clearly achieves an online cost several times lower than ObliviStore's.

Level caching reduces Burst ORAM's overall cost from 42X to 29X. Burst ORAM's higher cost is due to a combination of factors needed to achieve short response times. First, Burst ORAM uses the XOR technique, which is less efficient overall than ObliviStore's mutually exclusive level compression. Second, Burst ORAM handles smaller jobs first. Such jobs are more efficient in the short-term, but since they frequently write blocks to small levels, they create more future shuffle work. In ObliviS-

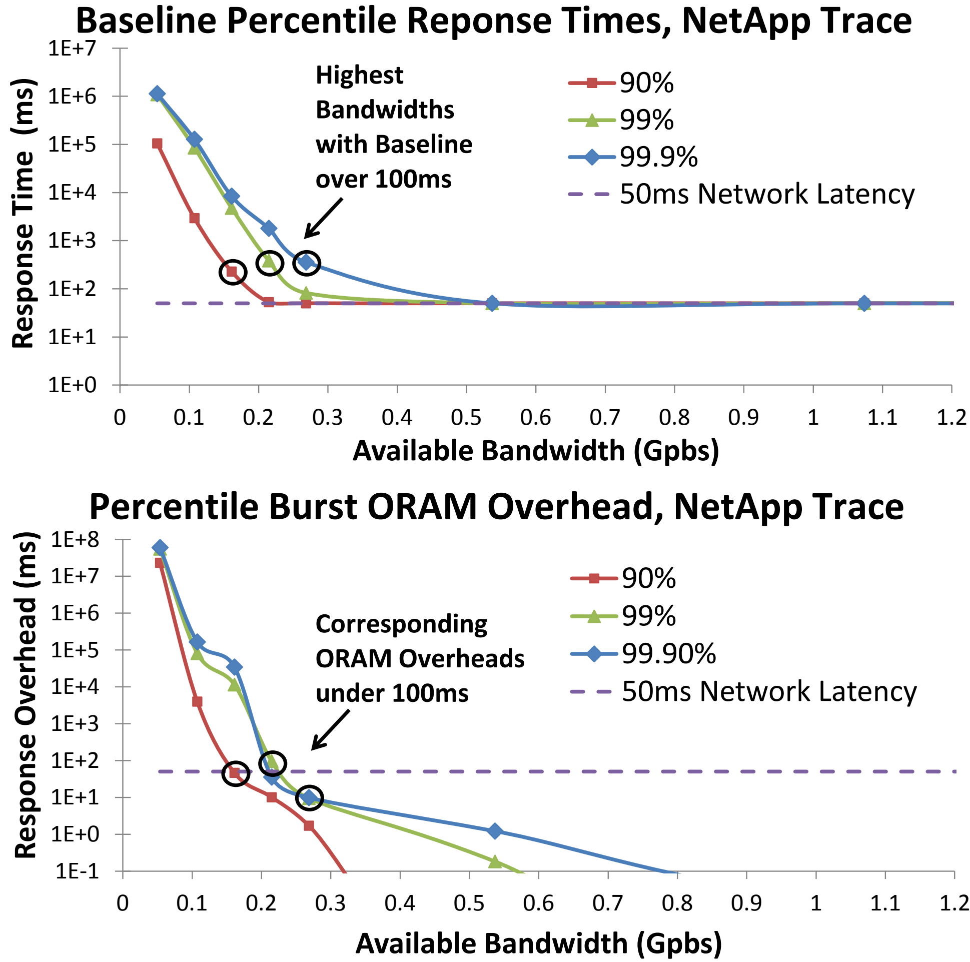

Figure 4.11: (Top) Insecure baseline (no ORAM) p-percentile response times for various p. (Bottom) Overhead (difference) between insecure baseline and Burst ORAM's p-percentile response times. Marked nodes show that when baseline p-percentile response times are < 100ms, Burst ORAM overhead is also < 100ms.

tore, such jobs are often delayed during a large job, so fewer levels are created, reducing overall cost.

4.10 Reducing Online Bandwidth Costs of SR-ORAM

SR-ORAM [86] is not part of our proposed Burst ORAM, but we briefly describe how it can benefit from our XOR block compression technique. Like ObliviStore, SR-ORAM requires only a single round-trip to satisfy a block request, and has online

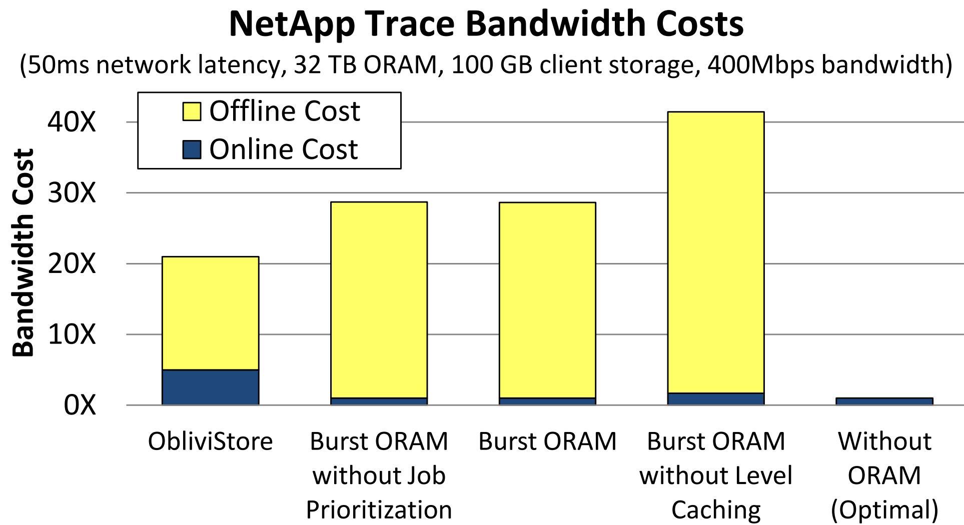

Figure 4.12: To achieve shorter response times, Burst ORAM incurs higher overall bandwidth cost than ObliviStore, most of which is consumed during idle periods. Level caching keeps bandwidth costs in check. Job prioritization does not affect overall cost, but does reduce effective costs and response times (Figures 4.8, 4.10).

IO bandwidth cost $O(\log N)$. SR-ORAM uses an encrypted Bloom filter to allow the server to obliviously check whether each level contains the requested block. Thus, the server retrieves the requested block from its level, and retrieves client-selected dummy blocks from all other levels. Since at most one block is real, the server can XOR all the blocks together and return a single compressed block, as in Burst ORAM.

One difference in SR-ORAM is that the client *does not* know a priori which level contains the requested block. Thus, the SR-ORAM must be modified to include the level-index of each retrieved block in its response. To allow the client to easily reconstruct dummy blocks, we must also change SR-ORAM to generate the contents of each dummy block using a pesudo-random function combining the block's index with a level-specific nonce. Since the client knows the indexes of the dummy blocks it requested from each level, it can infer the real block's level from the server's response. The client then reconstructs the contents of the dummy blocks returned from all other levels, and XORs them with the compressed block to obtain the requested block, as in Burst ORAM.

SR-ORAM is a synchronous protocol, so it has no notion equivalent to early shuffle reads. Thus block compression reduces SR-ORAM's online bandwidth cost from $O(\log N)$ to 1. The resulting reduction in overall cost is negligible, as SR-ORAM has an offline bandwidth cost of $O(\log^2 N \log\log N)$. SR-ORAM contains only one hierarchy of $O(\log N)$ levels, so block compression incurs only $O(\log N)$ extra storage cost for the level-specific nonces, fitting into SR-ORAM's logarithmic client storage.

4.11 Conclusion

We have presented Burst ORAM, an novel Oblivious RAM scheme based on ObliviStore and tuned for practical response times on bursty workloads. We presented a novel ORAM architecture for prioritizing online IO, and introduced the XOR technique for reducing online IO. We also introduced a novel scheduling mechanism for delaying shuffle IO, and described a level caching mechanism that uses extra client space to reduce overall IO. We simulated Burst ORAM on a real-world workload and showed that it incurs low online and effective bandwidth costs during bursts. Burst ORAM achieved near-optimal response times that were orders of magnitude lower than existing ORAM schemes.

Chapter 5

Combining ORAM with PIR to Minimize Bandwidth Costs

5.1 Introduction

Cloud computing allows customers to outsource the burden of data management and benefit from economy of scale, but privacy concerns limit its reach. Even if data blocks are encrypted by the client before being stored on the server, data access patterns may still leak valuable information [28, 44, 45]. *Private Information Retrieval* (PIR) [19] and *Oblivious RAM* (ORAM) [37] techniques both offer provable access pattern privacy for outsourced data, each with their own advantages and disadvantages. In this work, we combine the strengths of existing ORAM and PIR constructs to create a new ORAM with reduced bandwidth costs.

In PIR, data on the server may be encrypted or unencrypted, and the client issues an encrypted query for a particular bit or block of B bits. The server evaluates each query *homomorphically* (without decryption), returning the desired block without learning which block was requested. In order to achieve this degree of security, the server

must evaluate each query over all bits in the database, making PIR computationally prohibitive for most applications [74].

In ORAM, blocks of data are always encrypted by the client before being stored on the server. Informally, the ORAM defines a protocol that dictates how the client should fetch, permute (shuffle), re-encrypt, and store blocks from and to the server in order to prevent the server from learning any information about the pattern of plaintext block requests. The ORAM protocol guarantees that for any two same-length sequences of plaintext block requests, the resulting patterns of encrypted block accesses are computationally indistinguishable to all observers other than the client. ORAM requires negligible computation, but may incur substantial bandwidth or client/server storage overheads.

Definition 16. The *bandwidth cost* W of an ORAM or PIR technique is the number of blocks transferred for every block requested. Equivalently, it is the total number of bits transferred in order to retrieve B bits, where B is the size of a single block. If we upload and download 3 blocks to satisfy each request, we get $W = 6\text{X}$.

Bandwidth cost is particularly important when applying ORAM to mobile devices, where bandwidth costs are substantial. Existing ORAMs have bandwidth costs polylogarithmic in the total number of blocks N [35, 37, 39, 49, 76, 78, 87]. The ObliviStore ORAM [76] has the lowest bandwidth cost of any single-server ORAM proposed to date. ObliviStore's bandwidth cost is roughly $(\log_2 N)\text{X}$, with no hidden constants, though it requires extensive client storage. Other ORAMs require less storage [49, 58, 78] or emphasize reduced response times [11, 29, 86], but all incur higher total bandwidth costs.

5.1.1 Our Contributions

In this work, we show how to drastically reduce ObliviStore's bandwidth costs by combining it with PIR. The combination yields a new ORAM that we call OS+PIR that offers reduced bandwidth costs and permits tradeoffs between bandwidth cost, client/server storage, and computation. Combining ORAM and PIR was previously proposed in [58], but their scheme uses a different ORAM [73] and emphasizes constant client storage, while OS+PIR seeks to minimize bandwidth costs. OS+PIR treats its PIR component largely as a black box, so any efficient PIR technique may be used. For our experiments, we use the simple and efficient Trostle-Parrish PIR [81] that was also used in [57] and [58].

For each request in ObliviStore, the client retrieves and decrypts one block from each of $O(\log N)$ levels. At most one of the decrypted blocks is *real* and all the others are *dummies* which can be discarded. We use PIR to retrieve only the real block without revealing which block was accessed. Since we use PIR for only a small number ($O(\log N)$) of blocks, it is computationally feasible.

We divide OS+PIR's total bandwidth cost W into an ORAM component W_O and a PIR component W_P, where $W = W_O + W_P$. W_O depends on the ORAM's behavior, and W_P depends on the specific PIR and parameters such as block size. Using 2MiB blocks, the PIR in [81] yields W_P between 3X and 5X.

Applying PIR reduces ObliviStore's W_O by roughly 30%. We show how to amplify this reduction by altering the number and relative sizes of partition levels, balancing the reduced bandwidth cost with the resulting increases in PIR computation and server storage.

Increasing level sizes exacerbates a previously undiscovered issue of *unused dummy blocks* in ObliviStore. Unused dummies occur because ObliviStore makes more evictions than requests, as dictated by the *eviction rate*, which causes unnecessary block downloads. We mitigate this issue by securely altering OS+PIR's eviction pattern and creating fewer dummy blocks.

We also analyze two previously proposed techniques that use available client space to reduce bandwidth cost — reducing the eviction rate [76] and applying level caching [29] — and show how to strike a reasonable balance between the two. For systems with 2^{20} to 2^{30} blocks of 2MiB each, OS+PIR reduces the total bandwidth cost from ObliviStore's 18X–26X to only 11X–13X, 3X–4X of which is due to the PIR.

5.2 Related Work

Oblivious RAM was first proposed in [37] and required a bandwidth cost of $O(\log^3 N)$X blocks transferred per block requested, where N is the number of data blocks in the ORAM. Subsequent works have reduced the bandwidth cost to $O(\log^2 N/\log\log N)$X using constant client-side storage [49]. While constant client-side storage is desirable, it is not always necessary in practical settings. Recent works have reduced bandwidth costs to $O(\log N)$X by allowing additional client storage, specifically: $O(BN^v)$ client storage for constant $v > 0$ in [39], $O(B\log N)$ for large $B \in O(log^2 N)$ in [78], and $O(N\log N + B\sqrt{N})$ in [29, 76, 77].

Not all ORAMs that achieve $O(\log N)$X bandwidth cost are equally practical. For block sizes on the order of kilobytes or larger, the ObliviStore family of ORAMs [29, 76, 77] has the lowest practical bandwidth cost of any single-server ORAM proposed to date. ObliviStore's bandwidth cost is roughly $(\log_2 N)$X, with no hidden constants,

though it requires extensive client storage. In contrast, Path ORAM [78] requires closer to $(8 \log_2 N)$X, and more if client space is reduced using recursion. The scheme in [39] requires roughly $\log_2 N$ round-trips per request, but each trip may include several block transfers, making the total bandwidth cost at least 3 to 4 times that of ObliviStore. Since ObliviStore has the lowest bandwidth cost, we compare with it when evaluating OS+PIR.

Prior work [58] combined PIR with the tree-based ORAM of [73]. While their construction achieves the desirable property of constant client memory, it still requires $O(\log^2 N)$X bandwidth cost. In contrast, OS+PIR combines PIR with the partition-based ObliviStore ORAM [76] to achieve less than $(1/2)(\log_2 N)$X bandwidth cost in practice. Multi-cloud oblivious storage [75] achieves very low bandwidth cost (under 3X), but makes the strong assumption of multiple non-colluding servers not required by OS+PIR.

5.3 Preliminaries

Key notation used throughout the paper is shown in Table 5.1.

5.3.1 Private Information Retrieval (PIR)

Private Information Retrieval (PIR) was first proposed in [19]. PIR allows a client to retrieve specific bits from a server without revealing any information to the server, or any other observer, about which bits were accessed. In this work, we are primarily interested in PIR schemes in which each query returns a block of data, also known as Private Block Retrieval (PBR) schemes, but we will use the term *PIR* generally to include PBR.

Table 5.1: Notation

N	Total number of real (data) blocks in an ORAM
B	Size of each data block (in bits)
b	A specific plaintext data block
W	Total bandwidth cost
W_O	ORAM component of total bandwidth cost W
W_P	PIR component of total bandwidth cost W
ϵ	Eviction rate
p	A partition used in ObliviStore or OS+PIR
p_r	Request partition
p_a	Assignment partition
p_e	Eviction partition
k	Level size factor
K	Level configuration consisting of level size factors
r	Number of real blocks in a sub-level
L_M	Total number of main levels in a single partition
L_S	Total number of sub-levels in a single partition
D_i	Maximum number of dummies in main level i
s	Number of noise bits in PIR

PIR may be used to privately retrieve select plaintext data from a public server, such as to allow a patient to look up information about specific symptoms without disclosing the symptoms of interest. In this work, we use PIR to allow a client to retrieve specific blocks of his own encrypted, outsourced data without revealing which blocks are being accessed.

PIR comes in two flavors: *computational* and *information theoretic*. Computational PIR schemes are based on a hardness assumption, such that to retrieve information about a query, the adversary would need to solve a problem that is considered intractable. Information-theoretic PIR guarantees that query information remains secret, regardless of the adversary's computational resources, but generally requires an assumption of non-colluding servers [19]. The non-colluding server assumption is impractical when servers are untrusted, so we use computational PIR here.

Each PIR query is encrypted by the client, sent to the server, and evaluated *homomorphically* (under encryption) by the server. The server returns the requested

bits but learns neither the plaintext contents of the query nor which bits were returned. PIR must perform a computational operation over every bit in the PIR database in order to achieve its strong privacy guarantees, so the entire database must be loaded from disk for each query. Authors in [74] argue that computational PIR is more expensive than the trivial PIR of downloading the entire database for each query. While subsequent PIR developments challenge this conclusion [63], PIR is still considered prohibitively expensive for many applications.

Instead of using PIR for the entire database, we use it to reduce the bandwidth cost required to retrieve one of a small number of blocks that would otherwise all be returned. In this scenario, the PIR database is small, and the block size relatively large, so the substantial bandwidth cost reduction can outweigh the small increase in computation, given an appropriate PIR scheme. We discuss our use of PIR in more detail in Section 5.4.

5.3.2 Oblivious RAM (ORAM)

Oblivious RAM (ORAM) was first proposed in [37]. Like PIR, ORAM may be used to retrieve data from a server without revealing which data were accessed. In ORAM, unlike PIR, every block must be encrypted by the client before being stored to the server. Most ORAMs allow block contents to be updated, while standard PIR techniques are read-only.

Instead of evaluating queries homomorphically, ORAM defines a protocol for transferring and manipulating encrypted blocks such that the underlying plaintext block accessed by one query cannot be linked to any other query's block. ORAMs download encrypted blocks from the server to the trusted local client space, decrypt them, return the desired information, then re-encrypt the blocks using a semantically secure encryp-

tion scheme to break correlations with previous encrypted *contents*. The ORAM then randomly permutes the blocks to break correlations with the previous block *position*, a process referred to as oblivious shuffling. Some ORAMs also create *dummy blocks*, which are indistinguishable from real blocks but contain random or generated data, and download extra blocks, in order to break correlations.

ORAM security is defined as follows. For any two sequences of block requests of the same length, the resulting patterns of encrypted block accesses must be computationally indistinguishable to all observers other than the client [77]. Equivalently, the output of a simulator that has no access to any of the secret information (block contents and requested block addresses) should be able to produce a sequence of encrypted block transfers that is indistinguishable from that of the actual ORAM [49].

5.3.3 ObliviStore

OS+PIR builds on the ObliviStore ORAM [76]. A full description of ObliviStore and its underlying ORAM [77] is too extensive to include here, but we review the aspects most relevant to OS+PIR. See [76, 77] for a more comprehensive discussion. For N blocks of B bits each, ObliviStore uses a relatively large amount of client storage, $O(B\sqrt{N} + N \log N)$ bits with small constants, in order to achieve a low bandwidth cost of $(\log_2 N)$X.

5.3.3.1 Partition and Level Structure

In ObliviStore, blocks stored on the server are arranged logically into $O(\sqrt{N})$ *partitions*, each of which contains $O(\sqrt{N})$ blocks. Each partition is a simplified hierarchical ORAM, similar to those of [37], with roughly $\log_2 \sqrt{N}$ levels. The lowest level in each partition (level 0) holds 1 real and 1 dummy block. Successive levels double in

size, so level i has real-block capacity $r_i = 2^i$ and starts with 2^i dummy blocks. At any given time, each level may be occupied or empty, and only half the levels are occupied on average.

5.3.3.2 Requests and Evictions

Each block request involves three steps:

1) Partition Request When the client issues a request for block b, we direct the request to the partition p_r containing b. The choice of p_r is deterministic, but appears random to an observer since b was previously assigned to a randomly chosen partition. We then download exactly one block from every non-empty level in p. We fetch b from whichever level contains it, and fetch a dummy block from every other level. Since levels were previously randomized, each fetched block appears randomly chosen from its level. After downloading the blocks, we discard all dummies, return b to the client, update b if necessary, and *assign* it to a new partition.

2) Assignment: After downloading and updating b, we encrypt it and assign it to a partition p_a chosen uniformly at random. Each p_a maintains a local, hidden *eviction queue* of blocks assigned to p_a but not yet evicted to the server. We assign b to p_a by adding it to the end of p_a's eviction queue, but do not immediately evict it.

3) Eviction: After assigning b to p_a, we independently choose at least one partition p_e and evict the next block from its eviction queue, or a dummy block if the queue is empty. We perform ϵ evictions after each request, where ϵ is the real-valued *eviction rate*.[1] If ϵ exceeds 1.0, we add the fractional component $\epsilon - 1.0$ into a global accumulator. When the accumulator reaches 1.0, we make another eviction and

[1]ObliviStore uses slightly different notation, where v is the *background eviction rate*, equal to $1 - \epsilon$.

decrement the accumulator. Eviction partitions (p_e) may be chosen deterministically or randomly, as long as the choice is independent of p_a [77].

Choosing p_a randomly guarantees that future requests for b will appear to access a random partition. Choosing p_e independently of p_a prevents an observer from learning p_e and thus from tracking b between partitions. Thus, the independence of p_a and p_e is critical to ObliviStore's security.

Since p_a and p_e are chosen independently, blocks blocks may accumulate in eviction queues. ObliviStore calls the space needed to store these blocks the *eviction cache*. Revealing the size of the eviction cache (number of assigned but not yet evicted blocks) may leak information about prior choices of p_a, so a fixed amount of space sufficient for the eviction cache is reserved up-front. Statistically, the higher ϵ, the less space is required for the eviction cache.

5.3.3.3 Shuffling

Let level i in partition p initially contain r_i real and r_i dummy blocks. Once r_i evictions have been made to p since i was created, i is scheduled to be *re-shuffled*. Informally, shuffling i consists of:

1. Downloading all blocks left in level i
2. Removing any remaining dummies
3. Inserting any evicted blocks
4. Generating any additional dummies
5. Randomly permuting and re-encrypting all blocks
6. Uploading all blocks to a new level with $2r_i$ real and $2r_i$ dummy blocks

When two levels i and $i-1$ are both ready to be re-shuffled, the shuffle *cascades* upward. In fact, since level sizes increase by factors of 2, every time a level i is ready to

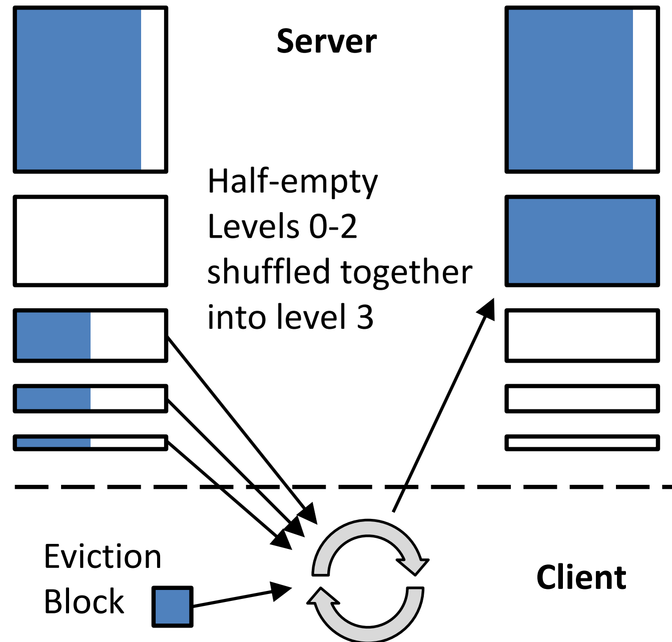

Figure 5.1: In ObliviStore, when shuffles cascade upward, all levels ready to be re-shuffled are downloaded at once, shuffled together with eviction blocks, and uploaded to a higher level.

be re-shuffled, all lower levels are necessarily also ready to be re-shuffled. When shuffling all levels up to i, we download all remaining blocks from levels 0–i and upload blocks to level $i+1$ with $2r$ real blocks, leaving levels 0–i empty to accommodate future evictions (see Figure 5.1).

5.3.3.4 Early Shuffle Reads

If level i has more than half of its original $2r$ blocks remaining, then there is at least one dummy block in i to return. If instead i has at most r blocks remaining, it is possible that all the blocks are real, so to maintain obliviousness we must always *treat* such blocks as real. We call such blocks *early shuffle reads*, since they would eventually be downloaded as part of the upcoming shuffle. ObliviStore refers to such blocks as either *early cache-ins* or *real cache-ins* depending on the context.

Early shuffle reads are relatively rare, and are caused by delayed shuffles in ObliviStore. Since ObliviStore performs a constant amount of shuffle work per request, some of the levels to shuffle may have not yet been downloaded when a subsequent request arrives, causing the early shuffle read. Other than early shuffle reads, all but one of the downloaded blocks are guaranteed dummies. Early shuffle reads are downloaded separately and stored.

5.3.3.5 Level Compression

ObliviStore's level compression algorithm, described in [77], allows the client to send k real and k dummy blocks to the server using only kB bits. The technique uses a pre-shared Vandermonde matrix $M_{2k \times k}$ to encode the k real blocks and their positions into a "compressed" stream of kB bits. The server decompresses the stream to get $2k$ blocks, including k encrypted real blocks and k dummies containing random data derived from the decompression. The dummy and real blocks are indistinguishable and intermixed.

We can alter the number of dummy blocks generated by changing the number of rows in the matrix M from $2k$ to the desired total number of blocks. This flexibility becomes important in Sections 5.5 and 5.6 where we use it to reduce OS+PIR's total bandwidth cost.

5.3.3.6 Bandwidth Costs

In ObliviStore, the client has enough space to store all $\sqrt{N}$ blocks from any given partition, and to store the location of all N blocks. Since every partition fits entirely in client memory, re-shuffling requires that each block be downloaded and uploaded only once. (If client space were smaller, we would need a more expensive oblivious

shuffling algorithm.) Thus, to shuffle r real blocks, we need only transfer $3r$ blocks: r real downloads, r dummy downloads, and r uploads for level compression.

The total bandwidth cost W of ObliviStore is determined by the number of times each block must be re-shuffled per request. Each partition has roughly $(\log_2 \sqrt{N})/2 = (\log_2 N)/4$ occupied levels at any given time. Each of the $\sqrt{N}$ real blocks is re-shuffled once per occupied level per $\sqrt{N}$ evictions, for a total of $(3/4)\log_2 N$ block transfers per eviction. Since ϵ gives the number of evictions per request, we get an expected cost for ObliviStore of roughly:

$$W \approx \frac{3\epsilon}{4} \log_2 N \tag{5.1}$$

ObliviStore reports an actual cost $W \approx \log_2 N$ for $\epsilon = 1.3$ [76]. The slight discrepancy with Equation 5.1 can be accounted for by the problem of *unused dummies* that we address in Section 5.6.

5.4 Integrating PIR

In ObliviStore, each request fetches one block from each level in a partition, at most one of which is real. At most one of the blocks fetched is real and all others are dummies, except in the case of *early shuffle reads*, which are always returned individually. Since the client discards returned dummies, they are only transferred to mask the real block's identity. We would like to retrieve the real block and hide its identity without paying to transfer dummies.

The recently-proposed Burst ORAM [29] combines these fetched blocks using XOR and returns a single combined block. The client then reconstructs the dummy blocks from a pseudo-random function and subtracts them out of the combined block to recover the real block. Unfortunately, the XOR optimization is incompatible with

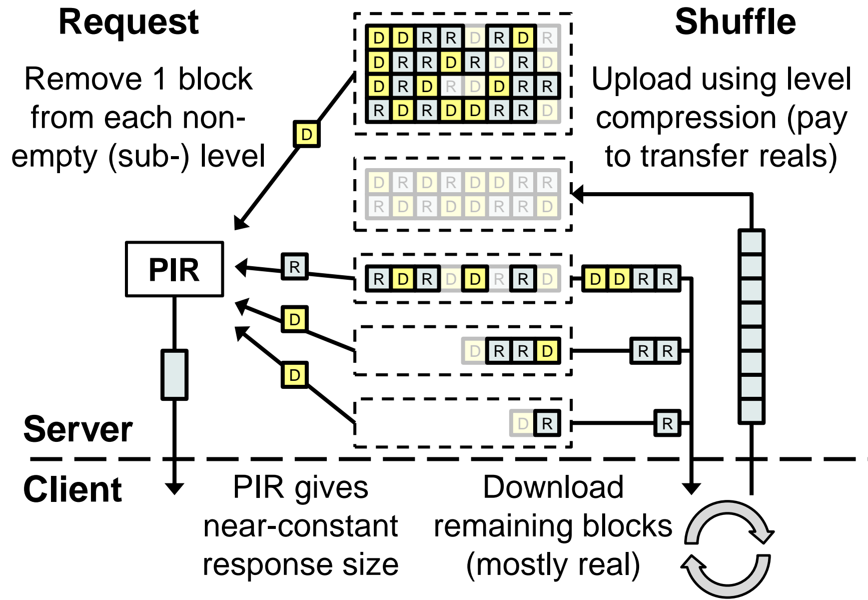

Figure 5.2: In an OS+PIR request, since only one removed block is real, PIR reduces the response cost for each request to a constant or near-constant size. During shuffling, only blocks remaining in each level (most of which are real) need to be downloaded. Because of level compression, we need only transfer data of size equivalent to the real blocks being uploaded. In all, dummy transfers in OS+PIR are "free" — we effectively pay only to transfer real blocks.

ObliviStore's level compression, since level compression derives dummies from real blocks during decompression, but the XOR requires that dummies be generated from a pseudo-random function. Thus, Burst ORAM avoids paying to *download* dummies, but any savings in total bandwidth cost is negated by the lack of level compression, which avoids paying to *upload* dummies.

In OS+PIR, we instead use PIR to retrieve the real block. Since PIR makes no stipulations on dummy block content, it can be used with level compression. PIR itself incurs a bandwidth cost W_P determined by the block size B, number of blocks (levels) over which it queries, and the specific PIR scheme used. Other than W_P, transfers of dummy blocks are essentially free (see Figure 5.2), reducing W_O by roughly 30% up-front and enabling additional modifications that further reduce cost (see Section 5.5).

5.4.1 Choosing a PIR Technique

In OS+PIR, we execute the PIR protocol once per block request. Each PIR instance operates over a *PIR database* of $L_S \in O(\log N)$ blocks, one from each level, and returns a single block. Since we query over only $O(\log N)$ blocks, PIR is far more computationally feasible than when used to query over all N blocks.

PIR schemes often measure bandwidth in terms of the cost of returning the full PIR database. To remain consistent with Definition 16, we instead measure PIR bandwidth cost as W_P: the total amount of data transferred during each PIR operation, divided by B. We want a PIR scheme with low W_P, as near the optimal 1X as possible. Since ObliviStore's bandwidth cost is already low, even a W_P as large as 10X could easily negate any advantages of using PIR. Ideally, W_P should be constant: independent of L_S and B. In practice, we can use any PIR that offers low, nearly-constant W_P for small L_S and large B.

One candidate is the Gentry-Ramzan PIR [36], which offers a constant W_P with a theoretic minimum of 2X, closer to 4X in practice due to the Coppersmith attack [24], but incurs substantial computation costs. Another option is the more computationally-efficient Trostle-Parrish PIR [81], which offers low, but not constant, W_P when $B \gg L_S$. OS+PIR treats its PIR as a black box, so any PIR that meets our criteria can easily be swapped in. For now, we will use the Trostle-Parrish PIR [81], also used in [57, 58], due to its simplicity and low communication and computation complexity.

5.4.2 Trostle-Parrish PIR

We now briefly describe the trapdoor group Trostle-Parrish PIR that we use in our experiments. A more detailed explanation can be found in [81]. The PIR database

of n bits is divided into a square matrix of $\sqrt{n}$ by $\sqrt{n}$ bits. The client generates an initial hidden large prime modulus q and a secret value $\beta \in \mathbb{Z}_q$.

The client generates $\sqrt{n}$ query elements $c_1, \ldots, c_{\sqrt{n}}$ for each query. Each c_i encodes a low-order bit that is 1 if row i should be retrieved from the matrix, and 0 if not. The client may request up to ρ rows at once by including ρ such bits in each query element, where at most one bit per element is set, and each bit is set by at most one element. The higher-order bits are filled in with s bits of random noise. The client then encrypts each query element as $\beta \cdot c_i \mod q$, and sends the vector of encrypted elements to the server.

The server applies a linear transform on the query vector, taking its dot product with each column in the matrix, and returns the resulting vector to the client. The client then multiplies the ith response by $\beta^{-1} \mod q$ to recover the ith plaintext sum, from which he can extract the ρ bits from the ith column.

When we use Trostle-Parrish to recover one of L_S blocks of B bits, we optimize bandwidth cost by choosing a modulus of $s + \sqrt{B/L_S}$ bits. The PIR then incurs a bandwidth cost of approximately $2+2s\sqrt{L_S/B}$, and requires an average of $n/2$ additions over $\mathbb{Z}_q$. We assume $s = 2^9$ bits of noise, guaranteeing a prime modulus at least as large as those used in [81]. Thus, for a 2MiB block size B, we can support $L_S = 2^6$ blocks at $W_P = 4X$.

5.5 Altering Level Size Factors

Combining PIR with ObliviStore's level compression technique effectively gives OS+PIR free dummy block downloads and uploads, aside from PIR computation and bandwidth W_P costs. As noted in Section 5.3.3.5, we can modify level compression to

produce additional dummy blocks at no extra cost. Similarly, given near-constant W_P, we can query over any number of additional levels at no extra bandwidth cost. We now show how to use these properties to reduce OS+PIR's bandwidth cost by increasing level sizes.

In ObliviStore, successive levels increase in size by a factor of 2, yielding $\log_2 \sqrt{N}$ levels per partition. In OS+PIR, we allow successive levels to increase by any integer factor. Let k_i be the *level size factor* of main level i, which defines the sub-level real-block capacity ratio r_i/r_{i-1}. We must allow up to $k_i - 1$ instances or *sub-levels* of main level i, which when shuffled together with all lower levels become a single sub-level of main level $i+1$. The real-block capacity of a sub-level in level i is given by $r_i = \prod_{j=0}^{i-1} k_j$.

To simplify the presentation of ideas throughout this Section, we assume that OS+PIR uses $\epsilon = 1.0$ (exactly one eviction per request). We address larger eviction rates in Sections 5.6 and 5.7.

5.5.1 Effects of Increasing Level Size Factors

We simplify our discussion of the high-level effects of increasing level size factors (k_i values), by assuming $k_i = k$ for all i, where $k = 2$ in ObliviStore. Figure 5.3 shows two level configurations ($k = 2$ and $k = 4$) for a partition with 15 real blocks. We discuss non-uniform level size factors and special handling of the top level later in this Section. Increasing k has the following effects:

It increases the total number of sub-levels L_S. For a partition of $\sqrt{N}$ blocks we need $L_M = \log_k \sqrt{N}$ main levels. With $k-1$ sub-levels per main level, the total number of sub-levels is given by $L_S = (k-1)\log_k \sqrt{N}$, which increases almost

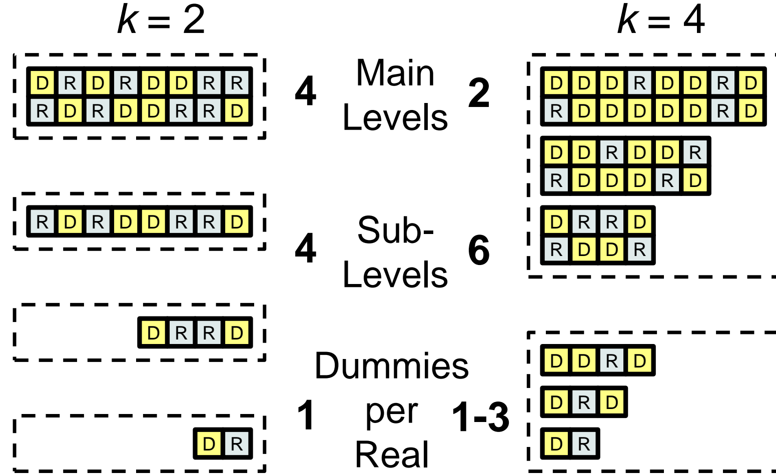

Figure 5.3: Level configurations with size factors $k = 2$ and $k = 4$, both with a 15 real-block capacity. When shuffling, all sub-levels in a main level are combined to form one new sub-level in the next largest main level. The $k = 4$ configuration has fewer main levels, and thus lower shuffling costs. The $k = 2$ has fewer dummies and sub-levels, and thus lower disk and PIR costs.

linearly with k. To maintain obliviousness, we must fetch one block from every sub-level during each request. Thus, without PIR, the bandwidth cost W would increase almost linearly with k, which is why ObliviStore uses only $k = 2$. With near-constant W_P, the effect on W is negligible. Even with PIR, the number of blocks that must be read from disk to satisfy a request increases with k.

It increases level and sub-level lifetimes. The ith main level holds $k - 1$ sub-levels containing k^i real blocks each, and is re-shuffled after every k^{i+1} requests. Once the ith level is shuffled into a higher level, it stays empty for k^i requests before its first new sub-level is created. The first sub-level must live through $(k-1)k^i$ requests and still have k^i blocks left over to avoid early cache-ins. Thus we need $(k-1)k^i$ dummies in addition to the k^i reals, for a total of k^{i+1} blocks. The second sub-level lives for k^i fewer requests, so it needs only $(k-2)k^i$ dummies, and so on. The increased number of dummies also increases server storage to a factor of roughly k, which we address in Section 5.5.2.

It decreases the ORAM component bandwidth cost W_O. Between every shuffle (k^{i+1} requests), the ith level receives $k^{i+1}(k-1)/2$ dummy blocks and $(k-1)k^i$ real blocks in all. The number of shuffle *downloads* per request is only $((k-1)/k)\log_k \sqrt{N}$, since we only pay to download the remaining $(k-1)k^i$ blocks from each sub-level. With level compression, the number of shuffle *uploads* per request is the same, giving:

$$W_O \approx 2\frac{k-1}{k}(\log_k \sqrt{N})\text{X} \tag{5.2}$$

Thus, for OS+PIR with $N = 2^{32}$ real blocks and $\epsilon = 1.0$, increasing k from 2 to either 4, 16, or 64 should reduce W_O from approximately 16X to either 12X, 7.5X, or 5.25X, respectively.

5.5.2 Non-Uniform Level Size Factors

A major limitation of using a large fixed level size factor k is that it increases server storage cost. The ith level stores a total of k^{i+1} blocks, only k^i of which are real, for a server storage factor of roughly k. However, the bulk of the extra dummy blocks, at least $k-1$ of each k, are stored in the largest level. With varying k_i, level $i \geq 1$ stores D_i dummy blocks in the worst case, given by:

$$D_i \approx \left(\frac{k_i}{2}\right)^2 \cdot \prod_{j=0}^{i-1} k_j \tag{5.3}$$

By allowing level size factors to differ, specifically using smaller k_i for larger levels and larger k_i for smaller levels, we can mitigate the storage cost increase but still keep the number of levels small.

For example, consider the two configurations $K_1 : (k_0 = 2^5, k_1 = 2^5, k_2 = 2^5)$ and $K_2 : (k_0 = 2^7, k_1 = 2^5, k_2 = 2^3)$. For both, the real-block capacity is roughly $\prod_{j=0}^{2} k_j = 2^{15}$. However, for K_1 we get $D_0 + D_1 + D_2 = 2^8 + 2^{13} + 2^{18} \approx 2^{18}$ dummy

blocks total, while K_2 gives $D_0 + D_1 + D_2 = 2^{12} + 2^{15} + 2^{16} \approx 2^{16}$ dummies total, reducing the server storage factor from 8X to 2X.

Asymptotically, by increasing level factors doubly-exponentially, we can reduce the number of main levels to $L_M \approx \log_2 \log_2 \sqrt{N}$ for a partition with $\sqrt{N}$ real blocks, while incurring only a L_M server storage cost factor. Consider the configuration:

$$K : (k_0 = 2^{(\log_2 \sqrt{N})/2}, k_1 = 2^{(\log_2 \sqrt{N})/4},$$
$$k_3 = 2^{(\log_2 \sqrt{N})/8}, \ \ldots, \ k_{L_M - 3} = 2^8$$
$$k_{L_M - 2} = 2^4, k_{L_M - 1} = 2^2, k_{L_M} = 2^1).$$

Since the level factor exponents in K grow exponentially from 1 to $(\log_2 \sqrt{N})/2$, we have $L_M \approx \log_2 \log_2 \sqrt{N}$, and thus $W_O \approx (2 \log_2 \log_2 \sqrt{N})$X. Applying Equation 5.3, we see that $D_i \approx \sqrt{N}$ for all i. Since the real-block capacity is $\sqrt{N}$, the server storage overhead is just $(\log_2 \log_2 \sqrt{N})$X.

5.5.2.1 Practical Limits on Level Size Factor Growth

Unfortunately, increasing or skewing the level size factors also increases the maximum total number of sub-levels L_S given by:

$$L_S \leq \sum_i (k_i - 1). \quad (5.4)$$

For every request, we must fetch L_S blocks from disk and perform PIR over L_S blocks. Thus, disk read and PIR computation costs are at least proportional to the largest k_i, limiting growth in practice.

In the simple example above, K_2 suffers from $L_S \approx 165$, while K_1 has only $L_S \approx 93$. For such small skews, the difference is not dramatic, but in a comparable configuration with all $k_i = 2$, we have only $L_S \approx 15$. In the double-exponential growth example, $k_0 = 2^{(\log_2 \sqrt{N}/2)} = N^{1/4}$. For large databases with $N \geq 2^{32}$, we end up with

$L_S \geq k_0 \geq 2^8$. Such large L_S values could easily make the disk and PIR costs outweigh any benefit of reduced bandwidth cost in practice.

A more practical approach is to follow the double-exponential growth only up to a maximum level factor determined by a fixed acceptable value of L_S that can be accommodated by the disk array and PIR computation hardware. In Section 5.8.4 we empirically evaluate the impact of different level size configurations on L_S.

5.6 Eliminating Unused Dummies

We have so far assumed that exactly one block is evicted to partition p for each request from p. However, in both ObliviStore and OS+PIR, the ratio of evictions to requests for a given partition can vary over time. We will now describe the problem of *unused dummies*, which occurs when p sees more evictions than requests.

5.6.1 Causes of Eviction/Request Difference

The variance in number of evictions and requests to a partition comes from the eviction rate ϵ and the independent selection of the request p_r and eviction p_e partitions. Recall from Section 5.3.3.2 that ϵ determines the number of evictions per request and is set greater than 1.0 to reduce the eviction cache size. Even with $\epsilon = 1.0$, since p_r and p_e are chosen independently at random, it is possible that in the short term a given partition will receive more evictions than requests, or vice-versa.

5.6.2 Source of Unused Dummies

In ObliviStore, and in OS+PIR as described so far, each level is created with one dummy block for each eviction that will occur before the level is re-shuffled. However, we actually remove one block *per request*, not *per eviction*. Thus, if we have fewer

than one request per eviction, some extra dummy blocks will be left in the level when it is ready to be shuffled. These extra dummy blocks serve no purpose, so we call them *unused dummies*. The shuffler must download these unused dummies in order to re-shuffle the level, incurring unnecessary bandwidth cost.

We can estimate the added bandwidth cost due to unused dummies as follows. We temporarily ignore partition selection independence, as it primarily impacts small levels. Let ϵ be the eviction rate, k_i the size factor of level i, and r_i the number of real blocks in each sub-level of i. For the jth sub-level, we create $r_i(k_i - 1 - j)$ dummies, but expect only $r_i(k_i - 1 - j)/\epsilon$ requests before it must be re-shuffled. We therefore have $r_i(k_i - 1 - j)(1 - 1/\epsilon)$ unused dummies. Since the sub-level already incurs a minimum of $2r_i$ block transfers, the fractional increase in bandwidth cost is:

$$\Delta_{i,j} \approx \frac{1}{2}(k_i - 1 - j)\left(1 - \frac{1}{\epsilon}\right). \tag{5.5}$$

For ObliviStore, with $\epsilon = 1.3$ and $k_i = 2$, $\Delta_{i,0}$ is only 11.5%. For OS+PIR with $k_i = 64$, the increase due to the top sub-level ($j = 0$) is far larger: $\Delta_{i,0} = 738\%$. $\Delta_{i,0}$ overestimates the total impact, since sub-levels $j > 0$ have smaller $\Delta_{i,j}$, but the problem is clearly exacerbated by OS+PIR's larger k_i.

5.6.3 Proposed Mitigations

In OS+PIR, we eliminate most of the unused dummies by simply generating fewer dummies for each level. Recall from Section 5.3.3.5 that ObliviStore's level compression lets us generate any number of dummies. We expect only $r_i(k_i - 1 - j)/\epsilon$ requests before i is re-shuffled, so we compensate for ϵ by generating only $r_i(k_i-1-j)/\epsilon$ dummy blocks when creating sub-level j.

Since p_r and p_e are chosen independently, we may still have more or fewer requests than expected. If the number of requests is large, we get early shuffle reads, and if small, we get unused dummies, both of which increase bandwidth cost.

In OS+PIR, we resolve this issue by choosing the eviction partition to be the request partition ($p_r \equiv p_e$). More precisely, after we request a block b from p_r, we assign it to a randomly chosen partition p_a as before, but perform ϵ evictions on the request partition p_r. We add any fractional component of ϵ to an accumulator specific to p_r. Whenever p_r's accumulator exceeds 1.0, we evict another block to p_r and decrement its accumulator.

Our strategy for choosing p_e guarantees that the number of requests differs by at most one from the expected number, eliminating the remaining unused dummies. The change also slightly increases eviction cache space, since assigned blocks wait longer to be evicted initially.

To clarify, we are not altering the method for selecting the assignment partition p_a, nor are we suggesting that b be re-assigned to p_r. It is critical for security that p_a be chosen randomly, secretly, and independently of p_r and p_e, so that the server cannot track blocks between partitions. However, the choice of p_e need not be random as long as it is independent of secret client information like p_a [77]. Since the choice of p_r is already public, choosing $p_r \equiv p_e$ does not affect security.

5.7 Bandwidth Cost Enhancements

We now present three enhancements to OS+PIR that use available client space to further reduce the ORAM component bandwidth cost W_O. We then show how to balance the client space allocated to each enhancement in order to minimize W_O.

5.7.1 Enhancement Techniques

Reduce Eviction Rate: Our first enhancement reduces the eviction rate ϵ, as suggested in [77]. Increasing ϵ increases the number of shuffles per request, and thus W_O. Reducing ϵ grows the client's eviction cache. The minimum ϵ is 1.0, but ObliviStore uses $\epsilon = 1.3$ to keep the eviction cache small.

Cache Small Levels: Our second enhancement caches the smallest levels of each partition on the client, as proposed by Burst ORAM [29]. Since each main level contributes equally to W_O (roughly 2X each), storing small levels locally avoids their bandwidth cost entirely while using only a small amount of space. Unlike lowering ϵ, level caching consumes publicly visible space on the client and does not grow the eviction cache. As in [29], to prevent deadlock, we cache only as many levels as will fit in client space even if all were full simultaneously across all partitions. As a rule of thumb, we set k_0 so that all sub-levels in the lowest main level are cached.

Shuffle Largest Jobs First: The third enhancement adapts Burst ORAM's technique of shuffling *efficient* partitions first to instead shuffle partitions with the largest minimum write level first. ObliviStore chooses partitions to shuffle using a FIFO approach. Our approach lets us delay shuffling when it would write small levels to the server, in hopes that after some time, the partition will receive more evictions and bypass filling in the small levels. The intent is similar to level caching in that if we avoid writing the lowest levels to the server, we never incur their associated bandwidth costs.

5.7.2 Effects and Tradeoffs

Caching levels and reducing ϵ both require extra client space. Savings due to level caching increase only logarithmically with additional space, and savings from

reducing ϵ also suffer from diminishing returns. To minimize bandwidth costs we must carefully allocate space to each enhancement. The best allocation depends on available client space and the level configuration K. We measure bandwidth costs for different values of ϵ in Section 5.8.5.

5.8 Experiments

We ran simulations to compare OS+PIR with the original ObliviStore [76] and a modified ObliviStore equipped with the enhancements discussed in Sections 5.6 and 5.7: eliminating unused dummies, reducing eviction rate, etc. In all our experiments, we assume a 2MiB block size ($B = 2^{24}$ bits). The large block size is needed primarily to keep PIR bandwidth cost W_P low when using the Trostle-Parrish PIR. Changing the block size has little effect on the ORAM component bandwidth cost W_O. For the unmodified ObliviStore, we use $\epsilon = 1.3$ as in [76], and use $\epsilon = 1.1$ for OS+PIR. We discuss our choice of eviction rate in Section 5.8.5.

5.8.1 Simulators and Implementations

5.8.1.1 ORAM Simulator

We evaluated bandwidth costs for OS+PIR and ObliviStore using a simulator written in Java. Since ORAM behavior is oblivious, performance is independent of the specific sequence of blocks requested. Thus, for efficiency, the simulator uses counters to represent the number of remaining blocks in each level of each partition, and avoids storing block IDs and contents explicitly. Since we are primarily interested in bandwidth and computation costs, the simulator does not explicitly measure the costs of permuting

blocks, looking up IDs, or performing disk reads. Block encryption costs are also not logged, as they are dominated by PIR costs.

Each experiment includes a run-up and evaluation phase of $4N$ requests each. We count the total number of blocks transferred (uploads plus downloads) during the evaluation phase, and divide by $4N$ to get W_O. For OS+PIR we also record the number of sub-levels L_S accessed during each request. L_S varies across requests depending on the number of empty sub-levels in each partition.

5.8.1.2 PIR Implementation

We implemented the Trostle-Parrish PIR [81], as described in Section 5.4, using Java. Our implementation caches partial sums to avoid redundant computations, but is otherwise unoptimized. For all our experiments, we used $s = 2^9$ bits of extra noise in the PIR as noted in Section 5.4.2. Server computation is trivially parallelizable.

We measure wall-clock times running PIR on a single thread of a third generation Amazon Web Services (AWS) Elastic Compute Cloud [3] instance (half of a c3.large instance), equivalent to 3.5 AWS ECUs. As of May 2014, the cost for running the full c3.large instance was \$0.105/hour, giving an approximate PIR cost of $\$1.46 \times 10^{-5}$/second on a single thread. Figure 5.4 gives PIR time and bandwidth costs for L_S up to 2^8, with a maximum server time under 160s, equivalent to \$0.0023 per 2MiB block request.

5.8.2 Evaluating OS+PIR for Mobile Devices

We start by evaluating OS+PIR on parameters suitable to current mobile devices. We consider an OS+PIR with $N = 2^{22}$ of our 2MiB blocks, giving a server storage capacity of 1TiB. We allocate 64GiB total client storage, such that the ORAM

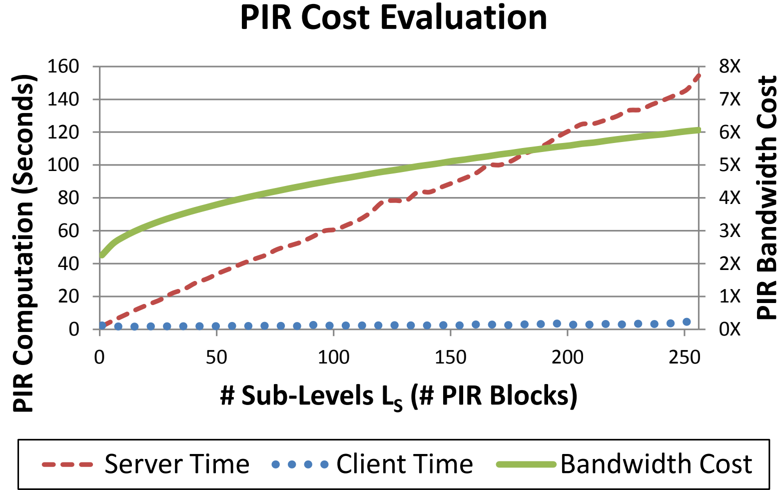

Figure 5.4: Timing and bandwidth costs of Trostle-Parrish PIR implementation for varying numbers of blocks in PIR database, using $s = 512$ bits noise and 2MiB block size.

increases effective storage capacity by a factor of 16. We can increase this factor by increasing N (see Section 5.8.3).

Table 5.2 shows our results. Comparing the modified and unmodified versions of ObliviStore, we see that our enhancements offer a slight improvement on their own, reducing total bandwidth cost W from 21.4X to 18.2X. Adding PIR offers another slight improvement, bringing W down to 16.2X. Finally, increasing the level factors reduces W to as little as 11.2X, but increases server storage and PIR computation costs.

We also give per-request costs in US cents (¢) for each scheme for two benchmark bandwidth costs. On one extreme we have the AWS [3] bandwidth cost of \$0.12 per GB for the first 10TiB per month, for a cost of 0.025¢ per 2MiB block. On the other we have cellular data, which may cost as much as \$10 per GB, for a cost of 2.10¢ per 2MiB block. At \$0.012/GB, OS+PIR roughly breaks even, with its added PIR computation cost canceling out the reduced bandwidth costs. At \$10/GB, OS+PIR is a clear win, as the bandwidth cost savings far outweigh the PIR cost, cutting total cost

Table 5.2: Comparison of different bandwidth-efficient protocols given parameters tuned for current mobile devices with at least 64GiB storage. Common parameters: $N = 2^{22}$ data blocks, 2MiB block size, 64GiB total client storage.

				PIR Computation per Request			Bandwidth Cost per Request (X)			Total Cost per Request	
Protocol	ϵ	Level Factors	Svr. Stor. Fact.	Server Time (s)	Server Cost (¢)	Client Time (s)	W_P	W_O	W	In ¢ at 12¢/ GB	In ¢ at $10/ GB
ObliviStore	1.3	$(2,\dots,2)$	3.2	—	—	—	—	21.4	21.4	0.054	44.9
ObliviStore Modified	1.1	$(2,\dots,2)$	2.9	—	—	—	—	18.2	18.2	0.046	38.2
OS+PIR	1.1	$(2,\dots,2)$	2.9	6.04	0.009	0.78	2.6	13.6	16.2	0.050	34.0
OS+PIR	1.1	$(4,32,16,2,2)$	4.5	20.55	0.030	0.92	3.2	8.5	11.7	0.059	24.6
OS+PIR	1.1	$(4,64,16,2)$	6.4	31.62	0.046	0.99	3.6	7.6	11.2	0.074	23.5

down to nearly half that of ObliviStore. OS+PIR is clearly most cost-effective when bandwidth costs dominate, as would be the case for mobile devices.

5.8.3 Varying Block Count N

Table 5.3 shows client/server space consumption and total bandwidth cost W as N increases. We show results for the unmodified ObliviStore, OS+PIR with $K = (2,\dots,2)$, and a custom K chosen to minimize L_M while keeping server storage costs low. Required client storage scales with $\sqrt{N}$, and for each N it is nearly the same for all three schemes. As N grows, the capacity/client space ratio grows as well, from 68 for $N = 2^{20}$ to 2189 for $N = 2^{30}$. For a large 2PiB database, the client needs 1TiB storage, and OS+PIR reduces W from 26.4X to 13.0X.

The bandwidth cost savings of OS+PIR depend on reducing the number of main levels $L_M = |K|$. However, to keep server storage costs low, we must use small level factors for the highest levels, which limits our savings when N is small. However, as N grows, our advantage increases, as L_M grows more slowly in OS+PIR than in ObliviStore due to OS+PIR's larger level factors.

Table 5.3: Effect of increasing N on client/server storage and bandwidth cost, with $\epsilon = 1.1$ for OS+PIR, $\epsilon = 1.3$ for ObliviStore.

		ObliviStore			OS+PIR $K=(2,\ldots,2)$			OS+PIR Custom K			
N	Capac. (TiB)	Client (GiB)	Server (TiB)	W (X)	Client (GiB)	Server (TiB)	W (X)	K	Client (GiB)	Server (TiB)	W (X)
2^{20}	2	30	6.2	18.2	30	5.9	14.3	$(4, 64, 8, 2)$	31	11.2	10.9
2^{22}	8	60	24.8	19.8	61	23.4	15.3	$(4, 16, 16, 4, 2)$	62	31.6	11.4
2^{24}	32	119	99.2	21.4	122	93.3	16.5	$(4, 32, 16, 4, 2)$	123	127.6	12.0
2^{26}	128	239	396.2	23.1	244	373.0	17.6	$(4, 64, 16, 4, 2)$	245	522.4	12.5
2^{28}	512	478	1584.6	24.7	488	1491.4	18.7	$(4, 64, 32, 4, 2)$	491	2452.4	12.7
2^{30}	2048	958	6335.8	26.4	980	5962.9	19.8	$(4, 64, 32, 8, 2)$	990	9756.0	13.0

Table 5.4: Effects of changing the level configuration K for $N = 2^{28}$ block count, 512TiB capacity, 512GiB total client storage. Product of all level size factors for each K is 2^{16}.

K	Server Storage Factor	W_O	Avg L_S	Max L_S	W_P
$(2,\ldots,2)$	2.9	16.1X	6.7	14	2.7X
$(4,\ldots,4,2,2)$	3.1	13.1X	9.5	20	2.8X
$(4,8,8,8,8,2,2)$	3.5	11.2X	14.5	30	3.0X
$(4,16,8,8,4,2,2)$	3.2	11.3X	16.5	34	3.0X
$(4,16,16,16,2,2)$	4.3	9.9X	23.0	47	3.2X
$(4,32,16,16,2)$	5.8	9.1X	30.5	62	3.4X
$(4,128,16,4,2)$	4.3	9.1X	72.5	146	4.1X
$(4,128,64,2)$	15.7	7.6X	95.0	191	4.4X

5.8.4 Varying Level Size Configuration K

Table 5.4 shows the results of running $OS + PIR$ for various K given a fixed $N = 2^{28}$ and 512GiB of client storage. We use the same maximum real-block partition capacity for all K, so the product of all level factors in each K is fixed (2^{16}). As predicted in Section 5.5.2.1, using larger level factors, and thus fewer main levels $L_M = |K|$, greatly reduces ORAM bandwidth cost W_O, but also increases L_S, which in turn increases PIR and disk access costs. Server storage costs increase substantially when level factors for the highest levels are increased, even when bandwidth costs remain the same (see $K = (4, 32, 16, 16, 2)$ and $K = (4, 128, 16, 4, 2)$).

5.8.5 Varying Eviction Rate

Table 5.5 shows the effects on W_O of using different ϵ for $N = 2^{28}$ with $K = (2,\ldots,2)$ and 1TiB of client storage. As predicted in Section 5.7.2, smaller ϵ reduce bandwidth cost but grow the eviction cache, leaving less space available for level caching.

Table 5.5: Effects of changing the eviction rate ϵ for OS+PIR with $N = 2^{28}$, 512TiB capacity, given a fixed total client storage of roughly 1TiB. Client space insufficient to support $\epsilon = 1.0$.

Eviction Rate ϵ	Eviction Cache Space Required	W_O
1.0	10.39TiB	—
1.05	0.43TiB	13.5X
1.1	0.23TiB	13.2X
1.2	0.12TiB	14.3X
1.3	0.08TiB	15.3X
1.5	0.06TiB	17.4X
2.0	0.03TiB	22.8X

Total client storage is large enough to accommodate $\epsilon = 1.05$, but not $\epsilon = 1.0$. W_O generally decreases with decreasing ϵ, but the large eviction cache needed for $\epsilon = 1.05$ leaves so little space for level caching that W_O is actually smaller for $\epsilon = 1.1$ than $\epsilon = 1.05$. The optimal choice of ϵ depends on N, available client storage, and level configuration K. We chose $\epsilon = 1.1$ as it offers a good compromise when client space is generous, but not extreme.

5.8.6 Evaluating Individual Enhancements

In Table 5.6 we compare the effectiveness of each of our enhancements to OS+PIR from Sections 5.6 and 5.7. We use $N = 2^{28}$ blocks with a level configuration of $K = (4, 64, 32, 4, 2)$. The first column gives the bandwidth cost of starting with no enhancements, then adding each one individually. The second columns gives costs starting with all enhancements, then removing each one individually. Since the interplay between enhancements in each pair makes them hard to evaluate in isolation, we combined *Generate Fewer Dummies* with *Evict to Request Partition*, and *Cache Smallest Levels* with *Shuffle Largest Jobs First*.

Table 5.6: Effects of selectively applying/excluding enhancements to/from OS+PIR. Common parameters: $N = 2^{28}$, $\epsilon = 1.3$ base and 1.1 reduced, $K = (4, 64, 32, 4, 2)$, 479–491GiB total client storage.

Enhancement	W_O with Only Specified Enhancement	W_O with All Except Specified Enhancement
Generate Fewer Dummies and Evict to Request Partition	11.6X	14.2X
$\epsilon = 1.1$	14.7X	10.1X
Cache Smallest Levels and Shuffle Largest Jobs First	23.9X	9.7X
None/All	25.5X	9.0X

Table 5.6 shows that W_O varies from 25.5X with no enhancements down to 9.0X with all enhancements. Since the level size factors in K are large, the *Generate Fewer Dummies* and *Evict to Request Partition* enhancements make a large impact. The remaining enhancements make smaller but still significant contributions.

5.9 Conclusion

We have presented OS+PIR, a new ORAM that combines the bandwidth-efficient ObliviStore ORAM [76] with PIR techniques in order to minimize total bandwidth costs. We have shown how to re-engineer ObliviStore to accommodate levels of varying relative sizes in order to fully exploit PIR, exposing a tradeoff between bandwidth cost, server computation, and server storage. OS+PIR also includes several enhancements that further reduce costs, including mechanisms for eliminating the unnecessary dummy blocks introduced in ObliviStore.

In all, OS+PIR achieves bandwidth costs at least 2 times lower than those of ObliviStore, making it especially advantageous for mobile devices, where bandwidth costs dominate. In other settings, OS+PIR's cost-effectiveness is limited by its PIR

computation cost, but its current PIR can be easily replaced by any future PIR schemes that prove more efficient.

Chapter 6

Tunably-Oblivious Memory: Generalizing ORAM to Enable Privacy-Efficiency Tradeoffs

6.1 Introduction

It has become common for resource-constrained clients to outsource data storage and management to *cloud* servers lying beyond their administrative control. Such outsourcing, however, raises data privacy concerns. Unfortunately, merely encrypting data does not ensure privacy. Much information is leaked even by access patterns on encrypted data [28, 44].

Oblivious RAM (ORAM) protocols [37] can guarantee full access pattern privacy in an outsourced block store. ORAM protocols use dummy block reads and periodic oblivious data block re-shufflings to guarantee that any two access patterns of the same length are computationally indistinguishable to any outside observer, including

the server itself. ORAM costs are generally dominated by *bandwidth cost*, which we measure as the number of actual block transfers needed to satisfy a single block *access* (read or write).

ORAM bandwidth costs range from $O(\sqrt{N \log N})$ [11] to $O(\log N)$ [39, 76, 77], where N is the ORAM block capacity. Recently, there has been a push to make ORAM practically, as well as asymptotically, efficient [54, 76, 78, 87]. The most bandwidth-efficient ORAM construction known to date [76, 77] still incurs a bandwidth cost of roughly $\log_2 N$. Other protocols [35, 49, 58, 78, 87] use less client space than [76, 77], but incur higher bandwidth costs.

This $\log_2 N$ cost is particularly disappointing for multi-block read-only queries, where we might expect better performance. To achieve full access pattern indistinguishability, ORAMs must ensure that all queries generate public access patterns of roughly the same length, regardless of access locality or ORAM state. As a result, all queries must incur the same, worst-case cost.

We avoid this limitation by building special-purpose ORAM-like protocols that leak a strictly bounded amount of access pattern information in order to obtain a bandwidth cost under $\log_2 N$ for large queries. Existing schemes that partially protect access patterns (e.g. [30, 61]) start with unprotected protocols and add obfuscation mechanisms to quantifiably limit the adversary's ability to make certain inferences. However, they do not consider all possible inferences, and thus cannot assess total information leakage. In contrast, we start with a fully protected protocol (ORAM) and carefully relax its privacy requirements in order to tightly bound the total access pattern information leaked.

6.1.1 Our Contributions

We propose Tunably-Oblivious Memory (TOM), a new model that relaxes and generalizes the traditional ORAM model, allowing controlled tradeoffs between efficiency and information leakage. TOM permits variable-length public access patterns, allowing properties such as locality to be exploited to improve efficiency. Queries are distinguishable by access pattern length, so for each query λ-TOM generates an access pattern with one of λ pre-determined lengths, limiting information leaked per query to $\log_2 \lambda$ bits. λ-TOM protocols with large λ are more flexible and efficient, but leak more information. Protocols with small λ are more rigid, but offer better privacy. 1-TOM leaks no information, and has security equivalent to a traditional ORAM.

TOM can directly improve efficiency for queries showing locality by simply enhancing ORAM with a local block cache. However, we address the more challenging problem of building a TOM that efficiently handles workloads that are *not* cache-friendly. To this end, we propose a novel, special-purpose TOM called *Staggered-Bin TOM* (SBT). We prove that SBT achieves bandwidth cost $O(\log N/\log\log N)$ for large queries with blocks chosen uniformly at random, but has worst-case cost $O(\sqrt{N})$.

We also propose three read-only SBT variants, culminating in the Multi-SBT, which combines the SBT with a traditional ORAM, storing three copies of each block. The Multi-SBT achieves bandwidth cost $O(1)$ for large uniform random block queries, and $O(\log N)$ in the worst case, while leaking only $O(\log\log\log N)$ bits per query. Thus, Multi-SBT can satisfy any ℓ-block uniform random block query using only $O(\ell + \log N)$ block transfers.

We developed a simulator to evaluate SBT and its variants, and compare practical costs of the Multi-SBT with the ORAM in [77]. We show that Multi-SBT maintains

Table 6.1: Comparison of [77] with results based on our proposed Multi-SBT using the ORAM component from [77], with the parameterizations and costs given below and in [77], with 64 KB block size. Multi-SBT average cost is for uniform random queries of length $\ell = 4\sqrt{N}, \lambda = 8$. Max. cost is three times ceiling of ORAM cost.

		ORAM [77]			Multi-SBT using ORAM from [77]				
N	ORAM Capacity	Client Store	Server Store	Cost	Client Store	Server Store	Avg. Cost	Cost Upper-Bound	Leaked Bits/ Access
2^{20}	64GB	204MB	205GB	22.5X	604MB	333GB	5.4X	69X	$3{\cdot}2^{-12}$
2^{22}	256GB	415MB	819GB	24.1X	1.2GB	1.3TB	6.0X	75X	$3{\cdot}2^{-13}$
2^{24}	1TB	858MB	3.2TB	25.9X	2.6GB	5.2TB	6.3X	78X	$3{\cdot}2^{-14}$
2^{28}	16TB	4.2GB	51TB	29.5X	13.6GB	83TB	5.8X	90X	$3{\cdot}2^{-16}$

a practical bandwidth cost of roughly 6X for queries of $4\sqrt{N}$ blocks, while [77] has substantially larger costs ranging from 22X to 29X for similar parameterizations (see Table 6.1).

Section 6.2 covers related work in protecting access pattern privacy. Section 6.3 presents the TOM model and its security definition. We describe the SBT in Section 6.4 and its variants in Section 6.5, with detailed performance analyses in Section 6.6. Section 6.7 gives experimental results from our simulator comparing SBT and its variants.

6.2 Related Work

6.2.1 ORAM Protocols

We focus on showing that the Multi-SBT outperforms the practical ORAM in [76, 77] because it remains the most bandwidth-efficient single-server ORAM, and incurs a similar client space cost ($O(N)$ with low constant). ORAMs that emphasize reduced client space incur even higher bandwidth cost. Assuming 64KB blocks, practical ORAM [76, 77] requires $\log_2 N$ bandwidth cost. Path ORAM [78] requires closer to $8 \log_2 N$, and more if client space is reduced using recursion. The ORAMs in [49] and [35] both

have asymptotic bandwidth cost $O(\log^2 N / \log \log N)$, and are outperformed in practice by Path ORAM [78]. Multi-cloud oblivious storage [75] achieves very low bandwidth cost (under 3X), but makes the strong assumption of multiple non-colluding servers.

6.2.2 Partial Access Pattern Protection

Several efficient protocols have been proposed that *partially* protect access patterns. One example is the *Shuffle Index* [30], which uses an unchained B+ tree to store encrypted blocks. *Cover searches* provide access pattern privacy by making dummy block requests to obscure the true request. The authors quantify the adversary's ability to recognize that two given accesses correspond to the same block, but ignore other information leaks. For example, the protocol may run indefinitely without retrieving certain blocks. Since the adversary knows that such blocks are rarely requested, he can use their eventual request pattern to make additional inferences. In contrast, TOM's bounds on total information leakage hold for all inferences. Shuffle Index bandwidth cost is 16X, but drops to 4X with enough client space to store pointers to each block.

The protocol in [61] reads two blocks for every one requested, does no oblivious shuffling, and achieves a bandwidth cost as low as 4X even with limited client space. Like the Shuffle Index it bounds the adversary's ability to correlate two accesses, but it leaks even more unquantified information via access patterns of rarely requested blocks than does the Shuffle Index.

Like TOM, the private computation protocol of [89] uses an ORAM and allows a bounded amount of access pattern information to leak in order to improve efficiency. However, the notions are otherwise fundamentally different. The protocol in [89] accesses main memory from trusted hardware via a black-box ORAM, using the additional space to enable more elaborate computations. Applications vary in the number of required

ORAM fetches per computation. Leakage comes through each application's one-time maximum fetch rate choice. In contrast, TOM allows fetch counts to vary dynamically, letting the ORAM adjust fetch counts to match workloads, leaking information per query instead of per application setup. Thus, TOMs see efficiency improvements when the average number of fetches is small, even if the worst-case number is large.

6.3 Tunably-Oblivious Memory

6.3.1 ORAM Review

Oblivious RAM (ORAM) techniques [37] provide a mechanism for outsourcing encrypted data while ensuring that all possible access patterns are computationally indistinguishable to all observers other than the client, including the server itself. In an ORAM protocol, the client arranges his data in N fixed-size blocks of B bits each. Each block has a unique address $a \in \{0, 1, \ldots, N-1\}$. Each of the N blocks is encrypted using a semantically secure encryption scheme and then stored on the server. Every time a block a is written to the server, it is re-encrypted using a different nonce, and assigned a new server-side ID, preventing it from being directly linked to previous encrypted versions of a.

The goal of ORAM is to define an efficient protocol that re-shuffles and re-encrypts blocks to ensure that no information is leaked about the address or contents of each block, how frequently a given block is accessed, and whether the access is a read or write. The protocol may incorporate *dummy* blocks, which contain no data but are indistinguishable from encrypted data blocks.

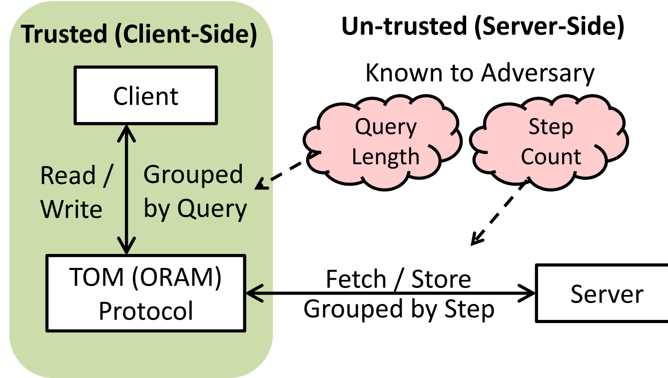

Figure 6.1: Client issues secret accesses (reads/writes) to the ORAM/TOM protocol, which translates them to a sequence of public accesses (stores/fetches) to the untrusted server.

A client interacts with the ORAM protocol as with a trusted block store (Figure 6.1), issuing a *secret access pattern* $\vec{S} = (s_1, \ldots, s_{|\vec{S}|})$ of block requests. Each secret access s is a triple $(type, a, data)$, where $type$ is the access type ($read$ or $write$), a is the local address of the block to access, and $data$ is the plaintext data to be written to block a, if any.

The ORAM translates $\vec{S}$ into a *public access pattern* $P(\vec{S}) = (p_1, \ldots, p_{|P(\vec{S})|})$ that is generally much longer than $\vec{S}$. Each public access p is also a triple $(type, id, edata)$, where $type$ denotes the access type ($store$ or $fetch$), id denotes the server-side ID of the accessed block, and $edata$ denotes the encrypted block data to be stored, if any. A fetch optionally removes the block from the server.

The term *access pattern* has been used in the literature ambiguously to refer to either $\vec{S}$ or $P(\vec{S})$. We disambiguate by calling $\vec{S}$ the *secret* access pattern and $P(\vec{S})$ the *public* access pattern. We now give the standard ORAM security definition of [77] in terms of our notation:

Definition 17. A protocol satisfies ORAM security if for every pair of secret access patterns $\vec{S_1}$ and $\vec{S_2}$ of the same length ($|\vec{S_1}| = |\vec{S_2}|$), $P(\vec{S_1})$ and $P(\vec{S_2})$ are computationally indistinguishable (to every observer other than the client).

If the ORAM block size B is reasonably large ($B \gg \log_2 N$), the communication cost is dominated by block transfers. The ORAM makes $|P(\vec{S})|$ block transfers to satisfy $\vec{S}$, while an unprotected protocol needs only $|\vec{S}|$ transfers. Thus the *bandwidth cost* of using ORAM to obscure an access pattern is given by $\frac{|P(\vec{S})|}{|\vec{S}|}$. The more efficient an ORAM, the lower its bandwidth cost.

6.3.2 TOM Model: Trading Off Obliviousness for Efficiency

We introduce the term *step* to refer to a discrete unit of work performed by an ORAM or TOM. Informally, each step retrieves a single encrypted *target* block from the server. Each step may also fetch and store other blocks in order to obscure the target block's identity or prepare for future requests (e.g. shuffling).

In a traditional ORAM, each secret access yields exactly one such step, and the target block is simply the block associated with the secret access. Each ORAM step is powerful in that it can obliviously retrieve any given target block from the server, but this power also makes each step expensive. The step count must match the number of secret accesses in order to satisfy ORAM's perfect privacy guarantee, so such powerful, expensive steps are mandatory.

In contrast, the TOM generalization allows the step count to vary, creating the possibility for more efficient but less powerful steps, and thus for more efficient protocols. For example, the Staggered Bin TOM (Section 6.4) partitions the blocks on the server into k bins (Figure 6.2). Each step may only retrieve a target block from a single, pre-determined bin. Each such step is thus less powerful than an ORAM step,

but it is also more efficient. In the worst case, k steps are needed to satisfy a single secret access, but by carefully scheduling secret accesses from the same multi-block query, we can obtain lower overall bandwidth cost than a comparable ORAM. Allowing the step count to vary inevitably leaks some access pattern information. We show how to tightly bound such information in Sections 6.3.4 and 6.3.5.

We now define the TOM model more precisely. For each secret access pattern $\vec{S}$, a TOM generates a public access pattern $P(\vec{S})$ divided into a sequence $\sigma(\vec{S})$ of discrete *steps*. We use $|\sigma(\vec{S})|$ to denote the *step count* of $P(\vec{S})$.

Definition 18. Each *step* is a series of stores and fetches used by the TOM protocol to retrieve a single *target* block from a subset of the blocks on the server. A step is complete when the TOM is ready to retrieve another target block.

Traditional ORAMs are special cases of TOM in which each secret access generates exactly one step ($|\sigma(\vec{S})| = |\vec{S}|$). Thus ORAM does not distinguish between *secret access* and *step*, necessitating our new terminology for TOM. In ORAM, the block subset accessible during a step includes all blocks on the server, while in Staggered Bin TOM (Section 6.4) it only includes blocks from one bin.

We say $\vec{S}$ is *satisfied* once all the steps in $\sigma(\vec{S})$ are complete. As in ORAM, if the TOM is *stateful*, some blocks updated by $\vec{S}$ may not be stored to the server immediately. Instead, even after the step completes, they are held locally as *dirty* blocks until they are written back to the server during a subsequent step.

Definition 19. A *query* is a secret access pattern $\vec{S}$ composed of a batch of secret accesses that may be satisfied in any order.

A TOM receives multi-block *queries* from the client. Queries are handled sequentially relative to each other, but accesses within a query may be processed in any order. For security, *query* and *secret access pattern* are interchangeable.

TOM decouples *steps* from *secret accesses*, allowing query length $|\vec{S}|$ to differ from step count $|\sigma(\vec{S})|$. This approach offers better efficiency than ORAM for two reasons. First, TOM need not generate steps for accesses to cached blocks. In ORAM, a repeat access to a recently cached block must still incur the overhead of a step, else the reduced step count would reveal the repeated access. Second, TOM need not require that each step be capable of accessing any block. By reducing the power of each step, TOM makes steps more efficient, potentially reducing a query's total bandwidth cost, even though the step count may increase. The SBT and its variants (Sections 6.4 and 6.5) exploit this second advantage.

6.3.3 TOM Security Definition

As in ORAM, we assume that query length $|\vec{S}|$ is public. We also make the worst-case assumption that the adversary can observe precisely when each query starts and ends, and thus knows the exact step count $|\sigma(\vec{S})|$ of each query.

In ORAM, $|\sigma(\vec{S})| = |\vec{S}|$, so $|\sigma(\vec{S})|$ reveals nothing new to the adversary. In TOM, $|\vec{S}|$ and $|\sigma(\vec{S})|$ may differ, so $|\sigma(\vec{S})|$ may leak information. For example, if $|\sigma(\vec{S})| < |\vec{S}|$, the adversary may infer that $\vec{S}$ contains repeated accesses. We limit such leakage by forcing $|\sigma(\vec{S})|$ to assume one of λ *milestone* values taken from a predefined set $\mathcal{M}_{|\vec{S}|}$. More milestones improve flexibility in generating $\sigma(\vec{S})$ and thus improve efficiency, but also leak more information about $\vec{S}$.

$\mathcal{M}_{|\vec{S}|}$ is defined up-front for each value of $|\vec{S}|$, so the milestones themselves do not leak information. Since the adversary knows $|\vec{S}|$, he already knows that $|\sigma(\vec{S})|$ will

be one of the λ milestones. Thus, he only learns information through the specific choice of milestone used for $|\sigma(\vec{S})|$. Equivalently, he learns which of λ equivalence classes $\vec{S}$ belongs to, limiting information leakage by the size of λ.

We now define security for λ-TOM, which translates a secret access pattern $\vec{S}$ into a public access pattern with one of λ milestone step counts.

Definition 20. A protocol satisfies λ-TOM security if both of the following conditions hold for every possible pair of secret access patterns $\vec{S_1}$ and $\vec{S_2}$:

1. Let $\ell = |\vec{S_1}|$. If $|\vec{S_1}| = |\vec{S_2}|$ then $|\sigma(\vec{S_1})|, |\sigma(\vec{S_2})| \in \mathcal{M}_\ell$, where $\mathcal{M}_\ell$ is a set of *milestones* of cardinality at most λ.
2. If $|\sigma(\vec{S_1})| = |\sigma(\vec{S_2})|$, then $P(\vec{S_1})$ and $P(\vec{S_2})$ are computationally indistinguishable (to every observer other than the client).

By ensuring that any two public access patterns with the same step count are indistinguishable, we guarantee that information about $\vec{S}$ only leaks through the observation of the step count $|\sigma(\vec{S})|$, which is in turn limited to one of λ milestones. We can bound the information leakage I_λ of a λ-TOM protocol by assuming the worst case, in which all milestones are equi-probable, giving:

Definition 21. A λ-TOM protocol leaks at most $I_\lambda \leq \log_2 \lambda$ bits per query.

When $\lambda = 1$, the leakage is $I_\lambda = 0$, which indicates that 1-TOM is as strong as ORAM. In fact, for $\lambda = 1$, we have by Condition 1 of Definition 20 that $|\vec{S_1}| = |\vec{S_2}|$ implies $|\sigma(\vec{S_1})| = |\sigma(\vec{S_2})|$, and thus by Condition 2 that $|\vec{S_1}| = |\vec{S_2}|$ implies $P(\vec{S_1})$ and $P(\vec{S_2})$ are indistinguishable. Therefore any 1-TOM protocol satisfies ORAM security (Definition 17). The reverse is also true for any ORAM with a notion of steps. In any case, we make no claim that 1-TOM is substantively more secure than ORAM, so we treat 1-TOM and ORAM as equivalent.

Since each query leaks at most I_λ bits, larger queries leak less information per access. Combining small, independent queries would reduce leakage, but may also increase latency. It is critical that no query results be released to the client until the entire query is satisfied. If the client used partial results, the partial completion time might leak, revealing additional information. Thus, query size is limited by the size of the results cache allocated to the TOM, and excessively large queries may need to be broken up. In standard ORAM, $I_\lambda = 0$, so there is no motivation to make queries larger than a single block access.

What the adversary *gains* from leaked access pattern information depends heavily on what other information the adversary holds. Other schemes that obscure access patterns (e.g. [30]) focus on quantifying the adversary's inability to make particular inferences, but do not assess holistic information loss. In contrast, we upper-bound the total access pattern information leakage, and leave it to the client to decide how much leakage is acceptable given the application.

6.3.4 Paddable TOM Protocols

We now show how to construct a λ-TOM for any given λ from a *paddable* TOM. Intuitively, we start by choosing λ milestones, then delay each query's completion by silently padding it with dummy steps until its step count reaches a milestone. We use S_{MAX} to denote the worst-case per-access step count.

Definition 22. A protocol is a *Paddable TOM* if it satisfies the following:

1. Condition 2 of Definition 20 (indistinguishable public access patterns)
2. It has finite upper bound $\ell \cdot S_{\text{MAX}}$ on step count $|\sigma(\vec{S})|$ generated from any secret access pattern of length $\ell = |\vec{S}|$.

3. Any $P(\vec{S})$ may be *padded* by adding any number of *dummy* steps, increasing $|\sigma(\vec{S})|$ by any amount.

We can coerce any paddable TOM into satisfying λ-TOM for any given λ. We first define appropriate milestones for $\mathcal{M}_\ell$, then instruct the protocol to pad every public access pattern with dummy steps, increasing $|\sigma(\vec{S})|$ to the smallest milestone in $\mathcal{M}_\ell$ greater than or equal to the original step count. If we trivially set $\mathcal{M}_\ell = \{\ell \cdot S_{\text{MAX}}\}$, and translate every secret access pattern of length ℓ, with padding, into a public access pattern with step count $\ell \cdot S_{\text{MAX}}$, we satisfy 1-TOM and thus ORAM security.

Efficient paddable protocols will often generate step counts much smaller than $\ell \cdot S_{\text{MAX}}$, so the padding required to reach $\ell \cdot S_{\text{MAX}}$ may incur substantial bandwidth cost. Increasing λ (adding milestones) can reduce cost, but also reduces privacy. To make the best possible tradeoffs, our strategy for choosing milestones should minimize cost due to padding for any given λ.

6.3.5 Log-Spacing Strategy for Paddable Protocols

Let $m = |\sigma(\vec{S})|$ be the original step count generated from query $\vec{S}$ of length $\ell = |\vec{S}|$. We may have $m < \ell$ if most queried blocks are cached, but such cases are too rare to merit dedicated milestones, so we assume $\ell \leq m \leq \ell \cdot S_{\text{MAX}}$.

Let m' be the smallest milestone in $\mathcal{M}_\ell$ such that $m' \geq m$. In a paddable TOM, the fractional increase in step count, and thus bandwidth cost, is given by the *padding factor* m'/m. Let δ be the maximum padding factor (maximum possible value of m'/m). Given λ, we propose to minimize δ by log-spacing milestones as multiples of ℓ over $[\ell, U(\ell)]$:

$$\mathcal{M}_\ell = \left\{ \left\lceil \ell \left(S_{\text{MAX}}\right)^{i/\lambda} \right\rceil \mid i \in \mathbb{Z}, 1 \leq i \leq \lambda \right\}. \tag{6.1}$$

This spacing strategy minimizes the maximum padding factor δ, ensuring:

$$\delta \leq \left\lceil (S_{\text{MAX}})^{1/\lambda} \right\rceil. \tag{6.2}$$

To minimize λ for given δ, we solve $(S_{\text{MAX}})^{1/\lambda} \leq \delta$ for λ:

$$\lambda \geq \frac{\log S_{\text{MAX}}}{\log \delta} = \log_{\delta} S_{\text{MAX}}. \tag{6.3}$$

These expressions reveal a clear tradeoff between privacy (λ) and efficiency (δ). Smaller S_{MAX} can improve privacy *and* efficiency, which is unsurprising since ORAMs fix privacy at $\lambda = 1$ and seek to reduce the worst-case per-access cost.

6.4 Staggered-Bin TOM

Here we present a novel λ-TOM protocol, called *Staggered-Bin TOM* (SBT), that reduces costs even for large queries that are not cache-friendly. In Section 6.5 we propose three read-only variants of SBT that store multiple copies of each block and reduce costs by choosing the most convenient copy to fetch. Table 6.2 gives some key notation, and Table 6.3 compares performance of SBT variants.

As noted in Section 6.3.3, an ORAM is simply a 1-TOM. TOM allows us to decouple steps from secret accesses, so we could improve on ORAM performance by simply increasing λ and adding a local block cache. We could then satisfy most cached block accesses without stepping the λ-TOM (without block transfers), while leaking only $\log_2 \lambda$ bits per multi-block query. However, caching only improves performance when secret access patterns exhibit temporal locality.

6.4.1 SBT Architecture

An SBT contains N blocks of B bits each placed in $n+1$ logical *bins*, each with a maximum capacity of n blocks. We initialize the SBT by filling the bins with $n, n-1, \ldots, 1, 0$ blocks, respectively, and storing them on the server. The SBT always keeps n more blocks locally, for $N = n(n+3)/2$ blocks total (Figure 6.2).

We choose n to be the smallest integer such that $N \leq n(n+3)/2$, and add up to n extra data blocks to increase SBT capacity N to exactly $n(n+3)/2$. *No unusable dummy blocks of any kind are added*, keeping server storage overhead minimal. Bins are purely logical structures, so the server is free to use any physical configuration for storing blocks.

The SBT needs local (client-side) storage space for three purposes. First, it requires Bn bits for the n blocks always stored locally. Second, it needs $B\ell$ bits to cache the results of an ℓ-block query, so that all ℓ blocks can be simultaneously released to the client. Finally, as in [77], the SBT needs a small amount of space for each of the N blocks to record its server ID, containing bin's index, and a list of block addresses in each bin, for a total of roughly $2\log_2 N$ bits per block.

In all, approximately $B(n+\ell) + 2N\log_2 N$ bits of client storage are required. Though these storage requirements may seem high, [77] and [76] note that B is large enough in practice that the space needed to store $n \approx \sqrt{2N}$ blocks is comparable to the

Table 6.2: SBT and TOM Notation

δ	Maximum padding factor (cost increase due to padding)
H	Maximum fetch queue length, before padding
All queries, strict upper bound:	
S_{MAX}	Worst-case per-access step count, before padding
C_{MAX}	Worst-case (per-access) bandwidth cost, after padding
Large uniform random block queries, high prob. upper bound:	
C_{HP}	High-probability (per-access) bandwidth cost, after padding

Table 6.3: Comparison of our $\lambda - TOM$ protocols, block size B. Numbers are approximate; estimated average costs taken from Figures 6.4–6.6. Smaller C_{MAX} improves privacy/efficiency tradeoff. λ set to log of S_{MAX} to keep δ constant. $\lg \equiv \log_2$

Protocol	Worst-Case C_{MAX}	Uniform Rand. Block C_{HP} for $\ell \approx 4\sqrt{N}$	Bits Leaked Per ℓ-Block Query	Efficient Write	Server Storage (Bits)	Client Storage (Bits)
Unprotected	1	1	$\ell \lg N$	Yes	NB	$O(1)$
ORAM [77]	$\lg N$	$\lg N$	0	Yes	$\leq 4NB$	$3B\sqrt{N} + 1.25N \lg N$
SBT	$2\sqrt{2N}$	$O\left(\frac{\log N}{\log \log N}\right)$	$\lg \lg(\sqrt{2N})$	Yes	NB	$(\ell + \sqrt{2N})B + 2N \lg N$
2-Choice SBT	$4\sqrt{N}$	$O(1)$, $\approx$ (3–5)	$\lg \lg(2\sqrt{N})$	No	$2NB$	$(\ell + 2\sqrt{N})B + 4N \lg N$
SBT+ORAM	$3 \lg N$	$O(\log \log N)$	$\lg \lg(3 \lg N)$	No	$\leq 5NB$	$(\ell + \sqrt{2N} + 3\sqrt{N})B + 3.25N \lg N$
Multi-SBT	$3 \lg N$	$O(1)$, $\approx$ (4–7)	$\lg \lg(3 \lg N)$	No	$\leq 6NB$	$(\ell + 5\sqrt{N})B + 5.25N \lg N$

space needed for the meta-data of all N blocks. For example, with $N = 2^{30}$ blocks, and block size $B = 64$KB, we need under 8GB for the meta-data, and up to 8GB local block storage for queries of $\ell = 3\sqrt{N}$ blocks.

6.4.2 SBT Operation

Each step in the SBT fetches one block from and stores one block to the server. SBT operation is best described in terms of *passes* of n steps each. A pass fetches and removes one block from each of the n non-empty bins in order, and stores n blocks to the previously empty bin. After each pass, the bin load pattern rotates by 1 bin, and fetches continue round-robin (Figure 6.3). After each pass, the SBT re-encrypts the n fetched blocks, randomly permutes them, assigns them to the empty bin, and generates new server-side IDs to prevent linking to old copies.

Each query consists of ℓ secret accesses for distinct block addresses. A query may begin or end at any point during a pass. The SBT cannot change the one-per-bin round-robin fetch pattern, but may choose which block to fetch from each bin. Thus, a single-block query can always be satisfied in n steps, since we fetch at least one block from every non-empty bin. Similarly, any ℓ-block query takes at most ℓn steps

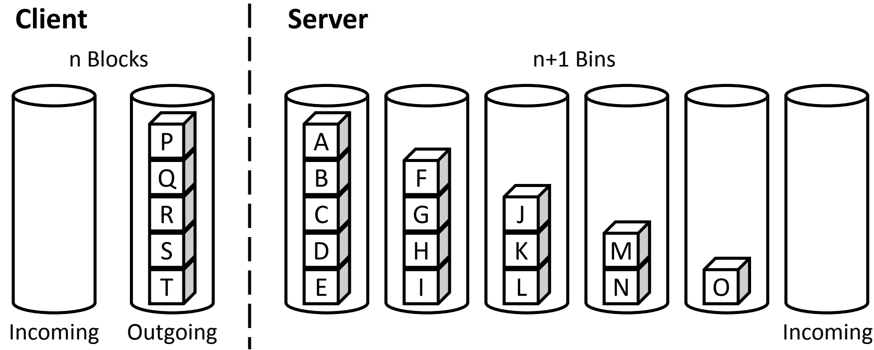

Figure 6.2: The SBT in its initial state, with $n = 5$ blocks on the client, and $n(n+1)/2$ on the server. The empty server-side *incoming* bin will be filled in, one block at a time, by the n blocks from the client.

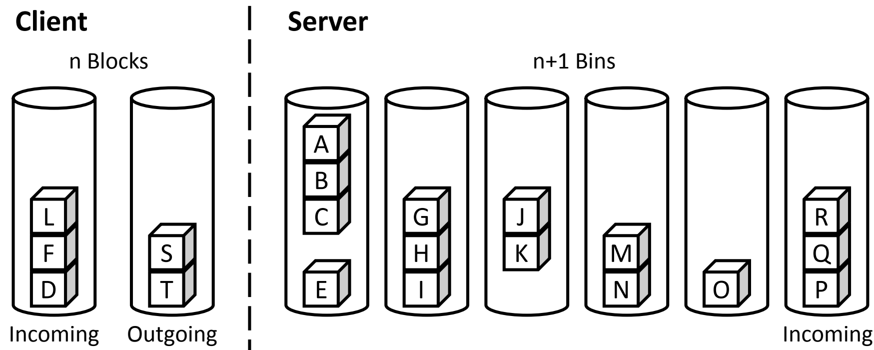

Figure 6.3: SBT after 3 steps. Server bins are accessed in a round-robin fashion. Blocks L, F, D have been fetched to the client-side *incoming* bin, and blocks R, Q, P stored to server-side *incoming* bin.

($S_{\text{MAX}} = n$), since we always retrieve at least one target block per pass, even if all ℓ blocks are in one bin.

The SBT maintains a *fetch queue* for each bin. To start a query, we identify the bin containing each block to be accessed, and add an appropriate fetch to that bin's fetch queue. When the SBT is ready to issue a fetch for bin i, it first checks i's fetch queue. If the queue is non-empty, the next fetch is dequeued and dispatched to the server. Otherwise, a dummy fetch is generated for a randomly chosen block from the bin. Once all fetch queues are empty and all outstanding fetches finish, the query is satisfied and results are released to the client.

All fetches must proceed in order, as must all stores. Further, to maintain n blocks on the client, a given step's store cannot begin until its fetch completes. However, stores may trail their corresponding fetches as much as necessary to ensure full network bandwidth utilization. That is, we may initially let fetches get several steps ahead of stores, so that many stores and fetches run concurrently.

6.4.3 SBT Security

We now show that SBT meets the Paddable TOM criteria in Definition 22. We have shown that SBT has a finite step count upper bound ℓn, so it remains to show that public access patterns with the same step count are indistinguishable (Condition 2 of Definition 20), and that public access patterns may be padded.

Theorem 7. *In the SBT, for any two public access patterns $P(\vec{S_1}), P(\vec{S_2})$, if $|\sigma(\vec{S_1})| = |\sigma(\vec{S_2})|$, then $P(\vec{S_1})$ and $P(\vec{S_2})$ are computationally indistinguishable.*

Proof. First, the order in which the SBT fetches from and stores to bins is fixed. Hence, any two public access patterns with the same step count must make fetches and stores to and from exactly the same sequence of bins.

Store Patterns: After each pass, the locally-stored bin of blocks to be sent to the server is randomly permuted and re-encrypted using a semantically secure encryption scheme and a fresh nonce. Blocks are then stored to the server in their permuted order. Re-encryption ensures that the server cannot distinguish the blocks. Random permutation ensures that blocks are always stored in a uniformly random order, independent of their fetch order. Thus any two *store* patterns of the same length are computationally indistinguishable.

Fetch Patterns: Since the blocks within each bin were randomly permuted, each block's location in the bin is independent of its data and any prior accesses. Thus each fetch is indistinguishable from a uniformly random choice from the bin's remaining blocks, and any two fetches from one bin are indistinguishable. Thus any two *fetch* patterns of the same length are indistinguishable.

Thus, since $P(\vec{S_1})$ and $P(\vec{S_2})$ have the same step count, and there is exactly one fetch and store per step, their fetch and store patterns each have the same length and are indistinguishable. Since both fetches and stores are indistinguishable, and the pattern of when to issue fetches and stores is predetermined, $P(\vec{S_1})$ and $P(\vec{S_2})$ are themselves computationally indistinguishable. □

Theorem 8. *Any public access pattern generated by SBT may be* padded *by adding any number of dummy steps.*

Proof. We can pad any public access pattern in SBT with any number d of additional steps by issuing d *dummy* fetches for randomly chosen blocks from each of the next d bins, along with their corresponding stores. □

Theorems 7 and 8 establish that SBT is a Paddable TOM, as per Definition 22. Thus we can coerce SBT into satisfying $\lambda - TOM$ for any λ. In particular, we apply the log-spacing strategy of Section 6.3.5 to choose the λ milestones in $\mathcal{M}_\ell$. The smaller our choice of λ, the greater our privacy but the poorer our performance. By Equation 6.2, for a given λ, we have a maximum padding factor:

$$\delta \leq \left\lceil (S_{\text{MAX}})^{1/\lambda} \right\rceil = \left\lceil n^{1/\lambda} \right\rceil \leq (2N)^{1/2\lambda}. \tag{6.4}$$

Similarly, by Equation 6.3, for a δ, we get a minimum milestone count λ given by:

$$\lambda \geq \log_\delta S_{\text{MAX}} = \log_\delta n. \tag{6.5}$$

6.4.4 SBT Performance

The upper-bound on the SBT's per-access step count is given by $S_{\text{MAX}} = n$, so the upper-bound bandwidth cost is given by $C_{\text{MAX}} = 2n \leq 2\sqrt{2N}$. We now determine the bandwidth cost C_{HP} that holds with high probability for large, uniform random block queries, which are queries composed of ℓ block addresses chosen uniformly at random, without replacement.

The size of each fetch queue decreases by at most one during a given pass, so the number of passes needed to satisfy a query depends on the initial length of the longest fetch queue. We use H to denote the maximum length of the longest fetch queue. The query generates step count roughly nH without padding. In the best case, each fetch queue is nearly the same length, and in the worst case all fetches are in the same queue, so we know that $\lceil \ell/n \rceil \leq H \leq \ell$.

Theorem 9. *Let $\ell \geq n$ (large queries). With high probability for the SBT with uniform random block queries, we have:*

$$H \in O\left(\frac{\ell}{n}\frac{\log n}{\log\log n}\right).$$

We prove this theorem in Section 6.6, using the observation that we can bound H by bounding the maximum urn height in the well-known *balls and urns* problem [46, 66], where balls are thrown into urns uniformly at random, with replacement. Thus, for uniform random block queries with $\ell \geq n$, the bandwidth cost, with high probability, is given by:

$$C_{\text{HP}} \in O\left(\delta\frac{n}{\ell}\frac{\ell}{n}\frac{\log n}{\log\log n}\right) \subseteq O\left(\frac{\delta\log N}{\log\log N}\right), \tag{6.6}$$

with constant δ for at least $\lambda \in \Omega(\log N)$ milestones. Thus SBT is able to satisfy large queries that are not cache-friendly with a lower asymptotic cost than the best existing

ORAM protocols (cost $O(\log N)$) while leaking only $I_\lambda \in O(\log\log N)$ bits per ℓ-block query.

Since our focus is on queries that are not cache-friendly, SBT's effectiveness is limited when queries *are* cache-friendly (query blocks repeat frequently). When a large query is satisfied, most fetched blocks will be stored in only H bins. If the same query repeats before bins disperse, the new query will have maximum queue length at least ℓ/H. Thus, the better the original query's performance, the poorer the repeat's performance. Fortunately, caching mitigates such disadvantages.

6.5 SBT Variants

We now propose three read-only SBT variants: *2-Choice SBT*, *SBT+ORAM*, and *Multi-SBT* (a combination of 2-Choice SBT and SBT+ORAM). These variants store multiple copies of each block and fetch the most convenient copy available, reducing bandwidth cost to as little as $C_{\text{MAX}} = 3\log_2 N$ in the worst case, and $C_{\text{HP}} \in O(1)$ for large uniform random block queries (see Table 6.3).

Read-only means that the client cannot update the contents of any of his blocks. However, blocks must still be re-encrypted and stored back to the server to preserve privacy. Writes *can* be supported, but would require all copies of a block to be updated, making writes substantially more expensive than reads.

6.5.1 The 2-Choice SBT Variant

We construct the 2-Choice SBT by creating two copies of each of the N data blocks, and adding them all to a single SBT with capacity $2N$, which treats both copies as independent blocks. The key difference from SBT is that when the 2-Choice SBT needs

to read block a, it may choose to fetch the block from either of 2 bins. It is possible that both copies of a will be in the same bin, but this state is rare and transient, persisting only until either copy is fetched.

For a given query, each of the ℓ secret block accesses yields a fetch that is assigned to one of two bins' fetch queues. We want to optimize the assignment of fetches to bins, reducing the maximum queue length H. Since we know the entire query and the block-bin mapping, the optimization resembles the *optimal multi-choice allocation* [15], and *offline Cuckoo hashing* [59] problems.

6.5.1.1 Random Round Robin Algorithm

We optimize fetch assignments using the iterative *Random Round Robin* (*RRR*) algorithm proposed in [15]. We describe RRR briefly, replacing *balls* with block *fetches* and *bins* with fetch *queues*.

We first guess a target maximum queue length H', starting with the minimum $H' = \lceil \ell/n \rceil$. We then run RRR to try to find an assignment of fetches to queues with actual $H \leq H'$. If the attempt fails, we increment H' and repeat. In practice, we rarely expect more than two iterations [15], so we fix a maximum iteration count $r = 5$, after which we return the best available result. The iterative RRR runs efficiently, requiring time and space in $O(r(n + \ell))$.

For each RRR iteration, each fetch starts out *uncommitted*: assigned to the queues of both bins containing its block. When we *commit* a fetch to a queue, we irreversibly remove it from its other queue. We identify any queue q with length at most H', and commit to q all its uncommitted fetches. The intuition is that since q's length is at most H' and cannot increase, it should accept its current assignment, freeing as

many fetches as possible from other queues. Any time we remove a fetch from a queue, we repeat this check.

We continue by stepping through all remaining queues with uncommitted fetches, for each queue randomly choosing one uncommitted fetch to commit to the queue, followed by the length check. We continue stepping through queues until all fetches are committed. If any queues are left with more than H' fetches, the RRR iteration is declared a failure.

6.5.1.2 2-Choice SBT Security

To an observer, the 2-Choice SBT behaves just like the SBT, except for its higher capacity. Thus, 2-Choice SBT's security follows from the arguments for SBT security in Section 6.4.3. In particular, 2-Choice SBT meets the criteria of Definition 22 for a Paddable TOM Protocol with worst-case per-access step count $S_{\text{MAX}} = n \leq 2\sqrt{N}$. Thus, it can be coerced to $\lambda - TOM$ using the log-spacing strategy. By Equation 6.2, the maximum padding factor δ is given by $\delta \leq \lceil n^{1/\lambda} \rceil \leq (4N)^{1/(2\lambda)}$. For a given δ, Equation 6.3 gives $\lambda \geq \log_\delta n$.

6.5.1.3 2-Choice SBT Performance

Since $S_{\text{MAX}} \leq 2\sqrt{N}$, we have $C_{\text{MAX}} \leq 4\sqrt{N}$.

Conjecture 10. *With high probability, the 2-Choice SBT with uniform random block queries gives $H \in O(\ell/n + 1)$.*

For a related balls and urns problem, the authors in [15] show empirically that *RRR* yields maximum urn height in $O(\ell/n + 1)$, with performance nearly indistinguishable from the more complex *Selfless Algorithm*, which is proven to have maximum height $O(\ell/n + 1)$ with high probability. While we cannot provide a formal proof of Conjecture

10, we give a detailed argument supporting it in Section 6.6, and show in Section 6.7 that it is borne out by our experiments.

Assuming Conjecture 10, the bandwidth cost of the 2-Choice SBT used on uniform random block queries with $\ell \geq n$ is, with high probability:

$$C_{\mathrm{HP}} \in O\left(\delta\frac{n}{\ell}\left(\frac{\ell}{n}+1\right)\right) \subseteq O(\delta), \tag{6.7}$$

with constant δ for $\lambda \in \Omega(\log N)$. Thus, for large uniform random block queries, the 2-Choice SBT is highly efficient, and leaks only $I_\lambda \in O(\log \log N)$ bits/query.

Relative to SBT, the 2-Choice SBT doubles the storage space required for the server ($2N$), and increases required client block storage from $\sqrt{2N}$ to $2\sqrt{N}$ blocks and client index space from about $2N \log_2 N$ to nearly $4N \log_2 N$ bits.

6.5.2 The SBT+ORAM Variant

We construct the SBT+ORAM by merging a SBT with any efficient ORAM. We store one copy of each block in the SBT and in the ORAM, and run both protocols in parallel. To read a block, we either fetch it using the SBT, or read it using a single ORAM step. For now we use the practical ORAM of [77] due to its low bandwidth cost of roughly $\log_2 N$ block transfers per secret access.

For each query, we first assign all fetches to the SBT component's fetch queues and let it run normally. After every $\log_2 N$ SBT steps, we remove one fetch from the current longest fetch queue and re-assign it to the ORAM.

6.5.2.1 SBT+ORAM Security

The ORAM component advances one step for every $\log_2 N$ SBT steps. Thus, in the worst case where we rely strictly on the ORAM, we need $\ell(1+\log_2 N)$ total steps to satisfy a query of length ℓ, so the per-access step count is bounded by $S_{\mathrm{MAX}} = 1+\log_2 N$.

We now show that SBT+ORAM satisfies the indistinguishability and paddability conditions of a Paddable *TOM*.

Theorem 11. *In SBT+ORAM, for any public access patterns $P(\vec{S_1}), P(\vec{S_2})$, if $|\sigma(\vec{S_1})| = |\sigma(\vec{S_2})|$, then $P(\vec{S_1})$ and $P(\vec{S_2})$ are computationally indistinguishable.*

Proof. Since ORAM uses exactly one step per secret access, two public access patterns with the same step count have the same secret access pattern length. Thus, by Definition 17, any two public access patterns with the same step count generated by the ORAM are indistinguishable. By Theorem 2, any two public access patterns with the same step count generated by the SBT are also indistinguishable. Since the public access patterns generated by both protocols are indistinguishable, and the pattern of when to issue fetches from the SBT and the ORAM is predetermined, the SBT+ORAM's combined public access patterns $P(\vec{S_1})$ and $P(\vec{S_2})$ are indistinguishable. □

Using the log-spacing strategy gives $\delta \leq \lceil (S_{\text{MAX}})^{1/\lambda} \rceil = \lceil (1 + \log_2 N)^{1/\lambda} \rceil$ and $\lambda \geq \log_\delta S_{\text{MAX}} = \log_\delta (1 + \log_2 N)$ (Equations 6.2, 6.3). Since S_{MAX} is smaller for SBT+ORAM than SBT, the privacy/efficiency tradeoff is more favorable. In particular, to limit padding to $\delta = 2$, we need only $\lambda \approx \log_2 \log_2 N$ milestones.

6.5.2.2 SBT+ORAM Performance

We incur $\log_2 N$ block transfers for each ORAM step and 2 transfers for each SBT step. In the worst-case, we make ℓ ORAM steps and $\ell \log_2 N$ SBT steps for an ℓ-block query, giving $C_{\text{MAX}} \leq 3 \log_2 N$.

We know from Theorem 9 that for large uniform random block queries, the SBT has maximum fetch queue length in $O(\ell \log N / n \log \log N)$. However, the expected queue length is only ℓ / n, so we rightly expect that relatively few queues have such

large lengths. Though the ORAM runs slowly, focusing it on the largest queues first asymptotically reduces the final maximum queue length H.

Theorem 12. *Let $\ell \geq n$ and $N \geq 32$. With high probability for the SBT+ORAM with uniform random block queries we have $H \in O((\ell/n) \log\log N)$.*

In Section 6.6, we present a proof for Theorem 12 based on a novel balls and urns analysis. Thus, the bandwidth cost of the SBT+ORAM used on uniform random block queries with $\ell \geq n, N \geq 32$ is, with high probability:

$$C_{\text{HP}} \in O\left(\delta \frac{n}{\ell} \frac{\ell}{n} \log\log N\right) \subseteq O\left(\delta \log\log N\right), \tag{6.8}$$

with constant δ for only $\lambda \in \Omega(\log\log N)$ milestones. Thus, for large uniform random block queries, the SBT+ORAM is more efficient than the SBT. At the same time, it leaks only $I_\lambda \in O(\log\log\log N)$ bits per query, yielding better privacy than 2-Choice SBT, but slightly higher C_{HP}.

The server storage costs of [77] are reported at roughly $4BN$ bits, and we estimate that client storage is $1.25 \log_2 N + 3B\sqrt{N}$ bits, based on results in Table 2 of [77]. Thus, the SBT+ORAM has a total server storage cost of roughly $5BN$, and client storage $(\ell + \sqrt{2N} + 3\sqrt{N})B + 3.25N \log_2 N$ bits.

6.5.3 The Multi-SBT Variant

The Multi-SBT replaces the SBT in a SBT+ORAM with a 2-Choice SBT. Thus, the Multi-SBT stores a total of three copies of each block. Its security follows directly from the security of 2-Choice SBT and SBT+ORAM.

The Multi-SBT inherits SBT+ORAM's excellent worst-case per-access step count $S_{\text{MAX}} = 1 + \log_2 N$ and bandwidth cost $C_{\text{MAX}} = 3 \log_2 N$. For large random block queries, it also inherits 2-Choice SBT's high-probability bandwidth cost:

$$C_{\mathrm{HP}} \in O(\delta), \tag{6.9}$$

while requiring only $\lambda \in \Omega(\log\log N)$ milestones for constant δ, and leaking only $I_\lambda \in O(\log\log\log N)$ bits per query. Thus, for uniform random block queries of any size, the Multi-SBT requires only $O(\ell + \log_2 N)$ block transfers!

Multi-SBT combines the best performance and privacy characteristics of 2-Choice SBT and SBT+ORAM, and can easily outperform both. Even in worst cases, the Multi-SBT incurs at most 3 times the bandwidth cost of SBT+ORAM, or 1.5 times the cost of SBT. Multi-SBT requires total server storage of roughly $6BN$, and client storage of roughly $B\sqrt{N}(3+\sqrt{2}) + 5.25N\log_2 N$ bits.

6.6 Performance Analyses and Proofs

Here we prove Theorems 9 and 12, and argue for Conjecture 10. In each case, our goal is to upper-bound the maximum fetch queue length H — the maximum number of blocks that must be fetched from any one bin by the SBT component to satisfy a query. Equivalently, H is the maximum number of SBT *passes* needed to satisfy a query.

For simplicity, we assume the SBT is at the start of a pass, so we are given n bins filled with $1, 2, \ldots, n$ blocks each.[1] Each query requests ℓ distinct blocks chosen uniformly at random, without replacement, from the set of all N blocks. We call such queries *uniform random block queries*. Every block has a unique *location*. Of the $N = n(n+3)/2$ blocks, n are located somewhere in the local cache, and the remaining $n(n+1)/2$ are located somewhere in one of the n bins. Requests for cached blocks are satisfied instantly.

[1] We can force the SBT to the start of a pass for each query, which increases H by at most 1 and thus does not affect our asymptotic analysis.

We assign each queried block a unique index i between 1 and ℓ. Let $\Pr_B(i,j)$ be the maximum probability that block i will be in bin j, given any possible arrangement of the remaining queried blocks. The maximum $\Pr_B(i,j)$ occurs when j is the n-block bin, and the other $\ell - 1$ blocks are located in bins other than j. In this case, i has $N - \ell + 1$ possible locations, n of which are in bin j, giving:

$$\Pr_B(i,j) \leq \frac{n}{N - \ell + 1} \leq \frac{n}{N - \ell}. \tag{6.10}$$

A great deal of work has been done on the closely-related *balls and urns* problem (e.g. [46, 66]), in which balls are thrown independently into one of several urns chosen uniformly at random (with replacement).[2] There are well-known bounds on the resulting maximum urn occupancy. To use these bounds, we first reduce our blocks and bins problem to a larger balls and urns problem.

6.6.1 Problem Transformation

Consider the balls and urns problem with 3ℓ balls, where 3 distinct balls are given each index $1 \leq i \leq \ell$. We throw these 3ℓ balls independently and uniformly at random into n urns. Let $\Pr_U(i,j)$ be the probability that at least one ball with label i will appear in urn j, which is given by:

$$\Pr_U(i,j) = 1 - \left(\frac{n-1}{n}\right)^3 = \frac{3n^2 - 3n + 1}{n^3} \tag{6.11}$$

Intuitively, if $\Pr_B(i,j) \leq \Pr_U(i,j)$, then a ball labeled i is at least as likely to be placed in urn j as block i is to be located in bin j, and so the number of blocks found in bin j should be no larger than the number of balls in urn j. If we can show that $\Pr_B(i,j) \leq \Pr_U(i,j)$ for every i, j, then any upper-bound on the maximum urn

[2]Such problems are commonly referred to as *balls and bins* problems, but since we use *bin* in our SBT construction, we use *urn* here for clarity.

occupancy in the balls and bins problem should hold for the maximum queue length H in the blocks and bins problem.

Lemma 13. $\Pr_B(i,j) \leq \Pr_U(i,j)$ *holds for all* $\ell \leq n^2/6$.

Proof. Substituting $N = n(n+3)/2$, we get:

$$\Pr_B(i,j) \leq \frac{n}{N-\ell} = \frac{2}{n+3-2\ell/n}.$$

Thus we have

$$\begin{aligned}
\Pr_B(i,j) \leq \Pr_U(i,j) \;&\Longleftarrow\; \frac{2}{n+3-2\ell/n} \leq \frac{3n^2-3n+1}{n^3} \\
\Longleftrightarrow\; & 2n^3 \leq 3n^3 + 6n^2 - 8n + 3 - \ell(6n - 6 + 2/n) \\
\Longleftrightarrow\; & \ell\,(6n - 6 + 2/n) \leq n^3 + 6n^2 - 4n + 3 \\
\Longleftarrow\; & \ell \leq n^2/6
\end{aligned}$$

□

6.6.2 SBT Analysis

We are now ready to prove Theorem 9.

Theorem 9. Let $\ell \geq n$ (large queries). With high probability for the SBT with uniform random block queries, we have:

$$H \in O\left(\frac{\ell}{n}\frac{\log n}{\log\log n}\right).$$

Proof. It is well known (e.g. [66]) that if we throw n balls independently and uniformly at random into n urns, we get a maximum urn occupancy in $O(\log n/\log\log n)$ with high probability. Thus, if we throw $m \geq n$ balls, we get a maximum height $O((m\log n)/n\log\log n)$. By Lemma 13, when $\ell \leq n^2/6$, an upper-bound on the maximum urn occupancy for $m = 3\ell$ balls and n urns applies to H, giving:

$$H \in O\left(\frac{3\ell\log n}{n\log\log n}\right) \subseteq O\left(\frac{\ell}{n}\frac{\log n}{\log\log n}\right), \text{ for } \frac{n}{3} \leq \ell \leq \frac{n^2}{6}$$

Further, since $H \leq n$, for $\ell > n^2/6$ we have trivially that: $H \in O\left(\frac{\ell}{n}\right) \subseteq O\left(\frac{\ell}{n}\frac{\log n}{\log\log n}\right)$.

□

6.6.3 2-Choice SBT Analysis

Recall Conjecture 10 from Section 6.5.1.3:

Conjecture 10. With high probability for the 2-Choice SBT with uniform random block queries, we have:

$$H \in O\left(\ell/n + 1\right).$$

Authors in [15] analyze the *Selfless Algorithm* for allocating m balls to n urns, where each ball may be placed in either of two urns chosen uniformly at random. They show, analytically, that the Selfless Algorithm yields a maximum final urn occupancy $U' \in O(\lceil m/n \rceil) \subseteq O(m/n + 1)$ with high probability. They also show empirically that the simpler *Random Round Robin* algorithm, which we use for the 2-Choice SBT, has nearly equivalent performance.

As we did for SBT, we can think of the 2-Choice SBT's blocks and bins problem as a balls and urns problem where we throw $m = 3\ell$ balls into n urns. However, since each block now belongs to two bins, and can thus be added to either of two fetch queues, the corresponding ball may be placed in either of two urns, but need not be placed in both. Though Lemma 13 no longer holds, we appeal to the intuition that a bound on the maximum urn occupancy H' is likely to hold for the maximum fetch queue height H as well.

We therefore contend that $H \approx U' \in O(m/n + 1) \subseteq O(\ell/n + 1)$. Clearly, this argument is far from a proof, both because *Random Round Robin* has not been fully analyzed, and because of the different models used for the two-choice blocks and bins

and two-choice balls and urns problems. However, we observe empirically that 2-Choice SBT does in fact appear to follow $H \in O(\ell/n + 1)$, as evidenced by Figure 6.13 in Section 6.7.

6.6.4 SBT+ORAM Analysis

In SBT+ORAM, the ORAM component assists the SBT by removing one block from the longest fetch queue after every $\log_2 N$ SBT steps, potentially reducing the required number of passes H. We can transform the problem into a balls and urns problem with $m = 3\ell$ balls and n urns, where we remove $\ell/(\log_2 N + 1)$ balls from the highest occupancy urns. As far as we know, we are the first to consider this problem, as prior work has only analyzed problems in which balls are removed randomly (e.g. [46]).

6.6.4.1 Balls and Urns Problem

We first consider the balls and urns problem of throwing m balls independently and uniformly at random into n urns, then deterministically removing $\rho m/\log_2 N$ balls. Removals take place by repeatedly removing a ball from the highest occupancy urn.

Let U_i be a random variable representing the original occupancy of a given urn i. That is, the number of balls in urn i before removals. Each U_i has the binomial distribution $Bin(m, 1/n)$. Let U be the maximum occupancy across all bins before removals, and U' the maximum occupancy after removals. We ultimately seek an upper-bound on U'.

Let $B_{i,k}$ be the number of balls originally in urn i in excess of $k-1$. That is, if urn i starts with d balls, then $B_{i,k} = d - k + 1$ if $d \geq k$, and $B_{i,k} = 0$ if $d < k$. Let $B_k = \sum_{j=1}^{n} B_{i,k}$. Intuitively, B_k is the minimum number of balls (minimum $\rho m/\log_2 N$) we must remove to get $U' < k$.

Outline: Our goal is to find an upper-bound on B_k such that for some $k \in O(\log\log n)$, $B_k \leq \rho m/\log_2 N$ with high probability, giving $U' \in O(\log\log N)$ with high probability. We first define the binomial tail probability $S(m,n,k)$ and its Chernoff upper bound. We then define a function $f(m,n,k)$ and express upper-bounds on $E[B_k]$ and $Var[B_k]$ in terms of $f(m,n,k)$. We then apply Chebyshev's inequality to upper-bound the probability that $B_k \geq \rho m/\log_2 N$. We then choose $k^* = (em/n)\log_2\log_2 N + 2$, and give the resulting upper-bound on $f(m,n,k^*)$. Finally, we show that for our chosen k^*, $B_{k^*} \leq \rho m/\log_2 N$ with high probability for $\rho \geq 1$, giving $U' < k^* \in O((m/n)\log\log N)$ for $m \geq n, N \geq 32$.

Let $S(m,n,k)$ be the binomial tail probability given by:

$$
\begin{aligned}
S(m,n,k) &= \Pr\left(Bin\left(m, \frac{1}{n}\right) \geq k\right) \\
&= \sum_{j=k}^{m} \binom{m}{j} \left(\frac{1}{n}\right)^j \left(1 - \frac{1}{n}\right)^{m-j}.
\end{aligned}
$$

We can upper bound $S(m,n,k)$ by applying Chernoff bounds (Equation 9 in [42]), giving:

$$
S(m,n,k) \leq e^{-(m/n)} \left(\frac{em}{nk}\right)^k, \;\; 1 < k < m, \;\; k \geq m/n. \tag{6.12}
$$

We now define $f(m,n,k)$, to be used in bounds below:

$$
f(m,n,k) = \frac{m}{n} e^{-(m-2)/n} \left(\frac{em}{n(k-2)}\right)^{k-2}. \tag{6.13}
$$

We now establish the following upper-bound on $E[B_k]$.

Lemma 14. *For $e + 1 < k \leq m, k \geq m/n + 1$, we have that $E[B_k] \leq \frac{m}{n} \cdot f(m,n,k)$.*

Proof.

$$\begin{aligned}
E[B_k] &= E\left[\sum_{i=1}^{n} B_{i,k}\right] = nE[B_{i,k}] \\
&= n\sum_{j=k}^{m}(j-k+1)\Pr(H_i = j) \\
&\leq n\sum_{j=k}^{m} j\Pr(H_i = j) \text{ for } k \geq 1 \\
&= n\sum_{j=k}^{m} j\binom{m}{j}\left(\frac{1}{n}\right)^{j}\left(\frac{n-1}{n}\right)^{m-j} \\
&= n\sum_{j=k-1}^{m-1}(j+1)\binom{m}{j+1}\left(\frac{1}{n}\right)^{j+1}\left(\frac{n-1}{n}\right)^{m-(j+1)} \\
&= n\sum_{j=k-1}^{m-1} m\binom{m-1}{j}\left(\frac{1}{n}\right)\left(\frac{1}{n}\right)^{j}\left(\frac{n-1}{n}\right)^{(m-1)-j} \\
&= mS(m-1,k-1,n) \\
&\leq me^{-(m-1)/n}\left(\frac{e(m-1)}{n(k-1)}\right)^{k-1} \\
&\leq mf(m,n,k) \quad (\text{for } e+1 < k \leq m, k \geq m/n+1).
\end{aligned}$$

□

We now upper-bound the variance $Var[B_k]$.

Lemma 15. *For $e+1 < k \leq m, k \geq m/n+2$, we have that*

$$Var[B_k] \leq 2mf(m,n,k).$$

Proof. We first observe that the variables $B_{i,k}, B_{j,k}$ for any two bins $i \neq j$ are negatively correlated. That is, increasing $B_{i,k}$ tends to decrease $B_{j,k}$, and vice-versa, so their covariances $Cov(B_{i,k}, B_{j,k}), Cov(B_{j,k}, B_{i,k})$ are always negative, giving:

$$
\begin{aligned}
Var[B_k] &= \sum_{i=1}^{n} Var(B_{i,k}) + \sum_{i,j,i\neq j}^{n} Cov(B_{i,k}, B_{j,k}) \\
&\leq nVar(B_{i,k}) + 0 \leq nE[B_{i,k}^2] \\
&= n\sum_{j=k}^{m} (j-k+1)^2 \Pr(H_i = j) \\
&\leq n\sum_{j=k}^{m} j^2 \binom{m}{j} \left(\frac{1}{n}\right)^j \left(\frac{n-1}{n}\right)^{m-j} \quad \text{for } k \geq 1 \\
&= m\sum_{j=k}^{m} j \binom{m-1}{j-1} \left(\frac{1}{n}\right)^{j-1} \left(\frac{n-1}{n}\right)^{m-j} \\
&= mS(m-1, k-1, n) + m\frac{m-1}{n} S(m-2, k-2, n) \\
&\leq me^{-((m-1)/n)} \left(\frac{e(m-1)}{n(k-1)}\right)^{k-1} \\
&\quad + \frac{m^2}{n} e^{-((m-2)/n)} \left(\frac{e(m-2)}{n(k-2)}\right)^{k-2} \\
&\leq 2mf(m,n,k) \quad (\text{for } e+1 < k \leq m,\ k \geq m/n + 2).
\end{aligned}
$$

□

Lemma 16. *For $e + 1 < k \leq m, k \geq m/n + 2$, we have that:*

$$
\Pr\left(B_k \geq \frac{\rho m}{\log_2 N}\right) \leq \frac{1}{m} \frac{2f(m,n,k)\log_2^2 N}{(\rho - f(m,n,k)\log_2 N)^2}
$$

Proof. We apply Chebyshev's inequality to bound the probability that B_k is at least $\rho m / \log_2 N$, giving:

$$
\begin{aligned}
&\Pr\left(B_k \geq \frac{\rho m}{\log_2 N}\right) \\
&\quad = \Pr\left(B_k - E[B_k] \geq \frac{\rho m}{\log_2 N} - E[B_k]\right) \\
&\quad \leq \Pr\left(|B_k - E[B_k]| \geq t\sqrt{Var[B_k]}\right), \\
&\qquad t = \frac{\frac{\rho m}{\log_2 N} - E[B_k]}{\sqrt{Var[B_k]}} \\
&\quad \leq \frac{1}{t^2} = Var[B_k] \left(\frac{1}{\frac{\rho m}{\log_2 N} - E[B_k]}\right)^2.
\end{aligned}
$$

Applying the results of Lemmas 14 and 15 gives:

$$\Pr\left(B_k \geq \frac{\rho m}{\log_2 N}\right) \leq \frac{2mf(m,n,k)}{\left(\dfrac{\rho m}{\log_2 N} - mf(m,n,k)\right)^2}$$
$$= \frac{1}{m}\frac{2f(m,n,k)\log_2^2 N}{(\rho - f(m,n,k)\log_2 N)^2}.$$

□

We now choose $k^* = (em/n)(\log_2 \log_2 N) + 2$.

Lemma 17. *For $m \geq n$, $N \geq 32$, and $k^* \leq m$, we have:*

$$f(m,n,k^*) \leq \left(\frac{1}{\log_2 N}\right)^3 \tag{6.14}$$

Proof. Given $k = k^*$, we get:

$$f(m,n,k^*) = \frac{m}{n}e^{-(m-2)/n}\left(\frac{1}{\log_2 \log_2 N}\right)^{(em/n)(\log_2 \log_2 N)}$$

Taking the log of both sides gives:

$$\log_2 f(m,n,k^*) = \left(\log_2 \frac{m}{n} - \frac{m-2}{n}\log_2 e\right) - \frac{em}{n}(\log_2 \log_2 N)(\log_2 \log_2 \log_2 N)$$
$$\leq -\frac{em}{n}(\log_2 \log_2 N)(\log_2 \log_2 \log_2 N)$$
$$(\text{for } N \geq 32, m \geq n).$$

Thus we have:

$$f(m,n,k^*) \leq \left(2^{-\log_2 \log_2 N}\right)^{(em/n)\log_2 \log_2 \log_2 N}$$
$$\leq \left(\frac{1}{\log_2 N}\right)^{(em/n)\log_2 \log_2 \log_2 N}$$
$$\leq \left(\frac{1}{\log_2 N}\right)^3 \quad (\text{for } N \geq 32).$$

□

Lemma 18. *With high probability, removing $\rho m/\log_2 N$ balls yields $U' \in O((m/n)\log\log N)$ for $m \geq n, N \geq 32, \rho \geq 1, k^* \leq m$.*

Proof. By Lemmas 16 and 17, we have that for $k^* \le m$:

$$\Pr\left(B_{k^*} \ge \frac{\rho m}{\log_2 N}\right) \le \frac{1}{m} \frac{2\frac{1}{\log_2^3 N}\log_2^2 N}{\left(\rho - \frac{1}{\log_2^3 N}\log_2 N\right)^2}$$

$$\text{(for } m \ge n, N \ge 32)$$

$$\le \frac{1}{m}\frac{2}{\log_2 N}\frac{1}{\left(\rho - \frac{1}{\log_2^2 N}\right)^2}$$

$$= \frac{1}{m}\frac{2\log_2^3 N}{\rho^2 \log_2^4 N - 2\rho\log_2^2 N + 1}$$

$$\le \frac{1}{m} \qquad \text{(for } \rho \ge 1).$$

Thus, with high probability (at least $1 - 1/m \ge 1 - 1/n$), we have that $B_{k^*} < \rho m/\log_2 N$. So, with high probability $U' < k^*$, and thus $U' \in O((m/n)\log\log N)$. □

6.6.4.2 Blocks and Bins Problem

We are now ready to prove Theorem 12 using our results from the balls and urns problem.

Theorem 12. Let $\ell \ge n$ and $N \ge 32$. With high probability for the SBT+ORAM with uniform random block queries, we have $H \in O\left((\ell/n)\log\log N\right)$.

Proof. After s steps by the SBT+ORAM, the ORAM component will have removed $s/(\log_2 N + 1)$ blocks from the longest fetch queues. We consider three cases.

Case 1: $s < 3\ell(\log_2 N + 1)/\log_2 N$. In this case:

$$H \le \frac{s}{n} < \frac{3\ell}{n}\left(1 + \frac{1}{\log_2 N}\right) \in O\left(\frac{\ell}{n}\right) \subseteq O\left(\frac{\ell}{n}\log\log N\right)$$

Case 2: $s \ge 3\ell(\log_2 N + 1)/\log_2 N$, $\ell \le n^2/6$. In this case, we make at least $3\ell/\log_2 N$ removals. We can transform our problem to the balls and urns problem with $m = 3\ell$ balls and n urns, where we remove $3\ell/\log_2 N$ balls from the tallest urns. By Lemma 18, the resulting maximum urn occupancy U' is in $O((\ell/n)\log\log N)$.

By Lemma 13, a given ball is at least as likely to appear in a given urn as the corresponding block is to be found in the corresponding bin. Since we remove the same number of balls from urns as we do blocks from fetch queues, the occupancy of each urn should be at least the length of its corresponding fetch queue, giving $H \leq U' \in O((\ell/n)\log\log N)$ with high probability.

Case 3: $\ell \geq n^2/6$. We know that $H \leq n$, so for $\ell \geq n^2/6$ we have that $H \in O(\ell/n) \subseteq O((\ell/n)\log\log N)$.

□

6.7 Evaluation

We implemented prototypes for SBT and its variants to estimate actual bandwidth costs for various query types. The prototypes simulate secure transfers of blocks between the client and server, tracking each block's location at all times.

6.7.1 Bandwidth Cost Experiments

Figures 6.4–6.12 give our experimental results measuring bandwidth cost for three types of queries and varying three parameters (N, ℓ, λ). Recall that bandwidth cost is given by the total number of block transfers (fetches and stores counted individually), divided by the number of secret accesses (reads or writes) ℓ.

All the experiments used a 64KB block size and allow 8GB of client space, which includes the SBT's block-ID map, space for recently fetched blocks, and space for the ORAM component, if any. Any leftover client space is used as a local block cache. Different block sizes alter storage capacity and client space, but leave bandwidth costs

largely unchanged. During a trial, we run $4N/\ell$ queries of fixed length ℓ, requesting each stored block four times on average.

Each experiment varies one of: block count N (Figures 6.4, 6.7, 6.10), query length ℓ (Figures 6.5, 6.8, 6.11), or milestone count λ (Figures 6.6, 6.9, 6.12). Our default block count $N = 2^{24}$ yields a 1TB TOM storage capacity. Our default query length $\ell = 4\sqrt{N}$ represents a 2^{14} block (1GB) query for the default N. Our default milestone count $\lambda = 8$ leaks at most $I_\lambda = 3$ bits per ℓ-block query.

6.7.1.1 Uniform Random Block Queries (Figures 6.4–6.6)

The *Uniform Random Block* queries are the best suited to the SBT protocols. For each query, we choose ℓ distinct blocks uniformly at random from all N blocks. We used the same type of query to derive our analytic bandwidth cost predictions. All SBT variants outperform ORAM for large uniform random block queries, with costs as low as 5X for the Multi-SBT (Figure 6.5).

6.7.1.2 Uniform/Zipf Fixed Sequence Queries (Figures 6.7–6.12)

For fixed sequence queries, we divide the N blocks into $s = N/\ell$ non-overlapping fixed sequences of ℓ distinct blocks each before permuting the blocks and storing them on the server. Each query consists of exactly one of these fixed sequences, simulating a file system in which each query requests an entire file. *Uniform* fixed sequence experiments choose sequences uniformly at random, while *Zipf* experiments choose sequences from a power law distribution in which the ith most common sequence is chosen with probability H_s/i, where H_s is the sth harmonic number.

There are few (N/ℓ) possible distinct fixed sequence queries, compared to the many (N choose ℓ) uniform random block queries. As a result, fixed sequence queries

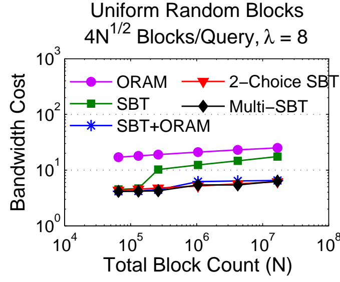

Figure 6.4: Uniform random block queries, varying N

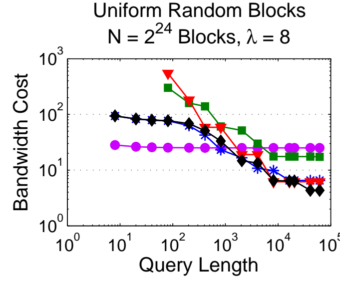

Figure 6.5: Uniform random block queries, varying ℓ

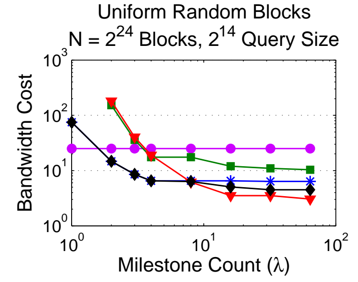

Figure 6.6: Uniform random block queries, varying λ

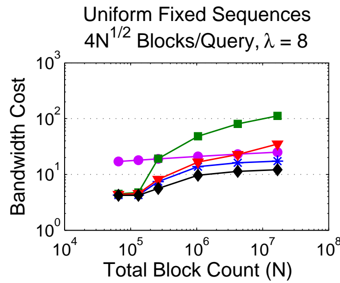

Figure 6.7: Uniform fixed sequence queries, varying N

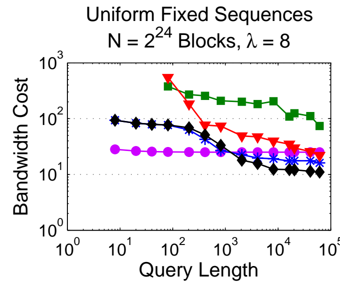

Figure 6.8: Uniform fixed sequence queries, varying ℓ

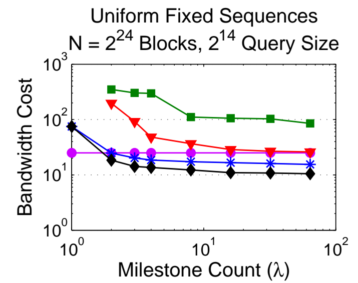

Figure 6.9: Uniform fixed sequence queries, varying λ

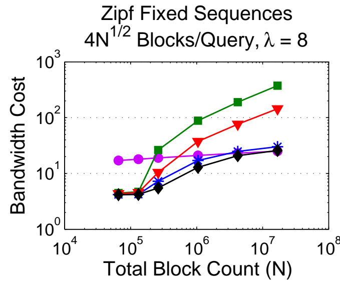

Figure 6.10: Zipf fixed sequence queries, varying N

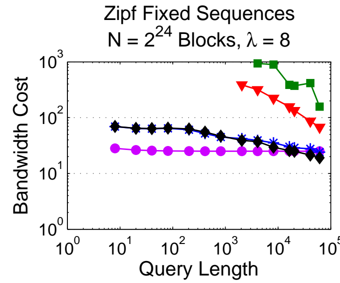

Figure 6.11: Zipf fixed sequence queries, varying ℓ

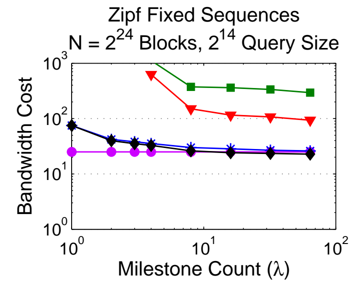

Figure 6.12: Zipf fixed sequence queries, varying λ

are far more likely to repeat, leading to poor SBT performance (Section 6.4.4). Zipf fixed sequence queries repeat frequently, so that ORAM nearly always outperforms the SBT variants (Figures 6.10–6.12). Uniform fixed sequence queries repeated less often, so several variants still outperform ORAM (Figures 6.7–6.9). We reiterate that SBT is a *special-purpose* TOM protocol. The more varied the query block distribution, the better SBT performs.

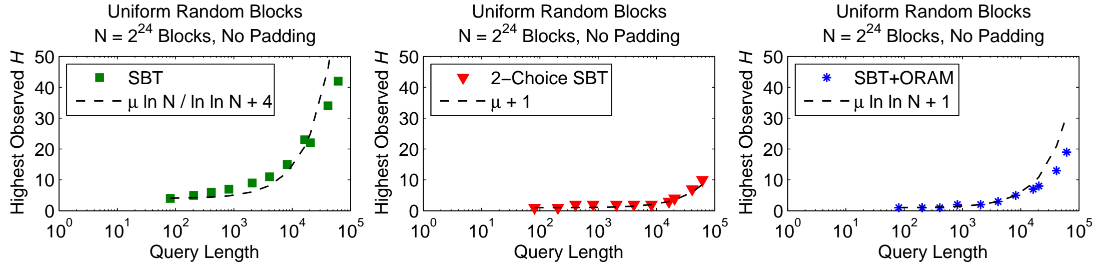

Figure 6.13: Confirmation of analysis. Maximum observed H asymptotically dominated by analytically predicted values. $\mu = \ell/n$

6.7.1.3 Other Observations

For small queries, SBTs with ORAMs converge to a worst-case cost $3\log_2 N$, while others converge to a much larger cost of n (Figures 6.5, 6.8, 6.11). Figures 6.6, 6.9, 6.12 show that we can improve performance by leaking more information (increasing λ) up to $\lambda \approx 32$ ($I_\lambda = 5$ bits). At this point padding costs become negligible, leaving the raw cost of the protocol. Since protocols with ORAM components have smaller worst-case costs, the milestones are packed more tightly, so padding effects become negligible sooner ($\lambda \approx 8$).

6.7.2 Maximum Queue Length Measurements

Theorems 9, 12 and Conjecture 10 give high-probability asymptotic bounds on H for large, uniform random block queries. We validated these bounds by running simulations for the corresponding SBT variants without padding, and measuring the highest observed H over $4N/\ell$ queries for various ℓ. Our results are shown in Figure 6.13 along with plots of concrete functions consistent with our bounds.

6.7.3 Simulator Details

We implemented our simulator in Java, fully modeling SBT behavior. The simulator properly accommodates padding, waiting to release query results until the step count reaches one of the milestones. Our simulator is synchronous, since asynchronous behavior is not needed to measure bandwidth cost.

On a single thread, the simulator requires 0.5 to $1.5\mu s$ per simulated block transfer, depending on the specific protocol and number of blocks. For the sake of speed, we do not manipulate actual block contents. Thus, we're able to efficiently evaluate SBT bandwidth costs for larger block counts and longer runs without the expense of actually performing network transfers, disk IO, and encryption.

We assume a fully de-amortized black-box ORAM with $\log_2 N$ bandwidth cost per step, based on the ORAM in [77]. When simulating the protocols with ORAM components, we step the SBT component $\log_2 N$ times, then step the ORAM once, retrieving the previous step's result.

6.8 Conclusion

We presented a novel ORAM generalization called Tunably-Oblivious Memory (λ-TOM), which permits a privacy/efficiency tradeoff controlled via milestone count λ. We introduced the log-spacing strategy for choosing milestones to minimize padding costs, and strictly bounded the information leaked by each λ-TOM query. We also developed the special-purpose *Staggered-Bin TOM* protocol, and several read-only variants, including the Multi-SBT. We showed analytically and empirically that the Multi-SBT is highly efficient for large queries that are not cache-friendly, achieving bandwidth costs as low as 6X compared to the 22X-29X costs of the best existing ORAM protocols, while

leaking at most 3 bits per query. We believe that the TOM model can be used in future work to build other highly secure special-purpose protocols, like SBT, that outperform current ORAM techniques on a variety of workloads.

Chapter 7

Conclusion

In Chapters 2 and 3 we presented two novel attacks that motivate data outsourcing protocols that protect access pattern privacy. We then introduced two new practical and efficient Oblivious RAM (ORAM) protocols, Burst ORAM and OS+PIR, in Chapters 4 and 5, respectively.

Burst ORAM allows large bursts of requests to be satisfied efficiently using minimal bandwidth. We introduced the XOR technique to reduce the number of online block transfers, and proposed a novel architecture that prioritizes online transfers, delaying shuffling until idle periods. We evaluated Burst ORAM on a real-world network application workload and showed that it achieved near-optimal response times that were orders of magnitude lower than those of existing ORAM protocols.

OS+PIR reduces total bandwidth costs by using PIR to enable efficient level configurations in ORAM partitions. We also introduce several enhancements that allow OS+PIR to achieve bandwidth costs as low as 11X–13X, outperforming the best existing protocols by a factor of 2. OS+PIR is particularly appealing as a practical ORAM solution for mobile devices, where bandwidth costs dominate.

In Chapter 6, we explored other mechanisms for reducing bandwidth costs, including exchanging a bounded amount of information leakage for improved efficiency, and employing special-purpose and read-only protocols. We proposed the Tunably-Oblivious Memory (TOM) privacy model, as well as several concrete TOM instantiations including SBT and the read-only Multi-SBT. We showed that the Multi-SBT is highly efficient for large queries that are not cache-friendly, and can achieve bandwidth costs as low as 6X while leaking at most 3 bits per query.

The advances in Oblivious RAM protocols made during the last few years have transformed it from a largely theoretic notion into a viable solution to the private data outsourcing problem. While ORAM users can still expect to pay several times the cost of an unprotected protocol, the costs continue to be driven down by developments such as those presented here, making ORAM feasible for a wider range of applications. Given the rapid pace of ORAM research, it would not be surprising to see ORAM primitives packaged into secure data outsourcing services in the near future.

Bibliography

[1] Salaries of federal employees located in the District of Columbia. Available: http://php.app.com/fed_employees11/search.php, 2011. Source: U.S. Office of Personnel Management.

[2] County of riverside class and salary listing. http://www.rc-hr.com/HRDivisions/Classification/tabid/200/ItemId/2628/Default.aspx, February 2012.

[3] Amazon web services. http://aws.amazon.com, June 2014.

[4] Google cloud platform. http://cloud.google.com, June 2014.

[5] Microsoft azure. http://azure.microsoft.com, June 2014.

[6] D. Agrawal, A. Abbadi, F. Emekci, A. Metwally, and S. Wang. Secure data management service on cloud computing infrastructures. *New Frontiers in Information and Software as Services*, pages 57–80, 2011.

[7] D. Agrawal, A. El Abbadi, F. Emekci, and A. Metwally. Database management as a service: Challenges and opportunities. In *Proc. ICDE*, pages 1709–1716, 2009.

[8] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proc. ACM SIGMOD*, pages 563–574, 2004.

[9] Christoph Bandt and Bernd Pompe. Permutation entropy: A natural complexity measure for time series. *Phys. Rev. Lett.*, 88:174102, Apr 2002.

[10] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In *Proc. EUROCRYPT*, pages 224–241, 2009.

[11] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.

[12] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *Proc. TCC*, pages 535–554, 2007.

[13] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. System Sci.*, 13(3):335–379, 1976.

[14] B. Buchberger and F. Winkler. *Gröbner bases and applications.* Cambridge Univ Pr, 1998.

[15] Julie Anne Cain, Peter Sanders, and Nick Wormald. The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 469–476. Society for Industrial and Applied Mathematics, 2007.

[16] Alberto Ceselli, Ernesto Damiani, Sabrina De Capitani di Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Modeling and assessing inference exposure in encrypted databases. *ACM Trans. Inf. Syst. Secur.*, 8(1):119–152, 2005.

[17] K. Chen, R. Kavuluru, and S. Guo. RASP: efficient multidimensional range query on attack-resilient encrypted databases. In *Proc. ACM CODASPY*, pages 249–260, 2011.

[18] Yanpei Chen, Kiran Srinivasan, Garth Goodson, and Randy Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 43–56, New York, NY, USA, 2011. ACM.

[19] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[20] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proc. ACM CCSW*, pages 85–90, 2009.

[21] Kai-Min Chung, Zhenmin Liu, and Rafael Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. `http://arxiv.org/abs/1307.3699`, 2013.

[22] Kai-Min Chung and Rafael Pass. A simple oram. `https://eprint.iacr.org/2013/243.pdf`, 2013.

[23] V. Ciriani, S. De Capitani Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. *Proc. ESORICS*, pages 440–455, 2009.

[24] Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In *EUROCRYPT*, pages 178–189. Springer, 1996.

[25] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.

[26] Ernesto Damiani, Sabrina De Capitani di Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proc. ACM CCS*, pages 93–102, 2003.

[27] J. Dautrich and C. Ravishankar. Security limitations of using secret sharing for data outsourcing. In *Proc. DBSec*, 2012.

[28] J. Dautrich and C. Ravishankar. Compromising privacy in precise query protocols. In *Proc. EDBT*, 2013.

[29] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM response times for bursty access patterns. In *USENIX Security*, 2014.

[30] Sabrina De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Efficient and private access to outsourced data. In *Proc. ICDCS*, 2011.

[31] F. Emekci, D. Agrawal, A.E. Abbadi, and A. Gulbeden. Privacy preserving query processing using third parties. In *Proc. ICDE*, pages 27–27. IEEE, 2006.

[32] B. Faaland. Solution of the value-independent knapsack problem by partitioning. *Operations Research*, 21(1):332–337, 1973.

[33] X.G. Fang and G. Havas. On the worst-case complexity of integer gaussian elimination. In *Proceedings of the 1997 international symposium on symbolic and algebraic computation*, pages 28–31. ACM, 1997.

[34] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Secure processor architecture for encrypted computation on untrusted programs. In *Proc. ACM CCS Workshop on Scalable Trusted Computing*, pages 3–8, 2012.

[35] Craig Gentry, Kenny Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *PETS*, 2013.

[36] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *Automata, Languages and Programming*, pages 803–815. Springer, 2005.

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

[38] M. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. *Automata, Languages and Programming*, pages 576–587, 2011.

[39] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 157–167. SIAM, 2012.

[40] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. ACM SIGMOD*, pages 216–227, 2002.

[41] M.A. Hadavi and R. Jalili. Secure data outsourcing based on threshold secret sharing; towards a more practical solution. In *Proc. VLDB PhD Workshop*, pages 54–59, 2010.

[42] Torben Hagerup and Christine Rüb. A guided tour of chernoff bounds. *Information processing letters*, 33(6):305–308, 1990.

[43] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multidimensional range queries over outsourced data. *The VLDB Journal*, pages 1–26, 2011.

[44] M Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[45] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Inference attack against encrypted range queries on outsourced databases. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 235–246. ACM, 2014.

[46] Norman Lloyd Johnson and Samuel Kotz. *Urn models and their application: an approach to modern discrete probability theory.* Wiley New York, 1977.

[47] M. Kantarcıoglu and C. Clifton. Security issues in querying encrypted data. In *Proc. DBSec*, pages 325–337, 2005.

[48] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Proc. EUROCRYPT*, pages 146–162, 2008.

[49] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.

[50] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 213–226, Berkeley, CA, USA, 2008. USENIX Association.

[51] Jun Li and Edward Omiecinski. Efficiency and security trade-off in supporting range queries on encrypted databases. In *Proc. DBSec*, pages 69–83, 2005.

[52] M. Li, S. Yu, N. Cao, and W. Lou. Authorized private keyword search over encrypted personal health records in cloud computing. In *Proc. ICDCS*, 2011.

[53] P. Lin and K.S. Candan. Hiding tree structured data and queries from untrusted data stores. *Information Systems Security*, 14(4):10, 2005.

[54] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. *FAST*, pages 199–213, 2013.

[55] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical oblivious computation in a secure processor. In *ACM CCS*, 2013.

[56] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM TKDD*, 1(1):3–es, 2007.

[57] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Pirmap: Efficient private information retrieval for mapreduce. In *Financial Cryptography and Data Security*, pages 371–385. Springer, 2013.

[58] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Efficient private file retrieval by combining ORAM and PIR. In *NDSS*, 2014.

[59] Michael Mitzenmacher. Some open questions related to cuckoo hashing. In *Algorithms-ESA 2009*, pages 1–10. Springer, 2009.

[60] Einar Mykletun and Gene Tsudik. Aggregation queries in the database-as-a-service model. In *Proc. DBSec*, pages 89–103, 2006.

[61] Yuto Nakano, Carlos Cid, Shinsaku Kiyomoto, and Yutaka Miyake. Memory access pattern protection for resource-constrained devices. In *Smart Card Research and Advanced Applications*, pages 188–202. Springer, 2013.

[62] A. Nergiz and C. Clifton. Query processing in private data outsourcing using anonymization. In *Proc. DBSec*, pages 138–153, 2011.

[63] F. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. In *Proc. FC*, 2011.

[64] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[65] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[66] Martin Raab and Angelika Steger. Balls into bins – a simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[67] M.O. Rabin. Probabilistic algorithm for testing primality. *Journal of number theory*, 12(1):128–138, 1980.

[68] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*. 2013.

[69] Pierangela Samarati and Sabrina De Capitani di Vimercati. Data protection in outsourcing scenarios: issues and directions. In *Proc. ASIACCS*, pages 1–14, 2010.

[70] A. Shamir. How to share a secret. *Communications of the ACM*, pages 612–613, 1979.

[71] Emily Shen, Elaine Shi, and Brent Waters. Predicate privacy in encryption systems. In *Proc. TCC*, pages 457–473, 2009.

[72] E. Shi, J. Bethencourt, T.-H.H. Chan, Dawn Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *Proc. IEEE S&P*, pages 350–364, 2007.

[73] E. Shi, H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT*, 2011.

[74] Radu Sion. On the computational practicality of private information retrieval. In *Proc. NDSS*, 2007.

[75] Emil Stefanov and Elaine Shi. Multi-Cloud Oblivious Storage. In *CCS*, 2013.

[76] Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy*, 2013.

[77] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. NDSS, 2012.

[78] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *ACM CCS*, 2013.

[79] J. Stein. Computational problems associated with racah algebra. *Journal of Computational Physics*, 1(3):397–405, 1967.

[80] X.X. Tian, C.F. Sha, X.L. Wang, and A.Y. Zhou. Privacy preserving query processing on secret share based data storage. In *Proc. DASFAA*, pages 108–122. Springer, 2011.

[81] Jonathan Trostle and Andy Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *Information Security*, pages 114–128. Springer, 2011.

[82] Jieping Wang and Xiaoyong Du. A secure multi-dimensional partition based index in DAS. In *Proc. APWeb*, pages 319–330, 2008.

[83] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proc. ACM CCS*, pages 139–148, 2008.

[84] Peter Williams and Radu Sion. Usable PIR. In *NDSS*, 2008.

[85] Peter Williams and Radu Sion. Round-optimal access privacy on outsourced storage. In *CCS*, 2012.

[86] Peter Williams and Radu Sion. Sr-oram: Single round-trip oblivious ram. *ACNS, industrial track*, pages 19–33, 2012.

[87] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A parallel oblivious file system. In *CCS*, 2012.

[88] Zhiqiang Yang, Sheng Zhong, and Rebecca N. Wright. Privacy-preserving queries on encrypted data. In *Proc. ESORICS*, pages 479–495, 2006.

[89] Xiangyao Yu, Christopher W Fletcher, Ling Ren, Marten van Dijk, and Srinivas Devadas. Generalized external interaction with tamper-resistant hardware with bounded information leakage. In *Proc. ACM CCSW*, pages 23–34. ACM, 2013.