THE UNIVERSITY OF CHICAGO

TOWARDS BETTER DATA PRIVACY AND UTILITY IN THE UNTRUSTED CLOUD

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
MIN XU

CHICAGO, ILLINOIS
DECEMBER 2020

Copyright © 2020 by Min Xu

All Rights Reserved

To my parents, my wife and my daughter

“Arguing that you don’t care about the right to privacy because you have nothing to hide is no different than saying you don’t care about free speech because you have nothing to say.”

— Edward Snowden, Permanent Record

Table of Contents

LIST OF FIGURES . . . viii

ACKNOWLEDGMENTS . . . xi

ABSTRACT . . . xiii

1 INTRODUCTION . . . 1

2 BACKGROUND AND RELATED WORK . . . 6
- 2.1 Privacy-preserving Cloud Analytics . . . 6
  - 2.1.1 Analytical Queries . . . 6
  - 2.1.2 Federated SQL Processing . . . 7
  - 2.1.3 Data Collection and Analytics . . . 14
- 2.2 End-to-End Encrypted Search . . . 17
  - 2.2.1 Information Retrieval . . . 17
  - 2.2.2 Searchable Symmetric Encryption (SSE) . . . 19

3 HERMETIC: PRIVACY-PRESERVING DISTRIBUTED ANALYTICS WITHOUT (MOST) SIDE CHANNELS . . . 24
- 3.1 Introduction . . . 24
- 3.2 The Hermetic System . . . 27
  - 3.2.1 Overview . . . 27
  - 3.2.2 Initialization . . . 28
  - 3.2.3 Attestation . . . 28
  - 3.2.4 Query submission and verification . . . 29
  - 3.2.5 Query execution . . . 30
- 3.3 Oblivious Execution Environments . . . 30
  - 3.3.1 OEE properties . . . 30
  - 3.3.2 Challenges in building an OEE today . . . 31
  - 3.3.3 The Hermetic OEE . . . 32
- 3.4 Oblivious operators . . . 33
  - 3.4.1 Extending oblivious operators . . . 35
  - 3.4.2 Differentially-private padding . . . 36
- 3.5 Privacy-aware Query Planning . . . 37
- 3.6 Implementation . . . 39
  - 3.6.1 Hermetic hypervisor . . . 39
  - 3.6.2 Oblivious execution environments . . . 40
  - 3.6.3 Trusted computing base . . . 41
- 3.7 Evaluation . . . 41
  - 3.7.1 Experimental setup . . . 42
  - 3.7.2 OEE security properties . . . 43
  - 3.7.3 Performance of relational operators . . . 43

3.7.4 End-to-end performance . . . 45
3.7.5 Comparison with the state-of-the-art . . . 46
3.7.6 Trading-off privacy and performance . . . 47

4 COLLECTING AND ANALYZING DATA OVER MULTIPLE SERVICES WITH LOCAL DIFFERENTIAL PRIVACY . . . 49
4.1 Introduction . . . 49
4.2 User-level Privacy across Multiple Services . . . 55
4.3 Attribute Aggregation . . . 57
4.3.1 Building Block: Frequency Oracles . . . 57
4.3.2 Sensitive-weight Frequency Oracles . . . 59
4.3.3 Partition-Rounding-Perturb Framework . . . 61
4.4 1-to-1 Joint Frequency Oracles . . . 64
4.4.1 Split-and-Conjunction Baseline . . . 64
4.4.2 Multi-Service Joint Frequency Oracles . . . 66
4.5 Handling One-to-Many Join . . . 68
4.5.1 Frequency-based Attack . . . 68
4.5.2 Hiding Existence and Frequency with τ-Truncation . . . 69
4.5.3 Double Rounding: Recovering Aggregation from Truncated Tuples . . . 71
4.6 Range Consistency Optimization . . . 73
4.6.1 Optimal Range Decomposition . . . 74
4.6.2 Consistency Optimization . . . 75
4.7 Evaluation . . . 76
4.7.1 Attribute Aggregation . . . 78
4.7.2 Range Consistency Optimization . . . 81
4.7.3 Joint Aggregation . . . 83
4.7.4 Handling Group-by Queries . . . 84

5 SEARCHING ENCRYPTED DATA WITH SIZE-LOCKED INDEXES . . . 88
5.1 Introduction . . . 88
5.2 Problem Setting . . . 91
5.3 Insecurity of Encrypting Standard Indexes . . . 92
5.4 Basic Encrypted Size-Locked Indexes . . . 96
5.4.1 A Size-Locked Index . . . 97
5.4.2 Security Analysis . . . 101
5.5 Partitioning Size-Locked Indexes . . . 103
5.5.1 Vertical Partitioning . . . 104
5.5.2 Horizontal Partitioning . . . 110
5.6 Secure Binary Index Lookup . . . 111
5.6.1 Secure Binary Index Lookup without Partitions . . . 111
5.6.2 Secure Binary Index Lookup with Partitions . . . 112
5.7 Evaluation . . . 113

6 CONCLUSION AND FUTURE WORK 121
6.1 Conclusion 121
6.2 Future Work 122
6.2.1 Shuffled LDP 122
6.2.2 Join with Star Schema 122
6.2.3 Adaptive Index Management System 123
6.2.4 Multi-user End-to-end Encrypted Search 124

REFERENCES 125

APPENDIX 138

A HERMETIC IMPLEMENTATION DETAILS 138
A.1 Query operators 138
A.1.1 List of query operators 138
A.2 Predictable timing for OEE operators 142
A.2.1 Avoiding data-dependent control flow and instructions 142
A.2.2 Making L1 cache misses predictable 143
A.2.3 Determining a conservative upper bound on execution time 145
A.2.4 Overhead of time padding 146
A.3 Oblivious primitives which use dummy tuples 146
A.3.1 Supporting dummy tuples 146
A.3.2 Adding dummy tuples to the primitive outputs 148
A.4 Hermetic multi-objective query optimization 148

B SIZE-LOCKED INDEXING SECURITY AND CONSTRUCTION DETAILS 155
B.1 Formal Cryptographic Analysis 155
B.2 CTR-DSSE Construction 158
B.3 Experimental Queries 160
B.4 Extended Security Analysis 161
B.4.1 Leakage Across Various Settings 162
B.4.2 Cap Undershooting 163
B.4.3 Timing Attack on the Lookup Table 164
B.4.4 Ranking Deflation Attack 165

List of Figures

2.1 Example scenario. Analyst Alice queries sensitive data that is distributed across multiple machines, which are potentially owned by multiple participants. An adversary has complete control over some of the nodes, except the CPU. 7
2.2 Games used in Definition 3. 21

3.1 OEE MergeSort with data oblivious instruction trace. `end`, `ascend`, `attrs` are function parameters pre-loaded into global `oee_buffer`, and `tb` points into the `oee_buffer` for storing tuple input and output. `OEE_SIZE` indicates the maximum number of tuples that one OEE invocation could process. 34
3.2 OEE isolation. 35
3.3 The OEE-assisted HybridSort primitive. 36
3.4 Query planning for query SELECT COUNT$(*)$ FROM C, T, P WHERE C.cid $= T$.cid AND T.location $= P$.location AND C.age ≤ 27 AND P.category = "hospital". 38
3.5 Cost model for Hermetic's relational operators. n: first input size; c: OEE capacity; m: second input size; k: output size. 39
3.6 Experimental configurations and their resilience to different channels (`MS`: MergeSort, `BS`: BatcherSort, `HS`: HybridSort, `CP`: cartesian product, `SMJ`: sort-merge join). 41
3.7 Schema and statistics of the relations. Synthetic was generated with a variety of tuples and multiplicities. 41
3.8 Performance of SELECT (S_1), GROUP BY (S_2) and JOIN (S_3) for different data sizes and join multiplicities. 44
3.9 Performance of all configurations for Q_4 – Q_6. 45
3.10 Comparison with Opaque. 46
3.11 Total privacy and performance costs at various points in the trade-off space. 47

4.1 Data collection and analytics across multiple services (top), and an algorithmic pipeline of our solution (bottom). 51
4.2 Frequency-based attack (# Txn = number of transactions). 52
4.3 Example of running AHIO on tuples with attributes Age $\in [1, 125]$ and Inc $\in [1, 125]$ (income), and aggregation on Inc for users with Age $\in [10, 20]$. ① partition the tuples by attribute, round their Inc or Age value, and augment with attribute X_{Inc} or X_{Age} for the rounding value; ② rewrite the query into two frequency aggregations for the min and max of Inc; ③ combine the frequency estimations, weighted by the min and max of Inc, and scale the result by 2 to compensate for the attribute partitioning. 63
4.4 HIO-JOIN across two services, each collecting one attribute using hierarchy with fan-out $B = 4$. 65
4.5 Optimal range decompositions for predicate C on attributes Age and Income, with fan-out of 5. 74
4.6 Aggregation on sensitive attribute using SYN-1. 79
4.7 Q1-Q3 on PUMS-P. 80
4.8 Effect of range consistency optimization using SYN-1. Left: aggregation on sensitive attribute. Right: aggregation on non-sensitive attribute. k indicates the number of top decompositions and d_q indicates the number attributes in the range predicate. 82

4.9 Joint aggregation over SYN-2. Left: one-to-one, C, O, CO indicate one categorical, one ordinal, one categorical and one ordinal attributes in the predicate, respectively. Right: one-to-many, τ is the truncation number. 83
4.10 Joint aggregation over PUMS-H. Left: Indiana. Right: Illinois. The predicate is "CITY $= x$ AND AGE $\in [l, r]$". 85
4.11 HIO-GROUP-BY: for group-by queries 86
4.12 One run of HIO-GROUP-BY estimation versus the ground truth. 86
4.13 Average errors (over 15 runs) of HIO-GROUP-By by group. 86

5.1 Search features in popular storage services. Services support either just conjunctions ($\wedge$) or additionally disjunctions ($\vee$) over keywords. The top-k results are ranked according to relevance (rel) or just date. Previews may of search results may include name (n), modification date (d), file size (s), and/or the parent directory (p). Search indices may be updated due to edits within a file (✓) or only when documents are added or deleted (✗). 92
5.2 Document-injection attack on Lucene to recover indexed term. The term s^* results in a noticeably smaller change in byte-length than other terms. 94
5.3 Four stages of basic encrypted size-locked indexing. ①: documents to be indexed; ②: primary index on the documents; ③: document updates and their encoding; ④: primary index merged with the encoded updates, with new entries shown in red. 97
5.4 Basic encrypted size-locked index construction. EDB and $\vec{C}_{up}$ denote the encrypted index and the outstanding encrypted updates. The semicolon in protocol input separates the input to client (left) and to server (right). 101
5.5 Comparing $\mathsf{Cap}(N)$ to N_{10}, the total number of postings in the top-10 of postings-lists across three datasets and for varying . We added documents in random order, measuring the ratio N_{10}/N after each addition. Heap's Law is apparent in the shape of each curve, and our choice of Cap proves conservative. 106
5.6 Vertically partitioned encrypted size-locked index construction. B_l and $\vec{C}_l$ denote the encrypted vertical partition and outstanding updates at level l. 109
5.7 The datasets used for our experiments. The first column is size of the datasets in MB, followed by: the number of documents $|D|$, the vocabulary size $|W|$, the number of unique word/document pairs $|DB|$, and the size in MB of a standard Lucene index for the corpus. 114
5.8 Search bandwidth cost (in MB) after adding all documents in corpus. L1/2 stands for level 1 and 2 in vertical partitioning. 114
5.9 End-to-end search time. The rows LOW, MED and HIGH indicate downstream network bandwidth at levels of 12-25Mbps, 80-160Mpbs and 700Mbps, respectively. PG-1 and PG-2 indicate performance of searching for the first and second pages. The error bar indicates the standard deviation. 115
5.10 End-to-end search time for the first result page, with various percentage of the Enron dataset added to the index. LOW and HIGH indicate downstream network bandwidth at levels of 12-25Mbps and 700Mbps, respectively. 117
5.11 Processing time to lazily merge as a function of the fraction of the document corpus inserted before performing the merge. 118

5.12 End-to-end search time, right after adding 10 documents, when various percentages of the document corpus have been added and merged. 119

A.1 The oblivious `filter` operator 141
A.2 Every x86 instruction used in Hermetic OEE 142
A.3 The `merge-sort` supported in OEE. 152
A.4 Cycle-resolution measurements of the actual timing of MergeSort (`MS`) and LinearMerge (`LM`) inside the OEE, and their padded timing, respectively. 153
A.5 Memory access patterns of OEEMergeSort on sorted, reverse sorted and random input. 153
A.6 L1 hit and miss latencies for MergeSort, as reported by Intel's specifications (l_{L1}, l_{L3}), and as measured on different datasets (l^*_{L1}, l^*_{L3}). The last columns show the values we used in our model. All values are in cycles. 153
A.7 Latency of MergeSort (`MS`), LinearMerge (`LM`) with increasing un-observable memory size, compared to the BatcherSort (`BS`). `-PD` indicates time padding. 154

B.1 CTR-DSSE construction. 160

ACKNOWLEDGMENTS

My PhD study can be literally divided into two phases:

In phase I (year 1-3), I was fortunate to work with Prof. Ariel Feldman on system security. Before getting into the PhD program, I had more experience working on storage systems, and planned to continue the direction for the PhD. Thanks to the supportive atmosphere of the department, I talked to faculties working on different areas in my first year to decide on my PhD research topic. Ariel's project on addressing urging vulnerabilities of the real-world systems attracted me, and I decided to work with him on topic of system security. I am grateful for what I have learned during the meetings and discussions with the group: Galen Harrison and Minhaj Us Salam Khan, as well as my project collaborators: Andreas Haeberlen and Antonis Papadimitriou from UPenn.

In phase II (year 3-5), I transferred to the crypto group, lead by Prof. David Cash, and started working on the intersection of system and crypto. First, I am extremely grateful to David for accepting me as his student to continue my PhD study. Furthermore, through our collaboration, David not only tried his best to direct the project to fit my background and interest, *i.e.*, system design, but also encouraged me to explore opportunities from the theorectical perspective, which I found both interesting and rewarding. All of these means a lot to me. In addition, I want to thank the group members: Akshima, Francesca Falzon and Alex Hoover, for the discussion and chatting, and my project collaborators: Thomas Ristenpart and Armin Namavari from Cornell.

During my PhD study, I had three great internship experiences: one with Microsoft Research at Redmond in 2017, mentored by Arvind Arasu; one with Alibaba DAMO Academy at Bellevue in 2019, mentored by Bolin Ding; and one with Facebook Boston in 2019, mentored by Srikanth Sastry and Lucas Waye. I learned and enjoyed a lot during the three internships, and the project with Alibaba also lead to a full paper at VLDB 2020, as well as a chapter of my dissertation.

I also want to thank the staff of the department: Nita Yack, Sandra Quarles, Margaret Jaffey, Patricia Baclawski, Bob Barlett, Phil Kauffman, Colin Hudler and Tom Dobes, for assistance and tech support throughout my PhD study.

Finally, I want to thank my wife, Zixiaojie Yang, for understanding and supporting me everytime I encountered obstacles. I can never survive these challenges without her. I also want to thank my parents for supporting me as they always do during the 5 years. And I thank my daughter, Allison Xu, whose birth cheered me up and encouraged me to endeavor to conquer whatever in front of me.

ABSTRACT

Users data are stored and utilized in the cloud for various purposes. How to best utilize these data while at the same time preserving the privacy of their owners is a challenging problem. In this dissertation, we focus on three important cloud applications, and propose solutions to enhance the privacy-utility tradeoffs of the existing ones.

The first application is the *federated SQL processing*, where multiple mutually-untrusted data owners hold valuable data of their own, and want to execute joint SQL queries on these data without leaking information about individual records in their own shares. The second one is the *cloud data collection and analysis*, where services collect their users data, with proper privacy guarantees, and want to enable expressive and accurate analysis on the collected data. The last one is the *end-to-end encrypted data retrieval*, where a single data owner outsources her end-to-end encrypted data to the cloud, and, later, wants to retrieve some of them that are most relevant to the keyword queries requests.

After comprehensive literature review of the existing solutions, we realize that the privacy-utility tradeoffs of state of the art can be substantially improved. For federated SQL processing, existing solutions leverage trusted hardware for efficient and secure computations in the cloud, while subsequent work demonstrate the devastating *side-channel* vulnerability of these solutions. We mitigate such vulnerability to improve the existing solutions. For data collection and analysis, existing solutions do not support joint analysis across data collected by separate services, and the supported analytics is limited, *i.e.*, counting frequency of certain value. We propose new mechanisms and estimation algorithms to achieve better utility on the collected data. For end-to-end encrypted data retrieval, existing solutions are vulnerable to the powerful yet practical *file-injection* attacks, and we propose new constructions that can defend against such attacks, with practical performance.

We thoroughly analyze the privacy and utility of the proposed solutions, when necessary. We also implement prototypes for all the solutions, and conduct extensive evaluations to show the

performance of our proposed solutions.

CHAPTER 1

INTRODUCTION

Cloud users have their data uploaded, intentionally or not, to the cloud everyday when they enjoy the various services backed in the cloud. For instance, when you go shopping on Amazon, every single action that you take, including searching, browsing and purchasing, is tracked by the app or webpage, and the tracked information gets pushed to the cloud server to drive the model there so that it can accurately predict your next shopping needs. In this case, hundreds of millions of users, like you, have their personal data, transparently, collected to and processed in the cloud, and the goal of such data collection and processing is to improve the user experience. On the other hand, users of cloud storage services, such as Dropbox, choose to upload their document data to the cloud server so that they can access their data anywhere reliably. In this case, uploading personal data to the cloud is an inherent part of the service.

Whether to improve or provide the services, collecting and processing users' data in the cloud turns out necessary to drive today's cloud business model. This does not mean that the service providers should collect whatever information they can from their users, and store and compute on the collected users' data without care. Such toxic privacy practice would doom the services, instead of helping them, because not only the privacy-related government agencies, such as the Federal Trade Commission, would seriously punish it, but the users themselves , with more awareness of the risk of data breaches, would also choose to avoid these services. Hence, once users agree with the service's privacy policy, and grant their data permissions, the service providers should only collect the granted data to drive their data model, and try their best to protect data collected from each individual user, either at rest or during computations.

Providing services based on users' data, while at the same time protecting the data from powerful adversaries, turns out very challenging. In practice, the powerful adversaries could observe, or even manipulate with, the the executions on the users' data in the cloud, and examples of such adversaries could obtain the capabilities by exploiting the vulnerabilities in the cloud infrastruc-

ture, or simply via insider threat of the service. Such adversaries can easily break the privacy if data collected by the services are stored or computed in plaintext. Even with stronger protection that obfuscates the data throughout the lifetime in the cloud, the adversaries might be able to glean information about the data from the execution *side channels*, such as the time, the input/output sizes and how the execution accesses data. To fully prevent the adversaries from learning anything about the users, the users should upload nothing that is computationally dependent on their personal information, *e.g.*, not uploading anything, or via semantically-secure encryption using secret keys only known to the users themselves. This basically makes the collected data useless, which might not be acceptable to certain services, and leads to service termination to the users.

In this thesis, we focus on the problem of how to enable useful computations on users data in the cloud, while guaranteeing strong privacy to each individual user. And we focus on three important cloud applications.

Federated SQL processing. The first application is the federated SQL processing. The motivation for such application is that separate parties, each with their own tables of users' data, need to assemble their data in the same cloud, and answer SQL queries on these tables. For instance, a COVID-19 testing center, with records of tested cases, a flight company, with records of passengers flight information, and a census bureau, with records of citizens' demographic information, want to answer the join query: *how many people living in Chicago and tested positive for COVID-19 flew from China to US in January 2020?*. The privacy requirement for this application is that the adveraries should not learn anything about individual record in each table. And the utility objective is to answer the SQL queries accurately and efficiently.

It is possible to achieve privacy-preserving federated SQL processing using cryptographic techniques [124, 128], but the resulting systems tend to have a high overhead and can only perform a very limited set of operations. An alternative approach [122, 139, 178] is to rely on *trusted execution environments (TEEs)*, such as Intel's SGX. With this approach, the data remains encrypted even in memory and is only accessible within a trusted *enclave* within the CPU. As long as the

CPU itself is not compromised, this approach can offer strong protections, even if the adversary has compromised the operating system on the machines that hold the data.

However, even though trusted hardware can prevent an adversary from observing the data itself, the adversary can still hope to learn facts *about* the data via various side channel attacks [27, 95, 121, 169]. Existing work [11, 12, 121, 122, 178] on mitigating side channel vulnerabilities suffer from two critical limitations: (1) none of them addresses the various powerful side channel attacks simultaneously, and, therefore, the performance cost of doing to is unclear; and (2) for certain side channel, *e.g.*, execution time, existing solution introduces prohibitive performance overhead. In Chapter 3, we address all of these limitations with the proposed system Hermetic [166, 167]. Hermetic mitigates the four major digital side channels – timing, memory access patterns, instruction traces, and output sizes – a challenging design problem in itself, while achieving performance that is as good or better than systems with weaker privacy guarantees.

Cloud data collection and analysis. The second application is motivated by the needs that sensitive data, or *attributes*, about users' profiles and activities be collected by enterprises and exchanged between different services in one organization to help make informed data-driven decisions. For instance, Google deploys collector on Chrome web browser to collect real-time activities of their users and their client-side software, and statistics, as simple as how frequently certain software features are used, derived from the collected data is a key part of an effective, reliable operation of online services by Cloud service and software platform operators.

In the absence of the central trusted party, directly collecting users' sensitive data to the cloud will likely be detrimental to the end-users' privacy. Thus, the *local differential privacy* model (LDP) [47] is adopted. Each user has her attribute values locally perturbed by a randomized algorithm with LDP guaranteed, *i.e.*, the likelihood of any specific output of the algorithm varies little with input; each user can then have the perturbed values leave her device, without the need to trust the data collector. Analytical queries can be answered approximately upon a collection of LDP perturbed values. Apple [5], Google [54], and Microsoft [43] deploy LDP solutions in their

applications on users' device to estimate statistics about user data, *e.g.*, histograms and means.

In Chapter 4, we investigate how to collect and analyze multi-dimensional data under LDP in a more general setting: each user's attributes are collected by multiple independent services, with LDP guaranteed; and data tuples from the same user collected across different services can be joined on, *e.g.*, user id or device id which is typically known to the service providers. Two natural but open questions are: *what privacy guarantee can be provided on the joined tuples for each user*, and *what analytics can be done on the jointly collected (and independently perturbed) multi-dimensional data*. Our main contribution [165, 168] is to extend the setting and query class supported by existing LDP mechanisms from single-service frequency queries to the more complex ones, including aggregations on sensitive attributes and multi-service joint aggregation; no existing LDP mechanisms can handle our target setting and query class with formal privacy and utility guarantees.

End-to-end encrypted data retrieval. The third application is the end-to-end encrypted search for cloud storage services. Having users directly uploading their personal data in plaintext to their services, like Dropbox, Google Drive and iCloud do, exposes users sensitive data to the powerful adversaries inside and outside the services. On the other hand, end-to-end encryption protects data stored at these services, but deploying it poses both usability and security challenges. Off-the-shelf file encryption disables server-side data processing, including features for efficiently navigating data at the request of the client. And even with well-designed special-purpose encryption, some aspects of the stored data and user behavior will go unprotected.

The problem of implementing practical text search for encrypted data was first treated by Song, Wagner, and Perrig [142], who described several approaches to solutions. Subsequently a primitive known as *dynamic searchable symmetric encryption (DSSE)* was developed over the course of an expansive literature (c.f., [23–25, 30, 32, 40–42, 56, 86, 88, 94, 110, 143, 171]). But DSSE doesn't provide features matching typical plaintext search systems, and more fundamentally, all existing approaches are vulnerable to attacks [22,29,61,64,79,116,130,152,162,176] that recover plaintext

information from encrypted data, by abusing their specified leakages.

In Chapter 5, we initiate a detailed investigation of the simple, folklore approach to end-to-end encrypted search: simply encrypt a standard search index, storing it remotely and fetching it to perform searches. We first introduce what we call *size-locked indexes*. These are specialized indexes whose representation as a bit string has length that is a fixed function of information we allow to leak, *i.e.*, the total number of documents indexed and the total number of postings handled. By coupling our size-locked index with standard authenticated encryption, we are able to build an encrypted index system that works with stateless clients and provides better search functionality (full BM25-ranked search) than prior approaches, while resisting both leakage abuse and injection attacks. Furthermore, we propose two orthogonal approaches to partitioning our size-locked indexes to reduce the end-to-end search cost. We formally analyze the security of our constructions, and evaluate their performance using real-world datasets and settings.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Privacy-preserving Cloud Analytics

Privacy-preserving analytics denotes a wide variety of computations on data with enhanced privacy guarantees. In particular, other than the public information on the computations, *e.g.*, circuits and parameters, as well as the computation output, no other information about each individual input data should be learned by those who execute the computations and obtain the output.

In this section, we introduce background of privacy-preserving analytics in the cloud, including the set of queries that we want to support, as well as the settings under which the data are collected and processed.

2.1.1 Analytical Queries

In this thesis, we focus on the following types of analytical queries:

- **Select** `SELECT` $\mathsf{F}(A)$ `FROM` T `WHERE` C, which returns the function F on attribute A for all rows in table T that satisfy the condition C;

- **Join** `SELECT` $\mathsf{F}(A)$ `FROM` T_1 `JOIN` $\ldots$ `JOIN` T_k `ON` UID `WHERE` C, which returns the function F on attribute A for rows in the table, from joining rows in tables $T_1, \ldots, T_k$ on the join key UID, that satisfy the condition C;

- **Groupby** `SELECT` $\mathsf{F}(A)$ `FROM` T `WHERE` C `GROUP BY` B, which returns the function F on attribute A for rows in each group of attribute B, *i.e.*, each unique value of attribute B denoting one group, in table T that satisfy the condition C;

- **Project** `SELECT` $A_1 \ldots A_k$ `FROM` T, which returns all the rows in table T, only with the attributes $A_1, \ldots, A_k$;

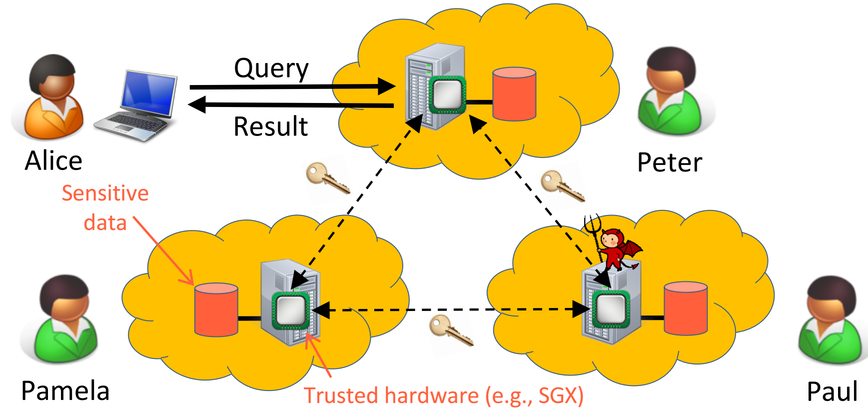


Figure 2.1: Example scenario. Analyst Alice queries sensitive data that is distributed across multiple machines, which are potentially owned by multiple participants. An adversary has complete control over some of the nodes, except the CPU.

- **Rename** `SELECT` A `AS` B `FROM` T, which returns attribute value of A, renamed as B, for all rows in table T;
- **Union** `SELECT` $*$ `FROM` T_1 `UNION` (`ALL`) `SELECT` $*$ `FROM` T_2, which combines the rows from the two `SELET` clauses into a single table. Duplicate rows are removed unless `ALL` is used;
- **Orderby** `SELECT` $*$ `FROM` T `ORDER BY` B, which returns all rows in table T, ordered by their values of attribute B.

This set of SQL queries is expressive, and can support all benchmark queries in the Big Data Benchmark [8] for SparkSQL.

2.1.2 Federated SQL Processing

In federated SQL processing, as depicted in Figure 2.1, there is a group of *participants*, who each own a sensitive data set, as well as a set of *nodes* on which the sensitive data is stored. An *analyst* can submit SQL queries that potentially involve data from multiple nodes, which we call *federated SQL queries*. We assume that the queries themselves are not sensitive – only their answers and the input datasets are – and that each node contains a trusted execution environment (TEE) that

supports secure enclaves and attestation, e.g., Intel's SGX. Federated SQL processing is a generalization of some earlier work [139, 178], which assumes only one participant that outsources one set of tables to a set of nodes, *e.g.*, in the cloud.

Threat model. We assume that some of the nodes are controlled by an adversary – for instance, a malicious participant or a third party who has compromised the nodes. The adversary has full physical access to the nodes under her control, except the CPU; she can run arbitrary software, make arbitrary modifications to the OS, and read or modify any data that is stored on these nodes; she can probe the memory bus between CPU and the memory to get the memory traces. This threat model inherently addresses all previous user-space side channel attacks [66, 67, 126, 170]. We explicitly acknowledge that the analyst herself could be the adversary, so even the queries could be maliciously crafted to extract sensitive data from a participant.

Central Differential Privacy (CDP).

CDP [50] is a rigorous notion about individual's privacy in the setting where there is a trusted data curator, who gathers data from individual users, processes the data in a way that satisfies DP, and then publishes the results. Intuitively, a query result is differentially private if a small change to the input only has a statistically negligible effect on the output distribution.

More formally, let $\mathcal{T}$ be the set of possible input tables. We say that two tables $T_1, T_2 \in \mathcal{T}$ are *neighboring* if they differ in at most one element. A randomized query processing algorithm $\mathcal{Q}$ with range R is (ϵ, δ)-differentially private if, for all possible sets of outputs $S \subseteq R$ and all neighboring input tables T_1 and T_2,

$$\mathbf{Pr}\left[\mathcal{Q}(T_1) \in S\right] \leq e^{\epsilon} \cdot \mathbf{Pr}\left[\mathcal{Q}(T_2) \in S\right] + \delta.$$

That is, with probability at least $1 - \delta$, any change to an individual element of the input data can cause at most a small multiplicative difference in the probability of *any* set of outcomes S. The parameter ϵ controls the strength of the privacy guarantee; smaller values result in better privacy,

but require more random noise to output. For more information on how to choose ϵ and δ, see, e.g., [72].

Differential privacy has strong composition theorems; in particular, if two query algorithms $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are (ϵ_1, δ_1)- and (ϵ_2, δ_2)-differentially private, respectively, then the combination $\mathcal{Q}_1 \cdot \mathcal{Q}_2$ is $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$-differentially private [49, 50]; note that it does not matter what specifically the queries are asking. Because of this, it is possible to associate each data set with a "privacy budget" ϵ_{max} that represents the desired strength of the overall privacy guarantee, and to then keep answering queries using $\mathcal{Q}_1, \ldots, \mathcal{Q}_k$ as long as $\sum_i \epsilon_i \leq \epsilon_{\text{max}}$. The latter is only a lower bound; the differential privacy literature contains far more sophisticated composition theorems [52, 84, 115, 137], and these can be used to answer far more queries using the same privacy "budget". However, to keep things simple, we will explain Hermetic using only simple composition.

Trusted Execution Environment

In this section, we introduce the necessary background on *trusted execution environment* (TEE), and discuss its state of the art in privacy-preserving computations.

In principle, privacy-preserving analytics could be achieved with fully homomorphic encryption [59] or secure multi-party computation [19], but these techniques are still orders of magnitude too slow to be practical [19, 60]. As a result, many systems use less than fully homomorphic encryption that enables some queries on encrypted data but not others. This often limits the expressiveness of the queries they support [124, 125]. In addition, some of these systems [63, 109, 128] have been shown to leak considerable information [48, 65, 117].

TEE is a technology that guarantees the confidentiality and integrity of data and code loaded into a special region of processor memory. Intel Software Guard Extensions (SGX) [38] is a set of instruction codes, built into commodity Intel CPUs, that implement the TEE. Two of its key components are *remote attestation*, which is used to prove to the user's client that it is interacting with the server setup with the correct environment, *e.g.*, SGX processor, and registered service

code, and *secure enclave*, which protects the confidentiality and integrity of code and data inside its memory region from any other processes, including the OS and hypervisor, on the same machine.

Recent work [11, 13, 122, 139, 178] rely on TEE, *e.g.*, Intel's SGX, to build privacy-preserving data analytics in the cloud. With this approach, the data remains encrypted even in memory and is only accessible within a trusted enclave within the CPU. As long as the CPU itself is not compromised, this approach can offer strong protections, even if the adversary has compromised the operating system on the machines that hold the data. But, as we will discuss in the next section, these systems are vulnerable to various side channel attacks.

Straw-man Solution

It may appear that the privacy-preserving federated SQL processing could be achieved as follows: 1) each participant leverages secure enclave and remote attestation to protect the confidentiality and integrity of data and computations on other participants, which is similar to the encrypt mode in Opaque [178]; 2) participant, P_i, sets a local privacy budget $\epsilon^{0}_{\text{max},i}$ for dataset d_i, and, for the t-th query, only *query plan* with valid ϵ_i – that is, $\epsilon_i \leq \epsilon^{t}_{\text{max},i}$, on each dataset d_i is allowed to execute, and then $\forall i, \epsilon^{t+1}_{\text{max},i} = \epsilon^{t}_{\text{max},i} - \epsilon_i$.

This approach would seem to meet our requirements: differential privacy ensures that a malicious analyst cannot compromise privacy, and the TEE ensures that malicious participants cannot get access to intermediate results and/or sensitive data from other nodes, even when the system must send such data to their own nodes as part of the query (e.g., to be joined with some local data). However, the straw-man solution implicitly assumes that the adversary can learn *nothing at all* from the encrypted data or from externally observing the execution in the TEE, which is not true in practice, due to the side channel vulnerabilties.

Side Channel Vulnerability

Even though trusted hardware can prevent an adversary from observing the data itself, the adversary can still hope to learn facts *about* the data by externally observing the execution in the enclave. In practice, there are several *side channels* that remain observable. The most devastating ones, first identified by Tople and Saxena [148], are:

- **Timing channel (TC) [95]**: The adversary can measure how long the computation in the enclave takes to infer the execution path, which leaks sensitive information if branches or indirect jumps depend on the data; and
- **Memory channel (MC):** The adversary can observe the locations in memory that the enclave reads or writes (even though the data itself is encrypted!), which leaks sensitive information if there are memory accesses indexed by the data; and
- **Instruction channel (IC):** The adversary can see the sequence of instructions that are being executed, e.g., by probing the branch target buffer [96], to infer the execution path, as in TC; and
- **Object size channel (OC):** The adversary can see the size of any intermediate results that the enclave stores or exchanges with other enclaves, and these size measurements could be used to reconstruct substantial information about the data [121].

In general, the connectivity pattern between the enclaves could be another channel, but in our setting, this pattern is a function of the (non-sensitive) query and not of the (sensitive) data, so we do not consider it here.

At first glance, the above channels may not reveal much information, but this intuition is wrong: prior work has shown that side channels can be wide enough to leak entire cryptographic keys within a relatively short amount of time [175]. To get truly robust privacy guarantees, it is necessary to close or at least mitigate these channels.

Side channel mitigation. Vuvuzela [151] and Xiao et al. [163] use differential privacy to obscure message sizes, Flicker [106] and SeCAGE [100] run protected code on locked-down cores,

CaSE [174] executes sensitive code entirely in the cache, CATalyst [99] uses Intel's CAT against cache-based side channels, and TRESOR [114] protect AES encryption key from physical memory attacks by locking the secret key inside persistent CPU registers. None of these systems would work in our setting because they either cannot address all the four side channels or cannot support the large-scale federated data analysis. As we have tried to show, the devil is usually in the details; thus, building a *comprehensive* defense against several channels remains challenging.

It is well known that SGX, in particular, does not (and was not intended to [82]) handle most side channels [38], and recent work has already exploited several of them. These include side channels due to cache timing [27], BTB [96], and page faults [169]. Raccoon is a compiler that rewrites programs to eliminate data-dependent branches [134]. Its techniques inspire the code modifications that Hermetic uses to mitigate the IC and TC. Another challenge is the fact that IA-32's native floating-point instructions have data-dependent timing; Hermetic uses `libfixedtimefixedpoint` [10] to replace them. Some solutions aim to *detect* the channels [33, 34, 140] rather than block them, and their effectiveness highly depends on how the adversary attacks, and the adversary in our threat model can easily bypass their detection.

We emphasize again that, after the discovery of the Foreshadow attack [150], current SGX implementations are no longer secure. However, the underlying problem is with the implementation and not with the design, so Intel should be able to fix it in its next generation of CPUs. Hermetic assumes a correct implementation of SGX – an assumption it shares with the entire literature on SGX-based systems.

Oblivious RAMs [62, 112, 144, 145] can eliminate leakage through MC in arbitrary programs, but they suffer from poor performance in practice [134]. Moreover, ORAMs only hide the addresses being accessed, not the number of accesses, which could itself leak information [178].

None existing systems can fully mitigate physical side channels, such as power analysis [90] or electromagnetic emanations [93] because the underlying CPU does not guarantee that its instructions are data-oblivious with respect to them. (Intel claims that AES-NI is resilient to digital

side-channels, but does not mention others [68].) However, these channels are most often exploited to infer a program's instruction trace, so, by making the IC of a query data-oblivious, Hermetic likely reduces these effectiveness of these channels.

Data Oblivious Analytics

M2R [46] and Ohrimenko et al. [121] aim to mitigate the OC in MapReduce. Both systems reduce OC leakage, but they use ad hoc methods that still leak information about the most frequent keys. By contrast, Hermetic's OC mitigation, based on differential privacy, is more principled. To address the MC in databases, Arasu et al. [12] introduce a set of data-oblivious algorithms for relational operators, based on oblivious sort. Ohrimenko et al. [122] extend this set with oblivious machine learning algorithms. Hermetic enhances these algorithms by making them resistant to IC, TC, and OC leakage and by speeding them up significantly using an OEE.

Opaque [178] combines TEEs, oblivious relational operators, and a query planner, and in that it shows substantial performance gains when small data-dependent computations are performed in oblivious execution environment. The key differences to Hermetic are: 1) Opaque does not mitigate the IC or TC, and it mitigates the OC by padding up to a public upper bound, which may be difficult to choose; and 2) by assuming (!) that the 8MB L3 cache of a Xeon X3 is oblivious, Opaque's implementation effectively assumes the existence of an OEE, but does not show how to concretely realize one. Hermetic's OEE primitive would be one way for Opaque to satisfy this requirement, and it would also add protections against two additional side channels.

There are systems [7,21,87] that combine TEEs and differential privacy for privacy-preserving data analysis. Prochlo [21] is focused on side channels during data collection, and could benefit from our techniques for protection during data analysis. The past year has seen the first work [7, 87] that investigates using differential privacy to compute padding for the OC in a data analytics system. This work proposes algorithms for several query operators, such as range queries, but its leaves supporting a more functional subset of SQL that includes joins to future work. It also does

not address how to integrate the privacy budget into query planning.

2.1.3 Data Collection and Analytics

The general data collection and analytics, namely *crowdsourcing* in [54], or *telemetry* in [43] consist of n users and K services. Each service collects users' data t, in the form of tuple(s) of *attributes*, with a client-side application. Some service collects exactly one tuple from each user. Other services may collect multiple tuples (*e.g.*, transactions). We denote the tuples collected by the i-th service as a relational table T_i. And we denote all the tuples collected from user u as T^u, and those from u collected by the i-th service as T_i^u. Thus, $\forall u \in [n], \bigcup_{i=1}^{K} T_i^u = T^u$, and $\forall i \in [K], \bigcup_{u=1}^{n} T_i^u = T_i$.

Besides the attributes about the users, we assume that each tuple has a pseudo-random user id (UID), which is known to the service provider and can serve as the non-sensitive join key. An analyst can jointly analyze the data across multiple services $S \subseteq [K]$ by joining their tuples, on the user id, as a *fact table* $\bowtie_{i \in S} T_i$.

In this thesis, we focus on answering *multi-dimensional analytical* (MDA) queries against joins of tables. Consider a function F that takes one of the form COUNT, SUM, or AVG on an attribute A. A multi-dimensional analytical query takes the format:

$$\texttt{SELECT}\ \mathsf{F}(A)\ \texttt{FROM}\ T_{i_1}\ \texttt{JOIN}\ \ldots\ \texttt{JOIN}\ T_{i_q}\ \texttt{ON}\ UID\ \texttt{WHERE}\ \mathsf{C},$$

where $\{T_{i_1}, \ldots, T_{i_q}\}$ are the tables to be joined on UID; C is the predicate on attributes of the joined fact table, and it can be conjunction of either point constraints for categorical attributes, or range constraints for ordinal attributes, or their combinations. We focus on conjunctions of constraints in this paper. Disjunctions can be handled by combinations of conjunctions, as described in [158].

Since user id is known to service providers, other attributes may leak sensitive information about each user. In the next two subsections, we will introduce the classical model of *local differential privacy*, and propose an enhanced version, called *user-level local differential privacy* (for

the setting with multiple services), to protect these sensitive attributes during data collection. In the rest part of this paper, for the ease of description, we assume that all the attributes in a query are sensitive. We can extend our techniques for queries with non-sensitive attributes by simply plugging their true values when evaluating aggregations or predicates as in [158].

Local Differential Privacy

Local differential privacy (LDP) can be used for the setting where each user possesses one value t from a fixed domain $\mathbb{D}$, and it offers a stronger level of protection, because each user only reports the noisy data rather than the true data, and even if the central curator is untrusted, the privacy of each individual participant is guaranteed.

Consider a randomized algorithm $\mathcal{R}(t)$ that takes $t \in \mathbb{D}$. The formal definition of privacy of $\mathcal{R}(t)$ is defined as follows:

Definition 1 (ϵ-Local Differential Privacy)**.** *A randomized algorithm $\mathcal{R}$ over $\mathbb{D}$ satisfies ϵ-local differential privacy (ϵ-LDP), where $\epsilon \geq 0$, if and only if for any input $t \neq t' \in \mathbb{D}$, we have*

$$\forall y \in \mathcal{R}(\mathbb{D}) : \ \mathbf{Pr}\left[\mathcal{R}(t) = y\right] \leq e^{\epsilon} \cdot \mathbf{Pr}\left[\mathcal{R}(t') = y\right],$$

where $\mathcal{R}(\mathbb{D})$ denotes the set of all possible outputs of $\mathcal{R}$.

In the definition, ϵ is also called the privacy budget. A smaller ϵ implies stronger privacy guarantee for the user, as it is harder for the adversary to distinguish between t and t'. Since a user never reveals t to the service but only reports $\mathcal{R}(t)$, the user's privacy is still protected even if the service is malicious.

Sequential composability. An important property, sequential composability [108], for LDP bounds the privacy on t when it is perturbed multiple times. We state the property in Proposition 2.

Proposition 2 (Directly from [108])**.** *Suppose $\mathcal{R}_i$ satisfies ϵ_i-LDP, the algorithm $\mathcal{R}$ which releases the result of each $\mathcal{R}_i$ on input t, i.e., $\mathcal{R}(t) = \langle \mathcal{R}_i(t) \rangle_{i=1}^{k}$, satisfies $\sum_{i=1}^{k} \epsilon_i$-LDP.*

LDP mechanisms. There have been several LDP frequency oracles [6, 16, 17, 54, 157] proposed. They rely on techniques like hashing (*e.g.*, [157]) and Hadamard transform (*e.g.*, [6, 16]) for good utility. LDP mean estimation is another basic task [47, 156]; and we use stochastic rounding [47] because it is compatible with *frequency oracles* (FO). FO is also used in other tasks, *e.g.*, finding heavy hitters [16, 28, 160], frequent itemset mining [133, 159], and marginal release [36, 135, 177].

Answering range queries. Range counting queries are supported in the centralized setting of DP via, *e.g.*, hierarchical intervals [71] (one-dim range queries) or via wavelet [164] (multi-dimensional range queries). [132] optimizes the hierarchical intervals in [71] by choosing a proper branching factor. McKenna *et al.* [107] propose a method to collectively optimize errors in high-dimensional queries of a given workload under the centralized setting of DP. It will be interesting to design such workload-dependent techniques under LDP.

In both marginal release and range queries, it has been noticed that constrained inference could boost the accuracy while enforcing the consistency across different marginal tables and intervals (*e.g.*, [15, 44, 71, 132]). Since it is a post-processing step, all the consistency enforcement techniques in the centralized setting of DP can be potentially used in the LDP setting.

Joint analysis in LDP. Several methods have been proposed to handle the joint analysis in LDP. In particular, [57] proposed to use EM algorithm. The starting point is to find on that maximizes the likelihood (possibility) of the observed report. Later, [160] derived formulas to direct evaluate the joint estimation, which essentially extends the aggregation function to multi-dimensional setting. Both methods work in the categorical setting. For the ordinal setting [158] proposed a method based on matrix inversion.

Frequent itemset mining in LDP. Existing work [133, 159] on LDP frequent itemset mining addresses the problem of identifying the frequent items from n users, each generates a subset of values from the domain $\mathbb{D}$, with ϵ-LDP. They share this core technique called *padding-and-sampling*, which pads the set of values of each user to a fixed size ℓ, and samples one to perturb and report. [159] further leverages the privacy amplification from sampling to boost the utility. The

key difference between our τ-truncation and the padding-and-sampling is that τ-truncation does not need padding because it enables unbiased estimation over tuples with distinct sampling ratios, and padding causes biased under-estimation (small ℓ) or amplified error bound (large ℓ).

SQL support in DP. In the centralized model of DP, where there is a *trusted party*, there are efforts to support SQL queries, including PINQ [108], wPINQ [131], Flex [81], and PrivateSQL [91]. These systems assume a trusted data engine [92] that maintains users' exact data, and injects noise during the (offline or online) analytical process so that query results transferred across the firewall ensures DP. Different from that, our paper assumes no such trusted party.

2.2 End-to-End Encrypted Search

2.2.1 Information Retrieval

In this section, we review the standard information retrieval (IR) techniques for keyword search. For a general introduction and literature overview we refer the reader to [103, 179].

Documents, terms, stemming. In this work, a *term* is an arbitrary byte string. A *document* is a multiset of terms; This is commonly called the "bag of words" model. This formalism ignores the actual contents of the document (e.g., binary data) and only depends on the terms that are output by a document parser, which we leave implicit for now and make explicit in our experimental evaluations. We assume all documents have an associated unique 4-byte *identifier* that is used by the underlying storage system, as well as a small amount of *metadata* for user-facing information (e.g. filename or preview). In our system we allow for 20 bytes of metadata.

The *term frequency* of a term w in document f, denoted $\mathsf{tf}(w, \mathsf{f})$, is the number of times that w appears in f. In a set of documents $D = \{\mathsf{f}_1, \mathsf{f}_2, \ldots\}$, the *document frequency* of a term w, denoted $\mathsf{df}(w, D)$, is the number of documents in D that contain w at least once.

Queries and ranking functions. In this paper, a *query* will be a set of terms. The most popular approach to ranking assigns a positive real-valued score to every query/document pair, and orders the

documents based on the scores. Intuitively, a document should receive a higher score if it contains a term from the query many times, and also terms that are rare in a corpus should be given more weight. This is accomplished via a formula that depends on term and document frequencies. This paper uses the industry standard ranking function BM25 [136]. For a query q, set of documents D, and document $\mathsf{f} \in D$, the BM25 score is

$$\mathsf{BM25}(q, \mathsf{f}, D) = \sum_{w \in q} \log\left(\frac{|D|}{\mathsf{df}(w,D)+1}\right) \frac{\mathsf{tf}(w,\mathsf{f})\cdot(k_1+1)}{\mathsf{tf}(w,\mathsf{f})+k_1(1-b+b\frac{|\mathsf{f}|}{|\mathsf{f}|_{avg}})} \ ,$$

where $|\mathsf{f}|$ ($|\mathsf{f}|_{avg}$) is the (average) document length (where length is simply the size of the multiset); k_1 and b are two tunable parameters, usually chosen in $[1.2, 2.0]$ and as 0.75, respectively. We note that to compute the BM25 score of a query for a given document, it is sufficient to recover the document frequencies of the term along with their term frequencies in the document.

Inverted indexes and compression. The standard approach for implementing ranked search is to maintain an *inverted index*. These consist of per-term precomputed *posting lists*. The posting list for a term w will contain some header information (e.g., the document frequency), and then a linked list of *postings*, which are tuples of document identifiers along with extra information for ranking (e.g., the term frequency):

$$\mathsf{df}(w, D)\ ,\ (\mathsf{id}_1, \mathsf{tf}(w, \mathsf{f}_1)), \ldots, (\mathsf{id}_n, \mathsf{tf}(w, \mathsf{f}_n))$$

where n is the number of documents that w appears within. A search is processed by retrieving the relevant posting lists (multiple ones in case of multi-keyword queries), computing the BM25 scores, and sorting the results. To improve latency, posting lists are usually stored in descending order of term frequency or document identifier (the latter improving multi-term searches, while the former allows for early termination of searches).

In practice, inverted indexes can be large but highly amenable to optimization and compression (c.f. [103], Chapter 5). Variable byte-length encodings are used for frequencies and other information. It is often profitable to use *delta encodings* where one stores the *differences* between consecutive document identifiers, rather than the values, since terms tend to cluster in nearby doc-

uments.

An example: Lucene Lucene [1] is a high-performance, full-featured text search engine library written in Java. It creates indices on different *field*s of documents, and, typically, the text body of the document are treated as `TextField` that are parsed into terms, which are indexed in the inverted index, and other information, *e.g.*, name, size and date, are treated as `StoredField` that are stored as is in the forward index. Lucene breaks the index into multiple sub-indices called *segment*s, each of which is a standalone index. Incoming updates are always added to the current opening segment in memory, and the opening one is committed to disk when closed or reaching the threshold on size. Lucene provides the flexibility on when to merge the segments into one for better space and search efficiency, and, depending on the application, the segments can be merged, *e.g.*, offline; when the number of segments reaches some threshold; or simply after every update. By default, the inverted index posting contains the document identifier and the term frequency. Optionally, it can include the positions of terms in the documents, *e.g.*, for phrase and proximity queries. As for the result ranking, Lucene provides two options, a TF-IDF relevance function and a BM25 relevance function with default $k_1 = 1.2$ and $b = 0.75$.

Lucene is also flexible on how a segment is encoded and written to disk. Along with the library, there are multiple index encoding classes, or *codec*s, and developers can customize with their own implementations. For Lucene 7.7.3, the default one is `Lucene50`, which encodes the inverted index in separate files for term index, term dictionary and postings, and applies the delta encoding and variable-byte encoding on numbers, *e.g.*, identifier and term frequency, in the index. The `Lucene50` codec implements the *skip list* technique with configurable skip levels to enable fast posting access in a posting list, *i.e.*, logrithmic to the its length, at the cost of extra index space of the skip pointers. In addition, `Lucene50` applies the `LZ4` compression on the `StoredField` in the forward index, but not the `TextField` in the inverted index. The simplest codec available in Lucene is `SimpleTextCodec`, which is a text-based index encoder that serializes the terms and postings into one big string, without any optimization.

2.2.2 Searchable Symmetric Encryption (SSE)

In this section we give our definitions for syntax, correctness, and security for encrypted indexes. Our definitions hew closely to prior ones on searchable symmetric (going back to Curtmola et al. [40]), except that we require a stateless client, relax the correctness definition, and change the search interface to request pages rather than entire lists.

Encrypted index syntax. Our setting requires a search scheme with no persistent client state beyond a key (e.g., a password). Any other needed persistent state must be stored at the server and downloaded as needed.

A *an encrypted index scheme* consists of two algorithms $\Pi = (\mathsf{Search}, \mathsf{Update})$, with associated sets KeySp, IDSp, MetaSp called the *key-space*, *identifier-space*, and *metadata-space* respectively. It also comes with an associated page size p (e.g., $p = 10$). Each protocol is between two parties, named the client and server, and we assume the server is deterministic. We require that they have the following syntax:

- $\mathsf{Update}(K, \mathsf{id}, \delta) \rightarrow C_{up}$: takes as input a key K, document identifier id and posting list (term/term frequency pairs) and outputs an update ciphertext C_{up}.
- $\mathsf{Search}(K, q, i, \mathrm{EDB}, \vec{C}_{up}) \rightarrow (R, \mathrm{EDB}')$: takes as input a key K, query q (one or more terms), a page number i, an encrypted index EDB, and a list of zero or more update ciphertexts $\vec{C}_{up}$. It outputs a new encrypted index EDB' as well as the plaintext result R, which is an ordered list of p document identifier, metadata pairs.

We assume inputs fall in their associated spaces, and that algorithms otherwise would abort on those inputs. In use, then, the output of Update is an encrypted update encoding. The sequence of ciphertexts taken as input to Search is the encrypted primary index (initially empty) and the outstanding update ciphertexts. Thus we have externalized from the semantics the storage and fetching of ciphertexts, which serves to simplify the formalism. We do not strictly define correctness, as we will instead measure accuracy via well-established scoring methods (see Section 5.7).

Security games. We use a real/ideal security definition parameterized by a *leakage profile*. Formally, a leakage profile is just a pair of functions $\mathcal{L}_{up}$ and $\mathcal{L}_{se}$. The first takes as inputs the same inputs as expected by Update, excepting the secret key K, and outputs the number of postings in the update. The second takes as input a history hist thus far of the queries to update and outputs the number of document identifiers n contained in hist and the aggregate number of postings N.

We formalize two games to capture security. Pseudocode descriptions appear in Figure 2.2. The first game, REAL_{Π}, captures an adversary $\mathcal{A}$ that is able to interact with an update oracle UP and search oracle SRCH on adaptively chosen inputs. The oracles run their respective algorithms using a secret key K (unknown to the adversary), and the outputs are the values that the server would see. The adversary outputs a bit, and we let "$\text{REAL}_{\Pi}(\mathcal{A}) \Rightarrow 1$" be the event that the output bit is one, defined over coins used by the game and $\mathcal{A}$.

The ideal game $\text{IDEAL}_{\mathcal{S}}^{\mathcal{L}}$ is parameterized by a leakage profile $\mathcal{L} = (\mathcal{L}_{up}, \mathcal{L}_{se})$ and a simulator $\mathcal{S} = (\mathcal{S}_{up}, \mathcal{S}_{se})$. In this world the two oracles are instead implemented by a combination of running the appropriate leakage function and handing the resulting input to the respective simulator algorithm (which can be randomized). The simulator algorithms share state; this is made explicit with an input and output bit string st shared by the algorithms. Ultimately again the adversary outputs a bit, and we let "$\text{IDEAL}_{\mathcal{S}}^{\mathcal{L}}(\mathcal{A}) \Rightarrow 1$" be the event that the output is one, defined over the coins used by the game including those used by $\mathcal{A}$ and $\mathcal{S}$.

Definition 3. *Let* $\Pi = (\mathsf{Update}, \mathsf{Search})$ *be an encrypted index scheme. Let* $\mathcal{A}, \mathcal{L}, \mathcal{S}$ *be algorithms. Define the* $\mathcal{L}$-advantage of $\mathcal{A}$ against Π and $\mathcal{S}$ *to be*

$$\mathrm{Adv}_{\Pi,\mathcal{S}}^{\mathcal{L}}(\mathcal{A}) = \Pr[\text{REAL}_{\Pi}(\mathcal{A}) \Rightarrow 1] - \Pr[\text{IDEAL}_{\mathcal{S}}^{\mathcal{L}}(\mathcal{A}) \Rightarrow 1] \ .$$

We take a concrete security approach, meaning we will upper bound adversarial advantage as an explicit function of the adversaries resources, rather than providing asymptotic definitions of security. Our results can be translated to an asymptotic treatment in a straightforward way.

This model does not include adversarial manipulation of ciphertexts before returning them to the client, which is standard to omit in searchable encryption settings. In fact our schemes resist

$\underline{\text{REAL}_{\Pi}(\mathcal{A})}$	$\underline{\text{IDEAL}^{\mathcal{L}}_{\mathcal{S}}(\mathcal{A})}$
$K \leftarrow \mathsf{KeySp}$ $\vec{C}_{up} \leftarrow \varepsilon$ $b \leftarrow \mathcal{A}^{\text{UP,SRCH}}$ Output b	$\mathsf{st} \leftarrow \varepsilon$ $\mathsf{hist} \leftarrow \bot$ $b \leftarrow \mathcal{A}^{\text{UP,SRCH}}$ Output b
$\text{UP}(\mathsf{id}, \delta)$ $C_{up} \leftarrow\$ \mathsf{Update}(K, \mathsf{id}, \delta)$ $\vec{C}_{up} \leftarrow \vec{C}_{up} \parallel C_{up}$ Return C_{up}	$\text{UP}(\mathsf{id}, \delta)$ $\mathsf{hist} \leftarrow \mathsf{hist} \Vert (\mathsf{id}, \delta)$ $(\mathsf{st}, \vec{C}_{up}) \leftarrow\$ \mathcal{S}_{up}(\mathsf{st}, \mathcal{L}_{up}(\mathsf{id}, \delta)))$ Return $\vec{C}_{up}$
$\text{SRCH}(q, i)$: $(R, \text{EDB}) \leftarrow\$ \mathsf{Search}(K, q, i, \text{EDB}, \vec{C}_{up})$ $\vec{C}_{up} \leftarrow \varepsilon$ Return EDB	$\text{SRCH}(q, i)$: $\mathsf{hist} \leftarrow \mathsf{hist} \Vert (q, i)$ $(\mathsf{st}, \text{EDB}) \leftarrow\$ \mathcal{S}_{se}(\mathsf{st}, \mathcal{L}_{se}(\mathsf{hist}))$ Return EDB

Figure 2.2: Games used in Definition 3.

many forms of such mauling attacks because we use authenticated encryption — a server could at best force replays of old ciphertexts or selectively drop updates. Preventing replays is impossible without client side state.

Related work. The study of efficient keyword search for encrypted data began with work by Song, Wagner, and Perrig [142] (SWP). They propose schemes for linear search of encrypted files, and mention that their approach can be applied instead to reverse indices that store a list of document identifiers for each keyword. Subsequent work by Kamara et al. [40] provided formal, simulation-based security notions and new constructions that leak less information than SWP's approaches. Cash et al. [30, 31] introduced some of the simplest known SSE schemes, by leveraging state on the client side (beyond the key).

SSE schemes fail to provide all the features to match the search features discussed above. The schemes mentioned so far only support single keyword searches, but follow-up works have suggested schemes that support boolean queries [31, 85]. Unfortunately they leak more information than needed about plaintext data, which impacts security negatively. Second, traditional SSE schemes do not support relevance ranking, pagination, or previews. Dynamic SSE [23–25, 30, 32, 40, 41, 56, 86, 88, 94, 110, 110, 142, 143, 171] schemes enable asymptotically efficient updating of encrypted indices in response to edited, new, or deleted documents, but view searching as an un-

ordered retrieval problem; Many recent works in fact do not refer to it as searching, but instead as building an encrypted multimap datastructure. Unfortunately all those schemes require state on the client side, which needs to be securely exported somehow to the server for lightweight clients.

In addition to mismatching offered and desired functionality, all existing, efficient SSE constructions are vulnerable to two classes of attacks. Leakage-abuse attacks (LAAs) [22, 29, 61, 64, 79, 116, 130, 152, 162, 176] exploit information leaked during search to recover information about searched keywords (which reveals partial information about a file's plaintext). LAAs arise because practical SSE schemes reveal result patterns: which (encrypted) files match on which (encrypted) searches. We emphasize that this is leaked whether or not a user actually (wants to) access a file. With some side information about the distribution of keywords across documents, an attacker can infer search terms, which also reveals some plaintext content for matching documents.

LAAs are mountable by what's often called a passive persistent adversary which observes all the queries to the service, but does not actively modify data. In contrast, the second class of attacks, called injection attacks [29], involve an adversary inserting maliciously chosen queries or files into the encrypted data store. For example, Zhang et al. [176] show an efficient attack that cleverly injects a small number of large files so that, when subsequent encrypted searches occur, the attacker immediately infers the searched keyword and, via the results pattern, partial information about other plaintexts. Blackstone et al. [22] combines document injection with the volumetric pattern leakage for more robust attacks.

A line of work on forward-private dynamic SSE [24, 25, 56, 88, 143] seeks to partially mitigate injection attacks by ensuring that past searches cannot be applied to newly added files. But this doesn't prevent injection attacks because they still work on all future queries. Applying expensive primtives, *e.g.*, ORAM [42, 146], on the encrypted index can prevent injection attacks based on the result pattern.

Summary. In summary, we do not currently have systems for searching encrypted documents that (1) come close to matching the functionality of contemporary plaintext search services; (2) that

work in the required deployment settings, including lightweight clients; and (3) that resist attacks.

CHAPTER 3

HERMETIC: PRIVACY-PRESERVING DISTRIBUTED ANALYTICS WITHOUT (MOST) SIDE CHANNELS

3.1 Introduction

Recently, several systems have been proposed that can provide *privacy-preserving distributed analytics* [139, 178]. At a high level, these systems offer functionality comparable to a system like Spark [172]: mutually untrusted data owners can upload large data sets, which are distributed across a potentially large number of nodes, and they can then submit queries across the uploaded data, which the system answers using a distributed query plan. However, in contrast to Spark, these systems *also* protect the confidentiality of the data. This is attractive, e.g., for cloud computing, where the data owner may wish to protect it against a potentially curious or compromised cloud platform.

It is possible to implement privacy-preserving analytics using cryptographic techniques [124, 128], but the resulting systems tend to have a high overhead and can only perform a very limited set of operations. An alternative approach [122, 139, 178] is to rely on *trusted execution environments (TEEs)*, such as Intel's SGX. With this approach, the data remains encrypted even in memory and is only accessible within a trusted *enclave* within the CPU. As long as the CPU itself is not compromised, this approach can offer strong protections, even if the adversary has compromised the operating system on the machines that hold the data.

However, even though trusted hardware can prevent an adversary from observing the data itself, the adversary can still hope to learn facts *about* the data by monitoring various side channels. Currently, the most notorious attacks exploiting such channels are the various Meltdown [98], Spectre [89], and Foreshadow [150] variants, which are due to vulnerabilities in current CPU designs. These channels are devastating because they affect a huge number of deployed CPUs, and, in the case of Foreshadow, even compromise the security of the trusted SGX enclave itself! However, from a scientific perspective, they are actually not the most dangerous ones: now that

the underlying vulnerabilities are known, they can be fixed in the next generation of CPUs [77]. In contrast, there are many other side channels – including the execution time of a target program [95], the sequence of memory accesses from the enclave [169], the number and size of the messages that are exchanged between the nodes [121], the contents of the cache [27], and the fact that a thread exits the enclave at a certain location in the code [96, 161] – that are more fundamental and will stay with us even after Meltdown, Spectre, and Foreshadow have been fixed.

Side channel leakage in privacy-preserving analytics has received considerable attention recently [11, 12, 121, 122, 178], but existing proposals still suffer from two main limitations. First, they do not attempt to mitigate the most critical side channels simultaneously or to evaluate the performance impact of doing so. For instance, Opaque [178] explicitly declares timing channels to be out of scope. Second, the mitigations that they do employ are unsatisfying. The most common approach is to pad computation time and message size all the way to their worst-case values. But as we show experimentally, full padding can drive up overhead by several orders of magnitude. Furthermore, existing attempts to avoid full padding suffer from one of the three problems: i) employing ad hoc schemes that lack provable privacy guarantees [121]; ii) relying on users to specify padding bounds a priori, which we believe is unrealistic [178]; and iii) failing to support complex data analysis due to limited expressiveness [7, 87].

This paper proposes Hermetic, which aims to address all of these limitations. Hermetic mitigates the four major digital side channels – timing, memory access patterns, instruction traces, and output sizes – a challenging design problem in itself, while achieving performance that is as good or better than systems with weaker privacy guarantees. Moreover, Hermetic combines a method of padding the execution time and result size of database queries that is substantially more efficient than full padding and a new privacy-aware query planner to provide principled privacy guarantees for complex queries, i.e., those with multiple join operations, and not unreasonably burdening the user.

To achieve these goals, we employ a three-pronged strategy either to optimize the critical com-

ponents or to fill in the missing parts in previous solutions. The first part is a primitive that can perform fast, non-oblivious sort securely, by "locking-down" core and carefully transforming the program to achieve several key properties. This primitive, namely *oblivious execution environment (OEE)*, protects against digital side channels and yet improves efficiency. Importantly, we also provide a concrete implementation of an OEE: although related primitives have been discussed previously (e.g., in [55, 178]), existing work either simply assumes it without details or achieves weaker security definitions than OEE does (e.g., in [39, 106].

The second element of our approach is a set of oblivious query operators that are faster and more secure than traditional oblivious operators. In particular, we enable the first secure inner equi-join that is resilient to the four major side channels, and avoids the prohibitive overheads from full padding by relaxing the privacy guarantee on output size channel to differential privacy [50]. The third element is a privacy-aware query planner that generates an efficient query plan that respects the user's specification on privacy and performance. Differential privacy introduces a set of privacy parameters that affect both the overall performance and privacy cost of the query plan to the query optimization problem, and we solve the problem over the entire privacy parameter space, as defined by user's input, while existing solutions have to assume a fixed set of privacy parameters, which misses huge optimization opportunity.

We have implemented a Hermetic prototype that runs on commodity machines. Since current SGX hardware is not yet able to fully support the "lockdown" primitive we propose, we have implemented the necessary functionality in a small hypervisor that can be run on today's equipment. Our experimental evaluation shows that our approach is indeed several orders of magnitude more efficient than full padding, which has been the only principled side channel mitigation in analytics systems against the timing and output side channels. At the same time, we show that Hermetic has comparable performance to existing SGX-based analytics systems while offering stronger privacy guarantees.

We note that Hermetic is not a panacea: like all systems that are based on trusted hardware,

it assumes that the root of trust (in the case of SGX, Intel) is implemented correctly and has not been compromised. Also, there are physical side channels that even Hermetic cannot plug: for instance, an adversary could use power analysis [90] or electromagnetic emanations [93], or simply de-package the CPU and attack it with physical probes [141]. These attacks are more difficult and expensive to carry out than monitoring digital side channels with software.[1] Thus, despite these caveats, Hermetic brings privacy-preserving distributed analytics closer to practicality. Our contributions are as follows:

- the design of the Hermetic system (Section 3.2),
- the OEE primitive, which performs fast, non-oblivious computations privately on untrusted host (Section 3.3),
- enhanced oblivious operators, including a new oblivious join algorithm with DP padding, that limit four different side channels (Section 3.4),
- a novel privacy-aware query planner (Section 3.5),
- a prototype of Hermetic (Section 3.6), and
- a detailed experimental evaluation (Section 3.7)

3.2 The Hermetic System

In this section, we describe the workflow of federated query processing in Hermetic, and defer the detailed discussions on each of the key components, as well as the prototype implementation, to Sections 3.3, 3.4, 3.5, and 3.6

3.2.1 Overview

Hermetic consists of a master node, and several worker nodes. These nodes can switch roles for data proximity, load balance or policy regulations. Each node runs the trusted hypervisor to support

1. They may also be impossible to eliminate without extensive hardware changes. Yet, as physical side channel attacks often aim to infer control flow, we speculate that Hermetic's mitigations of the instruction trace side channel may help mitigate these physical channels as well.

OEEs (Section 3.3), and the trusted runtime, inside a TEE, that includes the Hermetic operators (Section 3.4) and a light-weight plan verifier. The last component of Hermetic is the untrusted query planner (Section 3.5). The workflow of Hermetic consists of the following steps:

1. Initially, the master node launches the hypervisor and runtime, and the data owners contact the runtime to setup master encryption keys and upload their data (Section 3.2.2). Data owners verify the authenticity of both the hypervisor and the runtime via attestation (Section 3.2.3).
2. After initialization, analysts can submit queries to the query planner, which generates a concrete query plan and forwards it to the runtime (Section 3.2.4).
3. As the planner is outside trusted computing base, the runtime verifies incoming plans to make sure that all operators are annotated with the appropriate sensitivities and ε's (Section 3.2.4).
4. If verification passes, the runtime organizes the worker nodes to execute the query plan using Hermetic's oblivious operators. (Section 3.2.5).

We describe these steps in greater detail below.

3.2.2 Initialization

Hermetic is initialized after the data owners set up master encryption keys and upload their sensitive data to the server. Since no party in Hermetic is completely trusted, the master keys are created inside the trusted runtime, using randomness contributed by the data owners. After that, the keys are encrypted using a hardware-based key and persisted to secondary storage using, e.g., SGX's sealing infrastructure [9].

With the master key in place, data owners send their data, together with the associated privacy budgets, to the runtime, which encrypts it with the key and stores it to the disk.

3.2.3 Attestation

A prerequisite for uploading sensitive data is that data owners can be convinced that they are sending the data to a correct instantiation of the Hermetic system. This means that they need to

make sure that the Hermetic hypervisor is running on the remote machine and that the Hermetic runtime is running in a TEE. We achieve this level of trust in two stages. First, upon launch, the Hermetic runtime uses a mechanism such as Intel's trusted execution technology (TXT) [75] to get an attestation of the code loaded during the boot process. If the hypervisor is absent from the boot process, the Hermetic runtime halts. Second, data owners leverage enclave attestation, e.g., Intel SGX attestation [9], to verify that the correct runtime is running in the TEE.

3.2.4 Query submission and verification

Analysts write their queries in a subset of SQL that supports `select`, `project`, `join`, and `groupby` aggregations. Analysts can supply arbitrary predicates, but they cannot run arbitrary user-defined functions. Analysts submit queries to the query planner, which is outside Hermetic's TCB. The planner then prepares a query plan to be executed by the runtime.

As explained in Section 3.5, query plans are annotated with the sensitivity of each relational operator, as well as with the ε for adding noise to the intermediate results. Since the planner is not trusted, these parameters have to be verified, so that the enough amount of noise will be added for the required privacy, before the plan is executed: Hermetic has to check that the sensitivities are correct by computing them from scratch based on the query plan, and that the total ε annotations do not exceed the privacy budgets. δ is a system parameter enforced by Hermetic runtime, and it is not explicitly annotated or verified in the plan (Section 3.4.2). Correctness and efficiency are out of the scope of Hermetic's verification.

The untrusted platform could launch a rollback attack [20, 26, 104, 147], in which it tricks the trusted runtime into leaking too much information by providing it with a stale copy of the privacy budget. To ensure the freshness of the stored privacy budget, the runtime must have access to a protected, monotonically-increasing counter. This counter could be implemented using a hardware counter, such as the one optionally available with SGX [74] – possibly enhanced with techniques to slow wear-out of the counter [20, 147]. Alternatively, it could be implemented with a distributed

system consisting of mutually-distrusting parties [26, 104].

3.2.5 Query execution

If a plan passes the verification, it is executed by the runtime. Before execution starts, the privacy budget on each relation is decreased based on the εs in the plan, and the runtime generates the Laplace noise which determines the number of dummy tuples to pad intermediate results with. To execute a query plan, the Hermetic runtime sends all the individual operators of the plan to different Hermetic worker nodes, which in turn use the appropriate operators from Section 3.4 to perform the computation.

3.3 Oblivious Execution Environments

The first part of Hermetic's strategy to mitigate side channels is hardware-assisted oblivious execution, using a primitive we call an *oblivious execution environment (OEE)*.

3.3.1 OEE properties

The goal of oblivious execution is to compute a function $out := f(in)$ while preventing an adversary from learning anything other than f and the sizes $|in|$ of the input and $|out|$ of the output - *even if*, as we have assumed, the adversary has access to TC, MC, IC and OC.

To provide a solid foundation for oblivious execution without performing dummy memory accesses, we introduce a primitive OEE (f,in,out) that, for a small set of predefined functions f, has the following four properties:

1. Once invoked, OEE runs to completion and cannot be interrupted or interfered with;
2. OEE loads in and out into the cache when it starts, and writes out back to memory when it terminates, but does not access main memory in between;
3. The execution time, and the sequence of instructions executed, depend only on f, $|in|$, and $|out|$; and

4. The final state of the CPU depends only on f.

A perfect implementation of this primitive would plug all four side channels in our threat model: The execution time, the sequence of instructions, and the sizes of the in and out buffers are constants, so no information can leak via the TC, IC, or OC. Also, the only memory accesses that are visible on the memory bus are the initial and final loads and stores, which access the entire buffers sequentially, so no information can leak via the MC. Finally, since the adversary cannot interrupt the algorithm, she can only observe the final state of the CPU upon termination, and that does not depend on the data.

Note, however, that OEE is allowed to perform data-dependent memory accesses *during* its execution. Effectively, OEE is allowed to use a portion of the CPU cache as a private, un-observable memory for the exclusive use of f. This is what enables Hermetic to provide good performance.

3.3.2 Challenges in building an OEE today

Prior work has recognized the performance benefits of having an un-observable environment for performing data-dependent functions. Zheng et al. [178] speculate that a future un-observable memory would allow a system to perform ordinary quick-sort instead of expensive oblivious sort. Unfortunately, actually realizing an OEE, especially on current hardware, is challenging.

We can achieve property #3 by eliminating data-dependent branches and by padding the execution time to an upper bound via busy waiting (Section 3.6.2). We can also disable hardware features such as hyper-threading that would allow other programs to share the same core, and thus potentially glean some timing information from the OEE. Properties #2 and #4 can be achieved through careful implementation (Sections 3.3.3, 3.6.2). Finally, by executing the OEE in a TEE, e.g., SGX, enclave, we can ensure that the data is always encrypted while in memory.

However, even if we ignore the vulnerabilities from [150], today's TEE, e.g., SGX, *cannot* be used to achieve property #1. By design, SGX allows the OS to interrupt an enclave's execution at any time, as well as flush its data from the cache and remove its page table mappings [38]. Indeed,

these limitations have already been exploited to learn secret data inside enclaves [27, 96, 169]. Flicker [106] achieves property #1 by suspending the entire machine, except the sensitive program, using the SKINIT instruction, and we want to achieve property #1 without suspending concurrent processes.

3.3.3 The Hermetic OEE

Realizing an OEE requires that f in OEE be adapted with deterministic instruction sequences and constrained memory footprints, e.g., by avoiding recursion. Algorithm 3.1 shows how we achieve the oblivious instruction trace, in terms of op-codes, for the core `merge-sort` function. We unify all the conditional branches, including those depending on the data values and the operation mode, into one execution path using the `cwrite` primitive. Hermetic actually optimizes Algorithm 3.1 so that the sorted order is kept in the global `oee_buffer` after `merge-sort`-ing on the sorting attributes, and the rest of the attributes are linearly re-ordered following the sorting order, without the cost of `merge-sort`.

Furthermore, the CPU core that the OEE is running on must be configured so that no other processes can interrupt it or interfere with its state. To achieve the latter, Hermetic relies on a thin layer of hypervisor to configure the underlying hardware for the isolation, as shown in Algorithm 3.2. Before an OEE can execute, the hypervisor (1) completely "locks down" the OEE's core by disabling all forms of preemption – including IPIs, IRQs, NMIs, and timers; (2) disables speculation across the OEE boundary to mitigate Spectre-style attacks by setting the appropriate CPU flags or using serialization instructions [77]; (3) configures Intel's Cache Allocation Technology (CAT) [118] to partition the cache between the OEE's core and the other cores at the hardware level; (4) prevents the OS from observing the OEE's internal state by accessing hardware features such as performance monitoring; (5) flushes legacy data from previous executions in the isolated cache partition; and (6) when the function completes, restore the hardware configuration, and flushes the legacy state of the OEE. In Section 3.6.1, we present further details of the hypervisor's

design and its use of the CAT.

Second, the program in OEE should be prefixed with sanitization to preload all program instructions and memory states into the isolated cache partition, and postfixed with cleansing to deterministically pad the execution time and flush the cache-lines. In Section 3.6.2, we present further details of the preloading and time padding. Note that one OEE invocation can process upto `OEE_SIZE` tuples, and the total size is bounded by the last-level cache partition. We describe how to build large-scale operators in Section 3.4.

We view the hypervisor as interim step that makes deploying Hermetic possible today. In terms of security, the Hermetic hypervisor is equivalent to the *security monitor* in [39], and they both have small TCBs that can be formally verified and attested to. Its functionality is constrained enough that it could be subsumed into future versions of TEE, e.g., SGX. We believe that this paper and other recent work on the impact of side channels in TEEs demonstrates the importance of adding OEE functionality to TEEs.

3.4 Oblivious operators

OEEs provide a way to safely execute simple computations, e.g., sorting, on blocks of data, while mitigating side channels. However, to answer complex queries over larger data, Hermetic needs higher-level operators, as described next.

Our starting point is the set of so-called oblivious query operators, introduced in prior work [12, 122], whose memory access patterns depend only on their input size, not the specific values. These include inherently oblivious relational operators, such as `project`, `rename`, `union` and `cartesian-product` as well as operators based on oblivious sorting networks (e.g., Batcher's odd-even merge-sort [18] — BatcherSort). Oblivious sort is the basis for auxiliary oblivious operators like `group-running-sum`, `filter`, `semijoin-aggregation` and `expand`, which in turn can be combined to form familiar relational such as `select`, `groupby`, `orderby` and `join`. In particular, an oblivious inner equi-join could be constructed by first applying

```
MergeSort(mode):

1: for len ∈ {2^0, ..., 2^{log(end)}} do
2:   for off ∈ {0, 2·len, ..., ⌊end/(2·len)⌋·2·len} do
3:     right ← MIN(off + 2·len − 1, end)
4:     pos_1 ← off; pos_2 ← off + len
5:     for cur ∈ {0, ..., right − off} do
6:       cond_a ← (pos_1 ≤ off + len − 1)
7:       cond_b ← (pos_2 ≤ right); cond_c ← 0
8:       cwrite(cond_a, cur_1, pos_1, off + len − 1)
9:       cwrite(cond_b, cur_2, pos_2, right)
10:      for f ∈ attrs do
11:        cwrite((cond_c = 0), cond_c, tb[cur_1][f] − tb[cur_2][f], cond_c)
12:      cond_d ← ((cond_c ≤ 0) = ascend)
13:      cond_1 ← (mode = "PRELOAD")?(pos_1 <= (pos_2 − len)) : (cond_a · cond_d > cond_b − 1)
14:      for f ∈ {0, ..., FIELDS_PER_RUN} do
15:        cwrite(cond_1, tb[OEE_SIZE + off + cur][f], tb[cur_1][f], tb[cur_2][f])
16:      cwrite(cond_1, pos_1, pos_1 + 1, pos_1)
17:      cwrite(cond_1, pos_2, pos_2, pos_2 + 1)
18:    cwrite((mode = "PRELOAD"), l, 0, OEE_SIZE)
19:    for cur ∈ {0, ..., right − off} do
20:      tb[off + cur] ← tb[off + l + cur]

cswrite(cond, out, in_1, in_2):

1: asm volatile("test eax, eax"
2: "cmovnz %2, %0","cmovz %3, %0"
3: : "=r"(*out) : "a"(cond), "r"(in_1), "r"(in_2) : "cc",
   "memory")
```

Figure 3.1: OEE MergeSort with data oblivious instruction trace. `end`, `ascend`, `attrs` are function parameters pre-loaded into global `oee_buffer`, and `tb` points into the `oee_buffer` for storing tuple input and output. `OEE_SIZE` indicates the maximum number of tuples that one OEE invocation could process.

`semijoin-aggregation` [12] on the two input relations to derive the *join degree*, namely the number of matches for a tuple from the other relation, then `expand`-ing the two relations based on join degree following *equally-interleaved expansion* [122], obliviously sorting the expanded relations ordered by join attributes and expansion id, and, finally, `stitch`-ing them together to get the

OEESort($\mathcal{R}$, attr, order):

```
1:  Disable preemption/interrupts
2:  Set speculative execution boundary
3:  Configure CAT for last-level cache isolation
4:  Disable PMC read
5:  Flush the entire cache partition
6:  Load attr, order into global oee-buffer
7:  for every FIELDS_PER_RUN attributes ∈ R do
8:      Load the attributes of R to oee-buffer
9:      Linear scan over oee-buffer                        ▷ cache data
10:     MergeSort (mode = "PRELOAD")                       ▷ cache code
11:     MergeSort (mode = "REAL")
12:     Copy sorted attributes from oee-buffer to R
13: Pad the execution time
14: Flush cache & restore H/W configurations
```

Figure 3.2: OEE isolation.

result. Note that , before Hermetic, there is no existing system that supports data-oblivious inner equi-join for federated data analysis. See Appendix A.1.1 for more details about these operators.

3.4.1 Extending oblivious operators

Existing oblivious query operators are vulnerable to TC and IC. We eliminate the data-dependent branches by unifying them into one execution path using the `cwrite` primitive (Algorithm 3.1). We avoid instructions with data-dependent timing following [10].

Furthermore, we accelerate existing oblivious operators, by replacing BatcherSort with HybridSort that leverages OEEs (See Algorithm 3.3). HybridSort is faster than BatcherSort because each block data in OEE is sorted by faster MergeSort (Line 3). This would accelerate the `select`, `groupby` and `join` whose performance is dominated by oblivious sort.

Finally, enabling Hermetic's query planner to trade off privacy and performance (Section 3.5) requires obliviously collecting statistics about the input data. To do so, we introduce two new primitives that leverage OEEs: `histogram` that computes a histogram over the values in a given attribute, and `multiplicity` that computes the multiplicity of a attribute – i.e., the number of

```
HybridSort(R = {t_0, ..., t_n}, attr, order):

 1: if |R| ≤ OEE_SIZE then
 2:     OEESort(R, attr, order)
 3: else
 4:     HybridSort({t_0, ..., t_{n/2}}, attr, order)
 5:     HybridSort({t_{n/2+1}, ..., t_n}, attr, order)
 6:     HybridMerge(R, attr, order)

HybridMerge(R = {t_0, ..., t_n}, attr, order):

 1: if |R| ≤ OEE_SIZE then
 2:     OEEMerge(R, attr, order)
 3: else
 4:     HybridMerge({t_0, ..., t_{n/2}}, attr, order)
 5:     HybridMerge({t_{n/2+1}, ..., t_n}, attr, order)
 6:     for i ∈ {2, 4, ..., n − 2} do
 7:         swap ← ((t_i[attr] ≤ t_{i+1}[attr] = order)
 8:         cwrite(swap, t'_i, t_{i+1}, t_i)
 9:         cwrite(swap, t'_{i+1}, t_i, t_{i+1})
10:         t_i ← t'_i; t_{i+1} ← t'_{i+1}
```

Figure 3.3: The OEE-assisted HybridSort primitive.

times that the most common value appears.

3.4.2 Differentially-private padding

Hermetic adopts an efficient approach to mitigating OC effectively, without padding the number of output tuples of an intermediate query operator to its worst-case value. It determines the amount of padding to add based on a truncated, shifted Laplace mechanism that ensures non-negative noise size. In particular, for an operator O_i with estimated sensitivity s_i – the maximum change in the output size that can result from adding or removing one input tuple, and the privacy parameter ϵ_i, $\Delta \sim Lap(o_i, s_i/\epsilon_i)$, where o_i indicates the offset of the shifted Laplace distribution from 0, dummy tuples are added to the output if $\Delta \geq 0$; Otherwise, 0 dummy tuple is added. This mechanism provides (ϵ, δ)-differential privacy [49], where δ corresponds to the probability of truncation –

i.e., $\delta_i = P_{trunc} = Pr[Lap(o_i, s_i/\epsilon_i) < 0]$. In addition, P_{trunc} is configured, as a system parameter in Hermetic, with very small value to minimize leakage. Note that the idea of applying differentially private padding to OC is not new, and has been investigated in [7, 87]. But Hermetic enables DP padding on operators, such as inner equi-join, that are considered as future work of [7, 87]. Furthermore, the problem of optimizing a plan, the outputs of whose operators are padded using DP, for both privacy and efficiency is a challenging problem, and Hermetic introduces a new privacy-aware query planner to address this challenge (Section 3.5).

To implement this approach, relations must be padded with *dummy tuples* to hide the true size of query results. As in prior work, we identify dummy tuples by adding an additional `isDummy` attribute to each relation, adapt operators like `select`, `groupby`, and `join` to add dummy tuples to their results, and adapt query predicates to ignore tuples where `isDummy == TRUE` (See Appendix A.3 for details). Moreover, the actual noise value must be kept hidden from an adversary. As a result, the number of dummy tuples has to be sampled in an enclave, and the sampling process must be protected from side channels, especially timing [10].

3.5 Privacy-aware Query Planning

In this section, we describe how Hermetic assembles the operators from the previous section into query plans that can compute the answer to SQL-style queries. Query planning is a well-studied problem in databases, but Hermetic's use of differential privacy adds a twist: Hermetic is free to choose the amount of privacy budget ϵ it spends on each operator. Thus, it is able to make a tradeoff between privacy and performance: smaller values of ϵ result in stronger privacy guarantees but also add more dummy tuples, which slows down subsequent operators.

Query planning in Hermetic follows the same design as in [166], and Figure 3.4 illustrates the key query planning steps for a counting query over three datasets:

Sensitivity estimation: For each possible execution plan tree , Hermetic query planner derives the upper bound on the sensitivities of all the operators O_i in the plan. To do this, the untrusted query

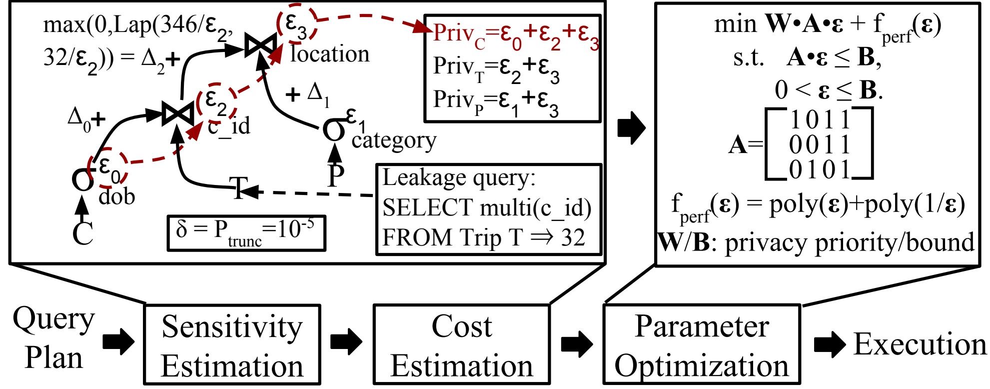


Figure 3.4: Query planning for query SELECT COUNT($*$) FROM C, T, P WHERE C.cid $=$ T.cid AND T.location $=$ P.location AND C.age ≤ 27 AND P.category $=$ "hospital".

planner could initiate auxiliary queries, which we call *leakage queries*, to compute the number of tuples in each operator's output. For instance, in Figure 3.4, the planner uses a leakage `select` query with the `multiplicity` operator on the joined attribute, `c_id`, to get an upper bound on the sensitivity of the leftmost join. The leakage queries are differentially private just like ordinary queries, and their (small) cost is charged to the (ϵ, δ) budget as usual; thus, the planner does not need to be trusted.

Cost estimation: Hermetic planner estimates the symbolic privacy and performance costs of the plan. The privacy cost of a plan, each operator O_i of which is assigned with privacy budget ϵ_i, is simply $(\sum_i \epsilon_i)$, but estimating the performance cost is more challenging. To obtain a performance model, we derived the complexity of the Hermetic operators as a function of their input size; the key results are shown in Table 3.5. Hermetic uses the `histogram` operator to estimate the size of intermediate results, following [70, 127]. These estimates are used only to predict the performance of the query plans; thus, if the adversary were to carefully craft the data to make the estimate inaccurate, the worst damage they could do would be to make Hermetic choose a slower query plan. To enable the planner to assess the performance implications of the dummy tuples, Hermetic takes them into account when estimating relation sizes, and the expected number of dummy tuples added by operator O_i, following the distribution $Lap(o_i, s_i/\epsilon_i)$, is simply the offset o_i.

Figure 3.5: Cost model for Hermetic's relational operators. n: first input size; c: OEE capacity; m: second input size; k: output size.

Operator	Cost
HybridSort	$h(n) = n \cdot \log(c) + n \cdot \log^2(n/c)$
`SELECT`	$h(n)$
`GROUP BY`	$3 \cdot h(n)$
`JOIN`	$4 \cdot h(n+m) + 2 \cdot h(m) + 3 \cdot h(n) + 2 \cdot h(k)$

Parameter optimization: Hermetic's query planner uses multi-objective optimization [149] to find the optimal query plan that matches the user's priorities. In particular, the user specifies a vector of bounds, $\mathbb{B}$, and a vector of weights, $\mathbb{W}$, for the privacy costs on all the input relations. The planner outputs the plan, whose weighted sum of all the costs, including privacy and performance whose weight is always 1, is optimal, and all of the privacy costs are within the bounds. We leave the details on the optimization technique in Appendix A.4 due to space limit.

3.6 Implementation

To evaluate our approach, we built a prototype, which we now describe, focusing on the hypervisor and OEEs.

3.6.1 Hermetic hypervisor

Hermetic's hypervisor extends Trustvisor [105], a compact, formally verified [153] hypervisor. To enable OEEs to "lock down" CPU cores, we added two hyper-calls — `LockCore` and `UnlockCore` — to Trustvisor. `LockCore` works as follows: (1) it checks that hyper-threading and prefetching are disabled (by checking the number of logical and physical cores using `CPUID`, and using techniques from [154]), (2) it disables interrupts and preemption (3) it disables the `RDMSR` for non-privileged instructions to prevent snooping on package-level performance counters of the OEE's core, (4) it flushes all cache lines (with `WBINVD`[2]), (5) it uses CAT [119] to assign a part of the LLC

2. Note that the timing variations of `clflush` and `prefetch` that are exploited in [66, 67] are not a problem for Hermetic simply because the memory addresses, during flushing and preloading, are observable, in a deterministic

exclusively to the OEE core. `UnlockCore` reverts actions 2–5 in reverse order. These changes required modifying 300 SLoC of Trustvisor.

3.6.2 Oblivious execution environments

Recall that the goal of OEEs to is create an environment where a carefully-chosen function can perform data-dependent memory accesses that are unobservable to the platform. To achieve this, we must make the function's memory access patterns un-observable and its timing predictable.

Un-observable memory accesses: To make memory accesses un-observable, we ensure that all reads and writes are served by the cache: we disable hardware prefetching and preload all data and code into the cache prior to executing the function. To preload data, we use `prefetcht0`[2], which instructs the CPU to keep the data cached, and we perform dummy writes to prevent leakage through the cache coherence protocol. To preload code, the function first executes in PRELOAD mode to exercise all the code — loading it into the icache — but processes the data deterministically. We align all buffers, especially `oee_buffer`, to avoid cache collisions.

Predictable timing: To ensure that an OEE function's execution time is predictable regardless of the input data, we employ a three-pronged approach. First, we statically transform the function's code to eliminate data dependent control flow (See Algorithm 3.1) and instructions with data-dependent timing. Second, although we allow the function's memory accesses to be data-dependent, we carefully structure it to constrain the set of possible memory accesses, thereby making the number of accesses that miss the L1 cache, and thus must be served by slower caches, predictable. For example, MergeSort (See Algorithm 3.1) performs a single pass over the input and output buffers to ensure that there are few cache misses per iteration. Finally, to account for timing variation that might occur in modern superscalar CPUs (e.g., due to pipeline bubbles), we pad the OEE's execution time to a conservative upper bound that is calibrated to each specific model of CPU. This bound is roughly double the execution time and was never exceeded in our

manner, to the adversary anyway.

Figure 3.6: Experimental configurations and their resilience to different channels (`MS`: MergeSort, `BS`: BatcherSort, `HS`: HybridSort, `CP`: cartesian product, `SMJ`: sort-merge join).

Configuration	Sort	Join	MC	IC	TC	OC
NonObl	MS	SMJ	✗	✗	✗	✗
DOA-NoOEE	BS	[12, 122]	✓	✗	✗	✗
Full-Pad	BS	CP	✓	✓	✓	✓
HMT I	HS	Section 3.4	✓	✓	✓	✗
HMT II	HS	Section 3.4	✓	✓	✓	✓

Figure 3.7: Schema and statistics of the relations. Synthetic was generated with a variety of tuples and multiplicities.

Query	Relation	Tuples	Multiplicities
S1-3	Synthetic	*	*
Q4-6	Trips	$10.00M$	m(cid)=32, m(loc.)=1019
	Customers	$4.00M$	m(cid)=1
	Poi	$0.01M$	m(loc.)=500
BDB1-3	rankings	$1.08M$	m(url)=1
	uservisits	$1.16M$	m(url)=22

experiments. For more information, see Appendix A.2.

3.6.3 Trusted computing base

Hermetic's TCB consists of the runtime (3,995 SLoC) and the trusted hypervisor (14,095 SLoC). The former may be small enough for formal verification, and the latter has, in fact, been formally verified [153] prior to our modifications. Recall that the hypervisor would not be necessary with a future TEE that natively supported "locking down" CPU cores.

3.7 Evaluation

This sections presents the results of our experimental evaluation of Hermetic's security and performance, a comparison with Opaque, the current state-of-the-art, and a discussion of the ability of Hermetic's query planner to trade off privacy and performance.

3.7.1 Experimental setup

Very recently, Intel has started offering CPUs that support both SGX and CAT; however, we were unable to get access to one in time. We therefore chose to experiment on an Intel Xeon E5-2600 v4 2.1GHz machine, which supports CAT, with 4 cores, 40 MB LLC, and 64GB RAM. This means that the numbers we report do not reflect any overheads due to encryption in SGX, but, as previous work [178] reports, the expected overhead of SGX in similar data-analytics applications is usually less than 2.4x. We installed the Hermetic hypervisor and Ubuntu (14.04LTS) with kernel $3.2.0$. We disabled hardware multi-threading, turbo-boost, and H/W prefetching because they can cause timing variations.

Table 3.6 shows the different system configurations we compared, and the side channels they defend against. NonObl corresponds to commodity systems that take no measure against side-channels; DOA-NoOEE uses data-oblivious algorithms from previous work [12, 122], without any un-observable memory; Full-Pad pads output of all operators to the maximum values: pads output of `SELECT` to the input size, and pads output of `JOIN` to the product of the two input sizes using Cartesian join; and HMT I and HMT II implement the techniques described in this paper – the only difference being that the former does not add noise to the intermediate results.

Table 3.7 lists all the relations we used in our experiments. The `Trips` relation has 5-days-worth of records from a real-world dataset with NYC taxi-trip data [120]. This dataset has been previously used to study side-channel leakage in MapReduce [121]. Since the NYC Taxi and Limousine Commission did not release data about the `Customers` and points of interest (`Poi`) relations, we synthetically generated them. To allow for records from the `Trips` relation to be joined with the other two relations, we added a synthetic customer ID attribute to the trips table, and we used locations from the `Trips` relation as `Poi`'s geolocations. To examine the performance of Hermetic for data with a variety of statistics, we use synthetic relations with randomly generated data in all attributes, except those that control the statistics in question. We use the `rankings` and `uservisits` from Big Data Benchmark (BDB) [8] for the comparison with Opaque.

3.7.2 OEE security properties

To verify the obliviousness of MergeSort (`MS`) and `linear- merge` (`LM`), we created synthetic relations, populated with random values and enough tuples to fill the available cache ($187,244$ tuples of 24 bytes each).

First, we used the Pin instrumentation tool [102] to record instruction traces and memory accesses; as expected, these depended only on the size of the input data. Second, we used Intel's performance counters to read the number of LLC misses[3] and the number of accesses that were served by the cache[4]; as expected, we did not observe any LLC misses in any of our experiments. Finally, we used `objdump` to inspect the compiler-generated code for instructions with operand-dependent timing; as expected, there were none. More details can be found in Appendix A.2.1.

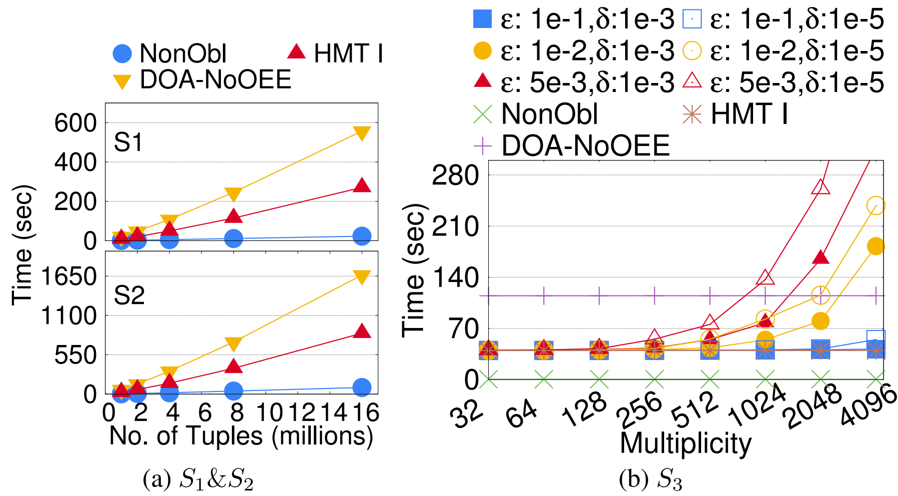
Next, we verify the timing obliviousness of `MS` and `LM` in OEE using cycle-level measurement. as expected, the execution time without padding could vary by tens of thousands of cycles between data sets, but with padding, the variations were only 44 and 40 cycles, respectively. (See Appendix A.2.3) As Intel's benchmarking manual [123] suggests, this residual variation can be attributed to inherent inaccuracies of the timing measurement code.

3.7.3 Performance of relational operators

Next, we examined the performance of Hermetic's relational operators: `SELECT` , `GROUP BY` and `JOIN`. For this experiment we used three simple queries ($S_1 - S_3$), whose query execution plans consist of a single relational operator. S_1 selects the tuples of a relation that satisfy a predicate, S_2 groups a relation by a given attribute and counts how many records are per group. S_3 simply joins two relations. To understand the performance of the operators based on a wide range of parameters, we generated relations with different statistics (e.g., selection and join selectivities, join attribute multiplicities) and used NonObl, DOA-NoOEE , and Hermetic to execute queries $S_1 - S_3$ on these

3. Using the `LONGEST_LAT_CACHE.MISS` counter.

4. Using the `MEM_UOPS_RETIRED.ALL_LOADS` and `MEM_UOPS_RETIRED.ALL_STORES` counters.



(a) S_1&S_2 (b) S_3

Figure 3.8: Performance of SELECT (S_1), GROUP BY (S_2) and JOIN (S_3) for different data sizes and join multiplicities.

relations.

Figure 3.8a shows the results for queries S_1 and S_2 for relations of different size. In terms of absolute performance, one can observe that HMT I can scale to relations with millions of records, and that the actual runtime is in the order of minutes. This is definitely slower than NonObl, but it seems to be an acceptable price to pay for protection against side channels, at least for some applications that handle sensitive data. In comparison to DOA-NoOEE , HMT I achieves a speedup of about 2x for all data sizes. S_3 displays similar behavior for increasing database sizes.

We also examined the performance of HMT II for query S_3 on relations of different multiplicities. The amount of noise added to the output in order to achieve differential privacy guarantee is proportional to $-s/\varepsilon \cdot ln(2\delta)$, and sensitivity s is equal to the maximum multiplicity of the join attribute in the two relations. Figure 3.8b shows the performance of HMT II with various ε and δ values, compared to other configurations, and differential privacy only affects the overall performance for very small ε, δ and large multiplicity (around 200). In addition, the line for Full-Pad is missing as it cannot finish the query within 7 hours due to its huge padding overheads for tackling OC.

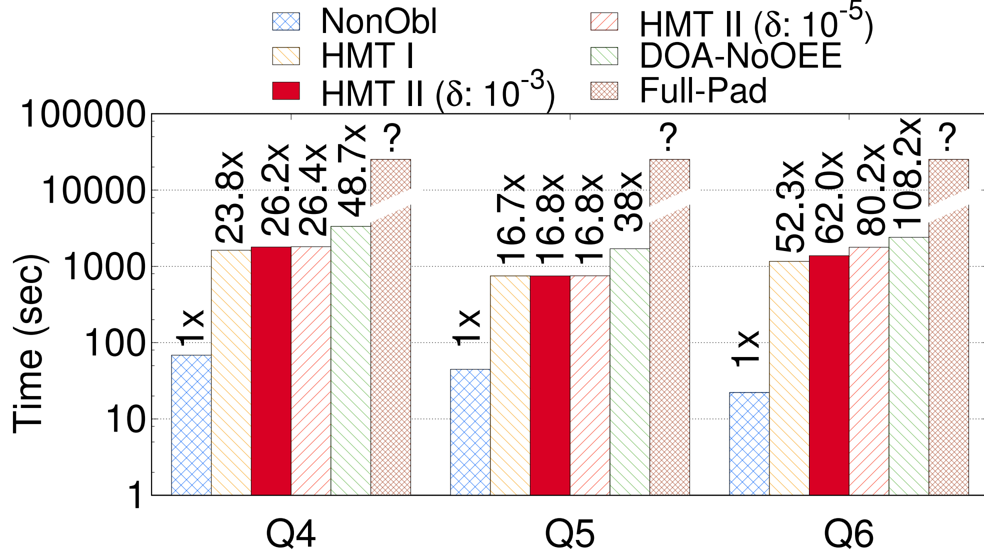


Figure 3.9: Performance of all configurations for Q_4 – Q_6.

3.7.4 End-to-end performance

We compared the different system configurations on complex query plans, each of which consists of at least one `SELECT` , `GROUP BY` , and `JOIN` operator. To perform this experiment, we used the relations described in Table 3.7, as well as three queries that perform realistic processing on the data. Q_4 groups the `Customer` relation by age and counts how many customers gave a tip of at most $\$10$. Q_5 groups the points of interest relation by category, and counts the number of trips that cost less than $\$15$ for each category. Q_6 counts the number of customers that are younger than 30 years old and made a trip to a hospital.

We measured the performance of all systems on these three queries, and the results are shown in Figure 3.9. For HMT II, the system optimized for performance, given a constraint $\varepsilon_{max} \leq 0.05$ on the privacy budget, and we set $\delta = P_{trunc}$ as 10^{-3} and 10^{-5} (Section 3.4.2). Full-Pad was not able to finish, despite the fact that we left the queries running for 7 hours. This illustrates the huge cost of using full padding to combat the OC. In contrast, HMT II, which pads using differential privacy, has only a small overhead relative to non-padded execution (HMT I). This suggests that releasing a controlled amount of information about the sensitive data can lead to considerable savings in

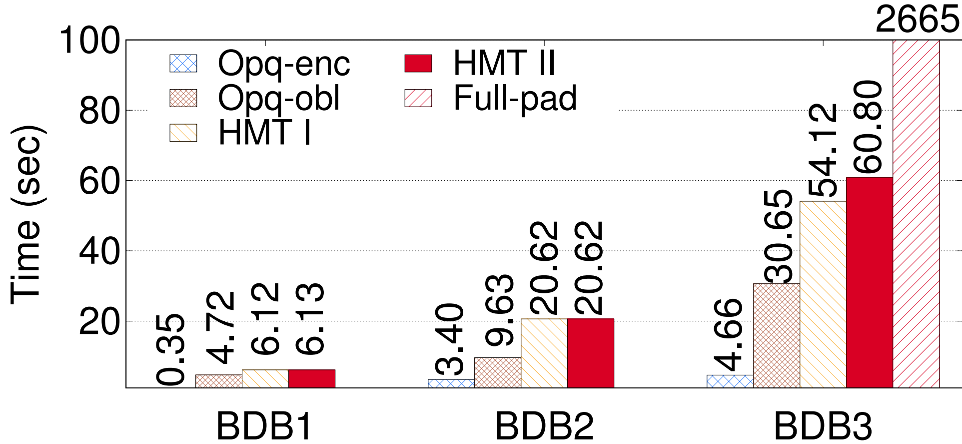


Figure 3.10: Comparison with Opaque.

terms of performance. Also, note how HybridSort helps Hermetic be more efficient than previous oblivious processing systems (DOA-NoOEE), *even though it offers stronger guarantees*.

3.7.5 Comparison with the state-of-the-art

We compare Hermetic with the state-of-the-art privacy-preserving analytics system, Opaque [178] using the big data benchmark (BDB) under the same configuration as in the previous experiments. In particular, we measure the execution times of the first three queries in BDB (BDB1-3) on the first 3 and 2 parts of the `1node` version of the `rankings` and `uservisits` datasets, respectively. Because the Hermetic prototype only supports integer attributes, we replace all `VARCHAR` and `CHAR` attributes with integers for both Hermetic and Opaque. For Hermetic, we consider the mode without differential privacy padding (HMT I) and the differential padding mode with ($\epsilon = 10^{-3}, \delta = 10^{-5}$) (HMT II). For Opaque, we consider the encryption (`enc`) and oblivious (`obl`) modes, and the oblivious memory size is fixed as 36MB. As the "oblivious pad mode" described in [178] is not implement in the release Opaque version, we estimate its performance using Hermetic's Full-Pad mode on BDB3. We also ensure that Hermetic and Opaque use the same query plans for all three queries.

The comparison results are shown in Figure 3.10. First, without differentially-private padding, Hermetic achieves comparable efficiency to Opaque in oblivious mode. The main reason is that

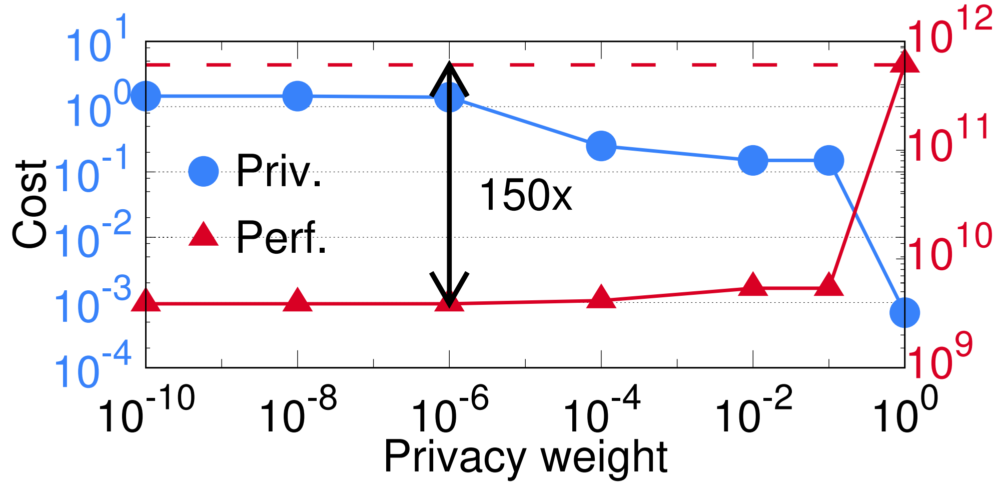


Figure 3.11: Total privacy and performance costs at various points in the trade-off space.

both Hermetic and Opaque leverage an un-observable memory to accelerate the oblivious sort, and Hermetic realizes such memory while Opaque simply assumes it. The difference in BDB2 and BDB3 is caused by the different implementations: Opaque requires two oblivious sorts for both `GROUP BY` and `JOIN` while Hermetic uses three and eleven oblivious sorts for `GROUP BY` and `JOIN`, respectively. Hermetic could be optimized using the `GROUP BY` algorithm in Opaque with one fewer oblivious sort. The Opaque's `JOIN`, although with fewer oblivious sorts, is restricted to primary-foreign key join, and Hermetic's `JOIN` can handle arbitrary inner equi-join, e.g. Q6. Second, because the sensitivities for BDB are small (1 for BDB1-2 and 22 for BDB3), the overhead from differential privacy padding in HMT II is very small compared to HMT I. Third, Full-Pad is 43x slower than HMT II due to huge padding overheads. In summary, even with stronger security guarantees, Hermetic achieves comparable efficiency to Opaque in oblivious mode. With comparable guarantees, Hermetic out-performs Opaque ("oblivious pad mode") substantially.

3.7.6 Trading-off privacy and performance

We tested the Hermetic query planner with weight vectors of various preferences to verify whether the planner could adjust the total privacy cost of all relations and the overall performance cost following the input weight vectors. We set the weight on privacy of each relation in Q_6 as iden-

tical, and increased the relative weight over performance cost. Figure 3.11 shows that the planner will sacrifice performance for lower privacy consumption when privacy is the first priority. The red dashed line indicates the performance cost of a plan without privacy awareness. Due to the unawareness, the plan has to assign the minimal privacy parameter to all the operators so as to handle the most private possible queries, and this could lead to huge inefficiency, e.g., 150x slow down, compared to Hermetic's plans for less private queries. Hermetic is also able to optimize the privacy cost on the individual relations based on the analyst's preferences, as expressed by the weight vector as shown in [166].

CHAPTER 4
COLLECTING AND ANALYZING DATA OVER MULTIPLE SERVICES WITH LOCAL DIFFERENTIAL PRIVACY

4.1 Introduction

Sensitive data, or *attributes*, about users' profiles and activities is collected by enterprises and exchanged between different services in one organization to help make informed data-driven decisions. The *de facto* privacy standard, differential privacy (DP) [51], has been deployed in several scenarios to provide rigorous privacy guarantees on how attributes are collected, managed, and analyzed. Informally, differential privacy requires that the output of data processing varies little with any change in an individuals' attributes.

The centralized model of DP assumes that a *trusted party* collects exact attribute values from users. The trusted party is inside a physical or logical privacy boundary [92], and injects noise during the (offline or online) analytical process so that query results transferred across the firewall ensure DP. Systems along this line [81, 91, 92, 108, 131] extend the class of data schemes and queries supported with improving utility-privacy trade-off. A use case is that a data engine managing customers' sensitive data, *e.g.*, in Uber [81], provides a query interface satisfying DP for its employees.

In the absence of the central trusted party, the *local differential privacy* model (LDP) [47] is adopted. Each user has her attribute values locally perturbed by a randomized algorithm with LDP guaranteed, *i.e.*, the likelihood of any specific output of the algorithm varies little with input; each user can then have the perturbed values leave her device, without the need to trust the data collector. Analytical queries can be answered approximately upon a collection of LDP perturbed values. Apple [5], Google [54], and Microsoft [43] deploy LDP solutions in their applications on users' device to estimate statistics about user data, *e.g.*, histograms and means.

In this paper, we investigate how to collect and analyze multi-dimensional data under LDP

in a more general setting: each user's attributes are collected by multiple independent services, with LDP guaranteed; and data tuples from the same user collected across different services can be joined on, *e.g.*, user id or device id which is typically known to the service providers. Two natural but open questions are: *what privacy guarantee can be provided on the joined tuples for each user*, and *what analytics can be done on the jointly collected (and independently perturbed) multi-dimensional data*.

Data model for joint collection and analytics. Figure 4.1 illustrates a target scenario. Two users u and u' are active in service 1 and service 2. Attribute tuples T_1^u and $T_1^{u'}$ are collected in service 1 from u and u', respectively; and it is similar for service 2. Indeed, each tuple is perturbed by a randomized algorithm $\mathcal{R}$ to ensure LDP. Conceptually, T_1 (or T_2) is a relational table holding users' information collected from service 1 (or service 2); and perturbed versions of T_1 and T_2 can be joined on user id, which is naturally known to both service providers. The questions are: whether it is safe to release the joined tuples in $\mathcal{R}(T_1) \bowtie \mathcal{R}(T_2)$, and what analytics can be conducted on it. Here is a motivating example.

Example 4.1.1 (E-Commerce)**.** *A user profiling service and a transaction processing service collect separate information about users in a E-commerce platform. The user profiling service collects attributes, including* Age *and* Income*, into table* User*. The transaction processing service collects the attributes about transactions, including* Amount *and* Category*, into table* Transaction*. Each user has a unique* UID*, which is randomly generated at sign-up, across the two services. An analyst wants to know the average amount on sports products for users in specific age group, e.g.,*

```
SELECT AVG(Amount) FROM
Transaction JOIN User ON User.UID = Transaction.UID
WHERE Category = Sports AND Age ∈ [20, 40].
```

The goal of this paper is *to support a class of* multi-dimensional analytical *queries against joins of tuples with attributes collected via multiple services from data owners*. A query here aggregates (COUNT, SUM, or AVG) on attributes of tuples under point and range predicates on the

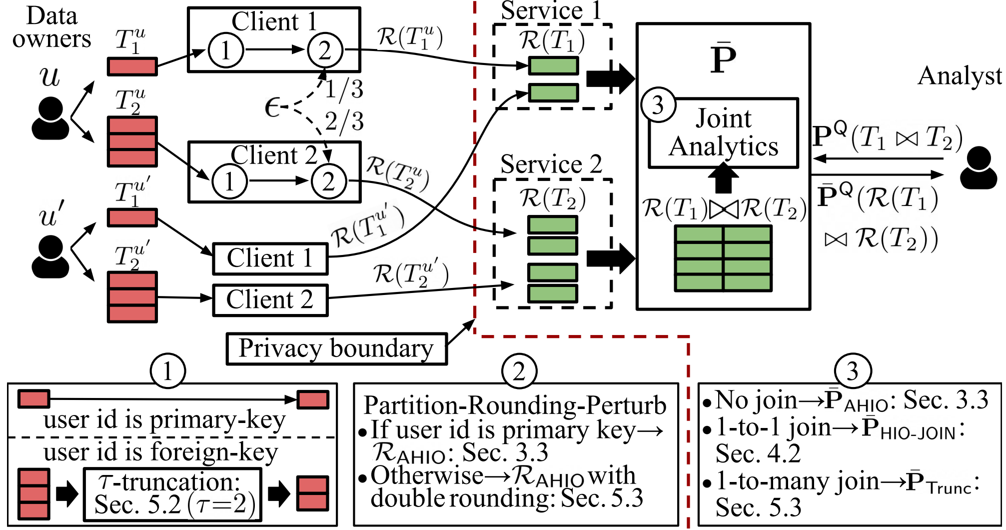


Figure 4.1: Data collection and analytics across multiple services (top), and an algorithmic pipeline of our solution (bottom).

attributes (in the WHERE clause)—all attributes in the aggregation and predicates could be sensitive and come from different services. Meanwhile, we want to provide strengthened privacy guarantee (*i.e.*, *user-level local differential privacy* as introduced later) for every single user given that her tuples collected from different services can be linked together with her user id (known to service providers).

Challenge I (joint aggregation). The setting and the query class studied in this paper are much broader than previous works in two important aspects. **1)** We allow all attributes (except user ids which are naturally known to service providers) in the multi-dimensional analytical queries to be sensitive, while existing works on answering range queries [37, 158] and releasing marginals [36, 135, 177] under LDP cannot handle aggregations on sensitive attributes. Note that it is easy to handle non-sensitive attributes by simply plugging their true values when evaluating aggregations or predicates as in [158]. **2)** Most previous works on multi-dimensional analytics, *e.g.*, [158] and [177], collect and analyze data from a single service only; in our setting, attributes in a query may come from different services, each of which perturbs its sensitive attributes independently—existing methods only work when these attributes are perturbed as a whole, which is impossible if

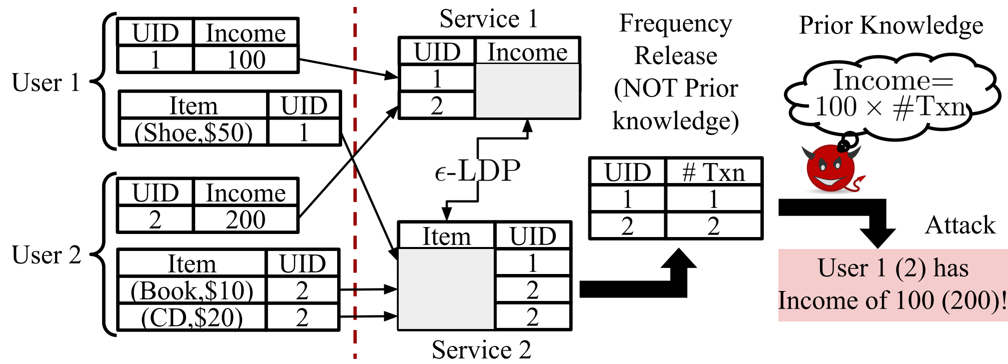


Figure 4.2: Frequency-based attack (# Txn = number of transactions).

these attributes are from different services.

Challenge II (frequency-based attack). We want to point out that it is *insufficient* (in terms of privacy) to ensure LDP independently in each service, given the fact that tuples collected in different services can be joined on user id which is known to service providers.

A straw-man solution based on ϵ-LDP works as follows: at data collection, divide the privacy budget ϵ among the tuples from different services for each user, and perturb each tuple using an LDP mechanism with proper budget. The hope is that, based on the sequential composition, the overall privacy guarantees for each user is ϵ-LDP. However, it turns out that this solution cannot provide any differential privacy for a user because it releases the sensitive information, the numbers of tuples that a user generates while using a service, after tuples are joined on user id. Such exact frequency release enables various attacks to users, depending on the prior knowledge of the adversary. For instance, as shown in Figure 4.2 (an instance of Example 4.1.1), if the adversary has the prior knowledge that the Income of a user is equal to the number of transactions multiplied by 100, then having access to the exact number of transactions of a user (even though each transaction is perturbed under LDP) enables the recovery of the sensitive Income of the user.

The above frequency-based attack is particularly feasible in our target scenario because we assume that the same user id is attached to tuples collected from a user, and is accessible by the service providers. User id is not a sensitive attribute, but can function as a join key and allows the adversary to group tuples of the same user.

Contributions and Solution Overview

Our main contribution is to extend the setting and query class supported by existing LDP mechanisms from single-service frequency queries to the more complex ones, including aggregations on sensitive attributes and multi-service joint aggregation; as we point out in challenges I-II, no existing LDP mechanisms can handle our target setting and query class with formal privacy and utility guarantees. We first give an overview of important algorithmic components in our solution and introduce the end-to-end pipeline.

• *Partition-rounding-perturb framework.* We propose a framework to extend the class of multi-dimensional analytical queries with aggregations on sensitive attributes, which is left as an open problem in [158]. The main idea is to first randomly round the value of an attribute to the two ends, with rounding probability carefully chosen so that the aggregation on the rounded variable is an unbiased estimation of the true aggregation of the attribute. The rounded variable with two possible values is treated as a new sentitive attribute of the user and then fed into the local perturbation mechanism for LDP, *e.g.*, the one in [158]. We note that the rounding may incur some utility loss, depending on how large attribute domain is, but overall, the loss is still dominated by the part incurred by the local perturbation. Moreover, users are randomly partitioned to boost the utilty when there are multiple attributes to be aggregated.

• *Independent perturbation and joint analytics (*HIO-JOIN*).* In our solution, tuples collected from different services are perturbed independently, and we need new estimation algorithm to estimate aggregations on the joins of these tuples. Our solution HIO-JOIN generalizes the split-and-conjunction technique and the mechanism HIO based on hierarchical intervals [158] to estimate the joint distribution on the vector of perturbed tuples and evaluate how likely the joint predicate is true. To improve the utility, we show that the standard trick of randomly partitioning users [37] also works here. In addition, we propose an utility optimization technique, by enforcing consistency across different levels in the hierarchical intervals.

• *User-level LDP and τ-truncation.* To tackle the frequency-based vulnerability for rigorous pri-

vacy guarantees, we formally define *user-level local differential privacy* (uLDP), which gathers all the information about a user (after the join) and makes it indistinguishable as whole just as in the classical LDP notation. For one-to-one joins, the above innovations suffice to ensure uLDP. However, in one-to-many joins, a user can generate one or more tuples in a particular service; these tuples can be joined with tuples generated by her in other services. How many tuples are joinable for her (*i.e.*, *frequency*) is part of the output of the data collection mechanism, and thus needs to be hidden (otherwise, uLDP is violated).

We propose the τ-truncation technique to hide such frequency information. Informally, no matter how many tuples are joinable for a user, we randomly sample (or copy) τ of them and feed them into the perturbation step. Each sample tuple is associated with a weight (which is inversely proportional to the sampling ratio) to compensate for the contribution to the aggregation from tuples not in the sample. This weight is a new sensitive attribute (of the user), as it depends on the true frequency. A technique called *double rounding* is proposed to perturb these weights together with other attributes and to derive unbiased estimation of the aggregation.

Overview of Solution Pipeline. Our solution requires minimum coordination between different services. The whole pipeline of data collection and analytics is described in Figure 4.1 (bottom part).

For each user and each service, ①: if user id is the primary key of tuples generated from this service (*e.g.*, user profiling service), one tuple is to be collected per user; if user id is a foreign key of tuples from this service (*e.g.*, transaction service), the number of tuples per user is uncertain. In the latter case, we use our τ-truncation technique to randomly sample (or create) τ tuples for each user and collect them, even if zero tuple is generated from this service (meaning that the user has not used this service), in order to hide the frequency/existence information about this user in the service. Then, depending on how many tuples in total are to be collected from all the services after τ-truncation (*e.g.*, in Figure 4.1, 3 tuples are collected per user in total), our privacy budget is evenly divided, and ②: each tuple from each service is perturbed independently using

our partition-rounding-perturb framework before collected to enable both frequency and attribute aggregations (if the tuple contains user id as the primary key, then a single rounding on an attribute is applied; otherwise, double rounding is needed).

Note that the data collection is *query-independent*—we do not have to know in advance which attributes will appear in the query and where they are. The only information that needs to be coordinated between services is the value of τ (which will be specified later) and the total number of tuples to be collected for each user.

To process a multi-dimensional aggregation query, tuples (with attributes in the query) collected from different services are joined on user id, and ③: the query is rewritten into frequency queries on their joint distribution, which are combined into the query result: if the query only contains attributes in a single table, *i.e.*, no join, we apply the estimation of partition-rounding-perturb, plugged with that of HIO; if it contains attributes across multiple tables, all with user id as primary key, *i.e.*, 1-to-1, we apply the estimation of partition-rounding-perturb, plugged with that of HIO-JOIN; and if it contains attributes across multiple tables, with user id as foreign key in some table, *i.e.*, 1-to-many, we apply the estimation of partition-rounding-perturb, plugged with those of HIO-JOIN and double rounding.

Organization. We first extend the basic LDP to the user-level privacy guarantee, *i.e.*, uLDP, in Section 4.2. Section 4.3 focuses on handling aggregations of sensitive attributes. Section 4.4 focuses on joint aggregation with one-to-one relation. We address the frequency-based attack in joint aggregation with one-to-many relation in Section 4.5. Section 4.6 introduces our utility optimization technique. Experimental results are reported in Section 4.7.

4.2 User-level Privacy across Multiple Services

In our setting, multiple services collect tuples from users and thus sensitive information comes from separate domains. For instance, in Example 4.1.1, one service collects users' profile, while the other collects users' online shopping transactions. Definition 1 cannot quantify the privacy for

this setting. Note that [43] studies LDP in an industrial deployment with multiple services, each of which collects one telemetry data. Here, we define the privacy guarantee formally in a more general setting, where a user can generate arbitrary number of tuples when using a service. In addition, we assume tuples of a user are collected only once, which is orthogonal to the continuous observation model [83].

In this paper, we consider the worst case where different services come together to infer the user's sensitive information, and we aim to protect the privacy of a user over the joint domain for all possible tuples across the K services. Suppose the i-th service collects tuples from the domain $\mathbb{D}_i$. A user u may generate zero, one, or multiple tuples when using the i-th service, denoted by a multiset $T_i^u \subseteq \mathbb{D}_i$. Across the K services, the user generates tuples $\langle T_1^u, \ldots, T_K^u \rangle$ in the joint domain. We define the user-level privacy spanning multiple services below:

Definition 4 (ϵ-uLDP)**.** *A randomized algorithm $\mathcal{R}$ over the joint domain across K services is user-level ϵ-locally differentially private (ϵ-uLDP), if and only if for any two users u, u', and their collected tuples $T^u = \langle T_1^u, \ldots, T_K^u \rangle$ and $T^{u'} = \langle T_1^{u'}, \ldots, T_K^{u'} \rangle$ in the joint domain s.t. $\exists i \in [K]$: $T_i^u \neq T_i^{u'}$, we have:*

$$\forall y \in \mathcal{R}(\mathbb{D}_{1,\ldots,K}): \ \mathbf{Pr}\left[\mathcal{R}(T^u) = y\right] \leq e^{\epsilon} \cdot \mathbf{Pr}\left[\mathcal{R}(T^{u'}) = y\right],$$

where $\mathcal{R}(\mathbb{D}_{1,\ldots,K})$ is the output domain across the K services.

When $K > 1$ and $\forall u, u', |T_i^u| = |T_i^{u'}|$, all users have the same number of tuples collected by each service, and the only privacy loss is from the joint distribution of values of the user's tuples, which we can bound using the basic LDP, plus sequential composition. And when $K > 1$ and $\exists u, u', s.t., |T_i^u| \neq |T_i^{u'}|$, uLDP covers the more general setting where a user can generate arbitrary number of tuples for a service, which is the focus of our work.

ϵ-uLDP is a general privacy definition for the multi-service data collection setting, and it covers the frequency-based attacks described in Section 5.1: for users u, u', with different numbers of tuples on the i-th service, *i.e.*, $|T_i^u| \neq |T_i^{u'}|$, the straw-man mechanism, *i.e.*, splitting ϵ among T^u ($T^{u'}$), and perturbing each of them with LDP, provides no uLDP guarantees. That is, for y with

$|T_i^u|$ perturbed values for the i-th service, where $i \in [K]$, the straw-man has $\mathbf{Pr}[\mathcal{R}(T^u) = y]$ being non-zero while $\mathbf{Pr}[\mathcal{R}(T^{u'}) = y]$ being zero, which implies ∞-uLDP, or no privacy.

In Section 4.3, 4.4, we mainly focus on the utility for our target query class, and assume each service collects exactly one tuple from each user. In Section 4.5, we propose mechanism that achieves ϵ-uLDP and prevents the frequency-based attacks in the general setting.

4.3 Attribute Aggregation

To handle multi-dimensional analytical queries under LDP, existing works propose multi-dimensional *frequency oracle*s over single tables. In this section, we first review two such LDP oracles. In order to estimate aggregations on sensitive attributes, we introduce a new class of *sensitive-weight frequency oracles* based on *stochastic rounding* and a new framework called *partition-rounding-perturb*, which allows the same sensitive attribute to appear in both aggregations and predicates of the queries. Two instantiations of this framework are introduced, with provable error bounds.

4.3.1 Building Block: Frequency Oracles

A multi-dimensional frequency oracle under LDP is a pair of algorithms, $(\mathcal{R}, \bar{\mathbf{P}})$: $\mathcal{R}$ follows LDP definition (Definition 1); and $\bar{\mathbf{P}}^{\mathsf{C}}(y)$ is a deterministic algorithm that takes y, *i.e.*, the output of $\mathcal{R}$, and outputs its estimated contribution to the predicate $\mathsf{C} \subseteq \mathbb{D}$. In addition, we denote the true result of the original tuple as $\mathbf{P}^{\mathsf{C}}(t)$.

For a fact table T of n tuples, we denote $\bar{\mathbf{P}}^{\mathsf{C}}(\mathcal{R}(T)) = \sum_{y \in \mathcal{R}(T)}$ $\bar{\mathbf{P}}^{\mathsf{C}}(y)$ as the estimated frequency aggregation of C against T, and $\mathbf{P}^{\mathsf{C}}(T)$ as the true frequency. And we evaluate the utility using *mean squared error* (MSE), over the randomness in $\mathcal{R}$:

$$\mathrm{MSE}(\bar{\mathbf{P}}^{\mathsf{C}}(\mathcal{R}(T))) = \mathbf{E}[(\bar{\mathbf{P}}^{\mathsf{C}}(\mathcal{R}(T)) - \mathbf{P}^{\mathsf{C}}(T))^2]. \tag{4.1}$$

Optimal Local Hashing

Optimal local hashing (OLH) [157] uses hash function $H : \mathbb{D} \mapsto [g]$ to compress the domain $\mathbb{D}$. Here, H is randomly selected from a pre-defined family of hash functions, whose g is the closest integer to $e^\epsilon + 1$. Given the tuple t, $\mathcal{R}_{\mathsf{OLH}}$ randomly selects H from the family of hash functions, and outputs H, together with either the hash value of t or any distinct hash value, with the following probabilities:

$$\mathcal{R}_{\mathsf{OLH}}(t) = \begin{cases} \langle H, h \leftarrow H(t) \rangle, & \text{w/p } \frac{e^\epsilon}{e^\epsilon + g - 1} \\ \langle H, h \xleftarrow{\$} [g] \setminus \{H(t)\} \rangle, & \text{w/p } \frac{g-1}{e^\epsilon + g - 1} \end{cases}, \tag{4.2}$$

where $\xleftarrow{\$}$ indicates uniform random sampling from the set $[g] \setminus \{H(t)\}$.

On receiving the perturbed value $y = \langle H, h \rangle$, $\bar{\mathbf{P}}_{\mathsf{OLH}}$ estimates its contribution to the frequency of $v \in \mathbb{D}$ as $\bar{\mathbf{P}}^v_{\mathsf{OLH}}(y) = \frac{1_{\{H(v)=h\}} - q}{p - q}$, where $p = \frac{e^\epsilon}{e^\epsilon + g - 1}$ and $q = 1/g$. As stated in [157], answering the frequency for any $v \in \mathbb{D}$ using OLH has the error bound $\mathrm{MSE}(\bar{\mathbf{P}}^v_{\mathsf{OLH}}(\mathcal{R}(T))) = \mathrm{O}\left(\frac{4n}{\epsilon^2}\right)$.

OLH works well for point query, and its error for range query increases fast as the range volume increases. This motivates optimized multi-dimensional frequency oracles for range query.

Hierarchical-Interval Optimized Mechanism

The *hierarchical-interval optimized* (HIO) mechanism [158] divides the domain $\mathbb{D}$ into hierarchy of nodes with various sizes on various layers. In particular, the hierarchy of a single ordinal attribute of cardinality m consists of one interval of the entire attribute domain at the root and m finest-grained intervals, *e.g.*, individual values, at the leaves, with intermediate intervals being the even split of their parents by the fan-out of B. For d ordinal attributes, the d hierarchies are cross-producted into one multi-dimensional hierarchy of nodes.

To perturb tuple t, $\mathcal{R}_{\mathsf{HIO}}$ samples one layer from the hierarchy and maps t to the node that

contain the tuple. The node is then perturbed by $\mathcal{R}_{\mathsf{OLH}}$ among nodes on its layer. To estimate the frequency of report y for range predicate C, $\bar{\mathbf{P}}^{\mathsf{C}}_{\mathsf{HIO}}$ first decomposes C into the set of hierarchical nodes $\mathcal{I}^{\mathsf{C}}$, and estimates the contribution of y to each of the nodes using $\bar{\mathbf{P}}_{\mathsf{OLH}}$, if y and the node are on the same layer. These frequencies, added together and multiplied by $\mathrm{O}\left(\log^d m\right)$, is the unbiased estimation of the frequency of C. As stated in [158], answering the range frequency query C on d_q attributes using HIO has the error bound:

$$\mathrm{MSE}(\bar{\mathbf{P}}^{\mathsf{C}}_{\mathsf{HIO}}(\mathcal{R}(T))) = \mathrm{O}\left(\frac{n \log^{d+d_q} m}{\epsilon^2}\right). \tag{4.3}$$

[158] also achieves frequency oracles with nonsensitive weights for aggregation on what they call *measures* that are not sensitive.

4.3.2 Sensitive-weight Frequency Oracles

Consider a virtual fact table T that is comprised of n users' tuples. A query asks for the aggregation on a sensitive attribute A:

$$\texttt{SELECT}\ \mathsf{SUM}(A)\ \texttt{FROM}\ T\ \texttt{WHERE}\ \mathsf{C}. \tag{4.4}$$

This class of query requires the underlying LDP oracles to weight the frequency contribution of a perturbed value y to C by its attribute value A, and we define the LDP primitive for such query $\mathsf{Q} = (A, \mathsf{C})$ as *sensitive-weight frequency oracle*, denoted as $\bar{\mathbf{P}}^{\mathsf{Q}}(y)$. Thus, query (4.4) can be estimated as $\bar{\mathbf{P}}^{\mathsf{Q}}(\mathcal{R}(T)) = \sum_{y \in \mathcal{R}(T)} \bar{\mathbf{P}}^{\mathsf{Q}}(y)$.

Baseline I: HIO. One baseline to sensitive-weight frequency oracles enumerates all possible values of A and estimates their frequencies using multi-dimensional frequency oracles, *e.g.*, HIO, which are summed up, weighted by values of A, as the aggregation result. In particular, $\bar{\mathbf{P}}^{\mathsf{Q}}_{\mathsf{HIO}}(y) = \sum_{v \in A} v \cdot \bar{\mathbf{P}}^{\mathsf{C} \wedge (A=v)}_{\mathsf{HIO}}(y)$. According to the error bound of HIO (Equation (4.3)), the error bound of this baseline is: $\mathrm{MSE}(\bar{\mathbf{P}}^{\mathsf{Q}}_{\mathsf{HIO}}(\mathcal{R}(T))) = \mathrm{O}\left(\sum_{v \in A} v^2 \frac{n \log^{d+d_q} m}{\epsilon^2}\right)$.

Baseline II: HIO with *stochastic rounding* (SR-HIO). For improved utility, we can combine *stochastic rounding* (SR) [47] with HIO to enumerate only the two extreme domain values at aggregation. In particular, for each attribute A, $\mathcal{R}_{\mathsf{SR\text{-}HIO}}$ first rounds $t[A]$, *i.e.*, the value of A in t, to either the min $A_{\min}$ or the max $A_{\max}$, using the *stochastic rounding* function $\mathcal{M}^{A}_{\mathsf{SR}}$ defined as follows:

$$\mathcal{M}^{A}_{\mathsf{SR}}(t[A]) = \begin{cases} A_{\min}, & \text{w/p} \quad \frac{A_{\max}-t[A]}{A_{\max}-A_{\min}} \\ A_{\max}, & \text{w/p} \quad \frac{t[A]-A_{\min}}{A_{\max}-A_{\min}} \end{cases} \tag{4.5}$$

Then, it perturbs the d original attributes in t, together with the d rounding values, using $\mathcal{R}_{\mathsf{HIO}}$ for ϵ-LDP. Note that perturbing the d original attribute values enables arbitrary range aggregation on the collected tuples while perturbing the d rounding values enables the attribute aggregation on any of the d attribute, both of which are necessary to support our target query class.

Given a perturbed value y, we can answer the frequency oracle for predicate C simply as $\bar{\mathbf{P}}^{\mathsf{C}}_{\mathsf{SR\text{-}HIO}}(y) = \bar{\mathbf{P}}^{\mathsf{C}}_{\mathsf{HIO}}(y)$, and, to answer query (4.4), we estimate the result by linearly combining the frequencies of the two extreme values, *i.e.*,

$$\bar{\mathbf{P}}^{\mathsf{Q}}_{\mathsf{SR\text{-}HIO}}(y) = A_{\min} \cdot \bar{\mathbf{P}}^{\mathsf{C}\wedge A_{\min}}_{\mathsf{HIO}}(y) + A_{\max} \cdot \bar{\mathbf{P}}^{\mathsf{C}\wedge A_{\max}}_{\mathsf{HIO}}(y).$$

The error bound of such an estimation for query (4.4) is

$$\text{MSE}(\bar{\mathbf{P}}^{\mathsf{Q}}_{\mathsf{SR\text{-}HIO}}(\mathcal{R}(T))) = \text{O}\left(\frac{(A_{\min}^2 + A_{\max}^2) 2^d n \log^{d+d_q} m}{\epsilon^2}\right), \tag{4.6}$$

because SR-HIO introduces d extra attributes from rounding, each of which is of cardinality 2, and increases Equation (4.3) by multiplicative factor of 2^d. Next, we propose the *partition-rounding-perturb* framework to eliminate the factor of 2^d in the error bound.

4.3.3 Partition-Rounding-Perturb Framework

The improvement of SR-HIO comes from the fact that only one attribute can appear in the aggregation function F. To leverage such fact, we first randomly *partition* the tuples into d groups, one for each of the d attributes, using a randomized partition function $\mathcal{S}$, *i.e.*, $\mathcal{S}(t) = t[A],$ for $A \xleftarrow{\$} \{A_1, \cdots, A_d\}$, where $\xleftarrow{\$}$ indicates uniform random sampling from the d attributes; then apply *rounding* function $\mathcal{M}_{\mathsf{SR}}^{A}$ on the attribute from $\mathcal{S}$ for each tuple independently; finally, for each tuple, we *perturb* its d attribute values, together with the single rounding value. Now to aggregate on attribute A, we only use tuples partitioned for A, aggregate their estimated contributions, and scale the result by d as the final result.

We call the overall framework *partition-rounding-perturb* (PRP), and, next, we introduce two instantiations using different ways to perturb the extra rounding value, with different error bounds.

Augment-then-Perturb (AHIO)

AHIO augments the tuple t with the extra attribute X_A for the rounding value of attribute A, and $\mathcal{R}_{\mathsf{AHIO}}$ perturbs t, including the value of X_A, using HIO. Overall, we have:

$$\mathcal{R}_{\mathsf{AHIO}}(t) = \mathcal{R}_{\mathsf{HIO}}(\langle t, X_A = \mathcal{M}_{\mathsf{SR}}^{A}(\mathcal{S}(t))\rangle). \tag{4.7}$$

At aggregation, $\bar{\mathbf{P}}_{\mathsf{AHIO}}$ estimates the frequencies of tuples that satisfy C and have certain rounding value by patching C with conjunctive equality condition for the rounding value. For instance, to estimate the frequency of tuples that satisfy C and have attribute A rounded to $A_{\min}$, $\bar{\mathbf{P}}_{\mathsf{AHIO}}$ answers the multi-dimensional frequency using $\bar{\mathbf{P}}_{\mathsf{HIO}}$ on condition $\mathsf{C} \wedge X_A = A_{\min}$. Then it adds the frequencies, multiplied by the corresponding rounding values and d, to answer the sensitive-

weight frequency oracles, *i.e.*, :

$$\bar{\mathbf{P}}^{\mathsf{Q}}_{\mathsf{AHIO}}(y) = \begin{cases} 0, & \text{if } y \text{ is not in the group for } A \\ d(A_{\min} \cdot \bar{\mathbf{P}}^{\mathsf{C}\wedge(X_A=A_{\min})}_{\mathsf{HIO}}(y) & \\ +A_{\max} \cdot \bar{\mathbf{P}}^{\mathsf{C}\wedge(X_A=A_{\max})}_{\mathsf{HIO}}(y)), & \text{otherwise} \end{cases} \tag{4.8}$$

Figure 4.3 shows how AHIO works using a toy example with two attributes and an aggregation query. The error bound of AHIO is:

Lemma 5. *Answering query* (4.4) *using* AHIO *under ϵ-LDP has the error bound of:*

$$\mathrm{MSE}(\bar{\mathbf{P}}^{\mathsf{Q}}_{\mathsf{AHIO}}(\mathcal{R}(T))) = \mathrm{O}\left(\frac{2(A^2_{\min}+A^2_{\max})n\cdot d\log^{d+d_q} m}{\epsilon^2}\right). \tag{4.9}$$

Proof Sketch: First, since the n tuples is partitioned into d groups, and the augmented attribute introduces an hierarchy of 2 layers, using HIO with ϵ-LDP to estimate the frequencies for $\mathsf{C} \wedge (X_A = A_{\min})$ and $\mathsf{C} \wedge (X_A = A_{\max})$ has error bound of $\mathrm{O}\left(\frac{2n\log^{d+d_q} m}{d\epsilon^2}\right)$. Then, scaling the frequencies by $A_{\min}$ and $A_{\max}$, and multiplying the weighted frequency sum by d, introduces multiplicative factors $(A^2_{\min} + A^2_{\max})$ and d^2 to the error. □

Embed-then-Perturb (EHIO)

EHIO embeds the rounding value into the partition attribute by doubling its domain. Specifically, it doubles the domain of A to $A^-(= [2A_{\min} - A_{\max} - 1, A_{\min} - 1]) \vee A^+(= [A_{\min}, A_{\max}])$. If the rounding value is $A_{\min}$, $\mathcal{R}_{\mathsf{EHIO}}$ maps $t[A]$ to $2A_{\min} - t[A] - 1 \in A^-$; otherwise, the value is unchanged, and in A^+. And the mapped tuple is perturbed by $\mathcal{R}_{\mathsf{HIO}}$.

To aggregate on A, $\bar{\mathbf{P}}_{\mathsf{EHIO}}$ patches range conditions, *i.e.*, $A \in A^-$ for $A_{\min}$ and $A \in A^+$ for

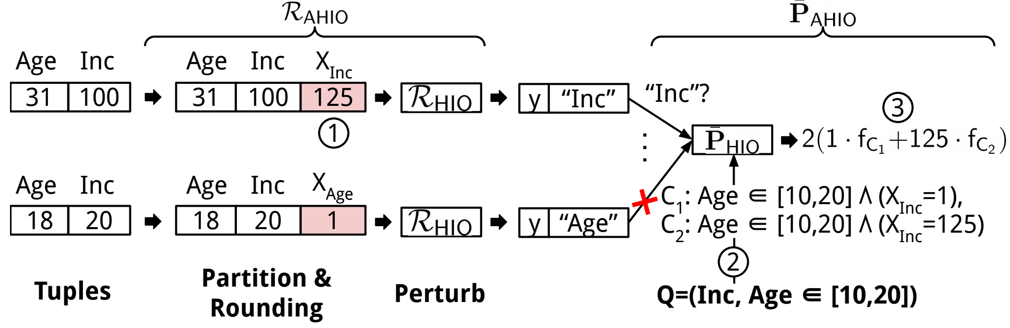


Figure 4.3: Example of running AHIO on tuples with attributes Age $\in [1, 125]$ and Inc $\in [1, 125]$ (income), and aggregation on Inc for users with Age $\in [10, 20]$. ① partition the tuples by attribute, round their Inc or Age value, and augment with attribute X_{Inc} or X_{Age} for the rounding value; ② rewrite the query into two frequency aggregations for the min and max of Inc; ③ combine the frequency estimations, weighted by the min and max of Inc, and scale the result by 2 to compensate for the attribute partitioning.

$A_{\max}$, to the predicate C to estimate the frequencies of the two rounding values:

$$\bar{\mathbf{P}}^{\mathsf{Q}}_{\mathsf{EHIO}}(y) = \begin{cases} 0, & \text{if y is not partitioned for A} \\ d(A_{\min} \cdot \bar{\mathbf{P}}^{\mathsf{C} \wedge (A \in A^-)}_{\mathsf{HIO}}(y) & \\ +A_{\max} \cdot \bar{\mathbf{P}}^{\mathsf{C} \wedge (A \in A^+)}_{\mathsf{HIO}}(y)), \text{otherwise} & \end{cases} \tag{4.10}$$

If C contains range condition $A \in [l, r]$, EHIO maps it to conditions $A \in [2A_{\min} - l - 1, 2A_{\min} - r - 1]$ and $A \in [l, r]$ for $A_{\min}$ and $A_{\max}$, respectively. EHIO doubles the domain size of attribute A from m to $2m$. Thus, its error bound for query (4.4) is:

$$\text{MSE}(\bar{\mathbf{P}}^{\mathsf{Q}}_{\mathsf{EHIO}}(\mathcal{R}(T))) = \text{O}\left(\frac{(A_{\min}^2 + A_{\max}^2) nd \log^2 2m \log^{d+d_q-2} m}{\epsilon^2}\right). \tag{4.11}$$

Remarks. The difference between the error bounds of AHIO and EHIO is $\frac{2\log^2 m}{\log^2 2m}$, which is between 0.5 (when $m = 2$) and 2 (when $m \to \infty$). It is better to use AHIO and EHIO when m is small and large, respectively.

4.4 1-to-1 Joint Frequency Oracles

In this section, we focus on the joint aggregation over tuples that are collected by K separate services:

$$\texttt{SELECT}\ \mathsf{F}(A)\ \texttt{FROM}\ T_1\ \texttt{JOIN}\ \ldots\ \texttt{JOIN}\ T_K\ \texttt{WHERE}\ \mathsf{C}, \tag{4.12}$$

where A is the aggregation attribute collected by one of the joining services, and C consists of $K \cdot d_q$ conditions, on d_q attributes of each service. The key task here is to, given the vector of perturbed values $\mathbf{y} = \langle y_i \rangle_{i=1}^{K}$ joined on the same user id, estimate its contribution to the frequency of C, and we call such LDP primitive *joint frequency oracles*, denoted as $\bar{\mathbf{P}}^{\mathsf{C}}(\mathbf{y})$. Given such frequency oracles, we can plug it into the partition-rounding-perturb framework, in place of HIO, to achieve attribute aggregation, *i.e.*, $\bar{\mathbf{P}}^{\mathsf{Q}}(\mathbf{y})$, for $\mathsf{Q} = (A, \mathsf{C})$, *i.e.*, $\bar{\mathbf{P}}^{\mathsf{Q}}(\bowtie_{i=1}^{K} \mathcal{R}(T_i)) = \sum_{\mathbf{y} \in \bowtie_{j=i}^{K} \mathcal{R}(T_i)} \bar{\mathbf{P}}^{\mathsf{Q}}(\mathbf{y})$.

Next, we focus on joint frequency oracles over tuples with one-to-one relation, and extend to one-to-many relation in Section 4.5.

4.4.1 Split-and-Conjunction Baseline

Split-and-conjunction (SC) in [158] enables single-table multi-dimensional frequency oracles over tuples with independently perturbed attributes. In our setting, we can use SC to independently perturb all attributes from the K services, each with privacy parameter $\frac{\epsilon}{K \cdot d}$, and estimate the frequency of C using the reported perturbed attribute values. We call such mechanism SC-JOIN.

To estimate the frequency for C, SC-JOIN first decomposes C into $K \cdot d_q$ atomic predicates $\mathsf{C}_1, \mathsf{C}_2, \ldots$, one for each attribute in C. For the joined value $\mathbf{y}$, SC-JOIN increments $\mathbf{f}'_{\mathsf{C}}[x]$ by one, where the i-th bit of x is 1 if $\mathbf{y}$ satisfies C_i; or 0 otherwise. Similarly, we can denote the vector $\mathbf{f}_{\mathsf{C}}$ as the vector of frequencies on the original tuples, and $\mathbf{f}_{\mathsf{C}}[11\ldots1]$, *i.e.*, 1 for all C_i in C, is the true frequency aggregation for C. The two vectors are connected via a state transition matrix $\mathbf{M}$, which defines the stochastic transition from the state $\mathbf{f}_{\mathsf{C}}$ to the frequency distribution $\mathbf{f}'_{\mathsf{C}}$ *i.e.*,

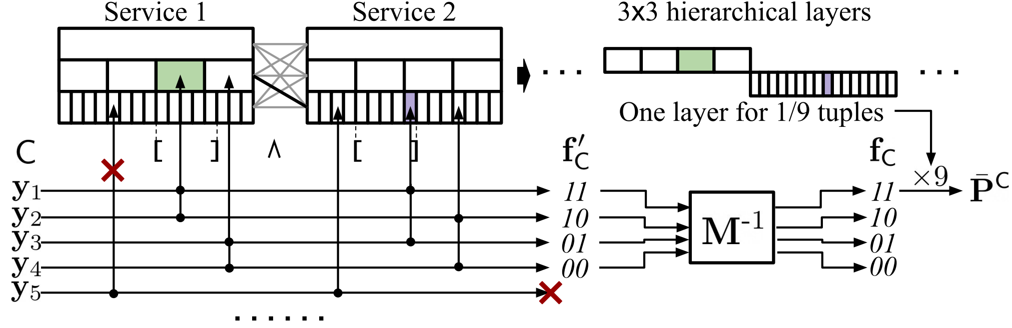


Figure 4.4: HIO-JOIN across two services, each collecting one attribute using hierarchy with fan-out $B = 4$.

$\mathbf{E}[\mathbf{f}'_{\mathsf{C}}] = \mathbf{M} \cdot \mathbf{f}_{\mathsf{C}}$. For instance, when C contains only one condition, we have

$$\mathbf{E}\left[\begin{pmatrix} \mathbf{f}'_{\mathsf{C}}[0] \\ \mathbf{f}'_{\mathsf{C}}[1] \end{pmatrix}\right] = \begin{pmatrix} \Pr[0|0] & \Pr[0|1] \\ \Pr[1|0] & \Pr[1|1] \end{pmatrix} \begin{pmatrix} \mathbf{f}_{\mathsf{C}}[0] \\ \mathbf{f}_{\mathsf{C}}[1] \end{pmatrix}, \tag{4.13}$$

where $\Pr[x'|x]$ indicates the probabilities of the perturbed value evaluated as true/false ($x' = 1/0$), given the original value evaluated as true/false ($x = 1/0$). Generally, for C with conjunctive conditions on $K \cdot d_q$ attributes, $\mathbf{M}$ is of dimension $2^{K \cdot d_q} \times 2^{K \cdot d_q}$, and we can derive $\mathbf{M}$ based on the probabilities p and q of the LDP mechanism (See [158] for details). With $\mathbf{f}'_{\mathsf{C}}$ and $\mathbf{M}$, we can calculate $\mathbf{M}^{-1} \cdot \mathbf{f}'_{\mathsf{C}}$ as the unbiased estimation of the initial state $\mathbf{f}_{\mathsf{C}}$, which contains the frequency for the given predicate. We call such technique *state inversion*, and we have $\bar{\mathbf{P}}^{\mathsf{C}}_{\mathsf{SC\text{-}JOIN}}(\mathbf{y}) = (\mathbf{M}^{-1} \cdot \mathbf{f}'_{\mathbf{y},\mathsf{C}})[11\ldots1]$, where $\mathbf{f}'_{\mathbf{y},\mathsf{C}}$ is the vector $\mathbf{f}'_{\mathsf{C}}$ on $\mathbf{y}$.

As stated in [158], for $\mathsf{Q} = (A, \mathsf{C})$ on $T = \bowtie_{i=1}^{K} T_i$, SC-JOIN has error bound of:

$$\mathrm{MSE}(\bar{\mathbf{P}}^{\mathsf{C}}_{\mathsf{SC\text{-}JOIN}}(\mathcal{R}(T))) = \mathrm{O}\left(\frac{n \log^{3K \cdot d_q} m}{(\epsilon/(K \cdot d))^{2K \cdot d_q}}\right). \tag{4.14}$$

To answer the joint frequency of C, SC-JOIN requires joint estimation on all attributes in C, whether from the same or different services. For better utility, we want to minimize the number of joint estimations to the number of services involved in C, *i.e.*, up to K.

4.4.2 Multi-Service Joint Frequency Oracles

While supporting joint analysis, SC-JOIN independently perturbs all the attributes. This makes single-table analysis more noisy as SC is used for each service, instead of HIO. To improve the utility for single-service analysis and at the same time support joint analysis, we propose to adapt the state inversion technique in SC-JOIN for HIO reports, and call such mechanism HIO-JOIN.

For each tuple of a service, $\mathcal{R}_{\text{HIO-JOIN}}$ applies $\mathcal{R}_{\text{HIO}}$ to perturb it. Formally, for user u with tuples $T_1^u, \ldots, T_K^u$ across the K services,

$$\mathcal{R}_{\text{HIO-JOIN}}(T_1^u, \ldots, T_K^u) = \mathcal{R}_{\text{HIO}}^{\epsilon/K}(T_1^u), \ldots, \mathcal{R}_{\text{HIO}}^{\epsilon/K}(T_K^u) \tag{4.15}$$

is reported, where $\mathcal{R}_{\text{HIO}}^{\epsilon/K}(T_i^u)$ perturbs the single tuple in T_i^u using HIO with privacy parameter $\frac{\epsilon}{K}$. Here, tuples of the K services are independently perturbed by $\mathcal{R}_{\text{HIO}}$, and the privacy parameter ϵ is evenly divided for all the K tuples of the user so that the overall privacy loss is bounded by ϵ. Next, we focus on the joint aggregation over these perturbed HIO reports.

We adapt the state inversion technique in SC-JOIN for HIO-JOIN with the key insight that, after mapping a tuple to the node in the hierarchy, $\mathcal{R}_{\text{HIO}}$ perturbs the node in the exact same manner as SC perturbs the single attribute. Thus, for each hierarchical layer across the K services, we can derive the state transition matrix for the nodes on that layer across the K services, following what SC does for nodes across the $K \cdot d_q$ attributes, which gives us a state transition matrix of dimension $2^K \times 2^K$. The joined tuples that are sampled by $\mathcal{R}_{\text{HIO}}$ on the same layers as the predicate decompositions are classified into the vector $\mathbf{f}'_{\text{C}}$, which is then multiplied by $\mathbf{M}^{-1}$ to recover the frequency of the original tuples. The estimated frequencies of all the decompositions are added together, and scaled by the number of layers across K services, *i.e.*, $\log^{K \cdot d} m$, to compensate for the layer sampling in HIO.

Thus, we have the frequency estimation of $\mathsf{HIO\text{-}JOIN}$ as:

$$\bar{\mathbf{P}}^{\mathsf{C}}_{\mathsf{HIO\text{-}JOIN}}(\mathbf{y}) = \begin{cases} 0, & \text{if } \mathbf{y} \text{ and } \mathsf{C} \text{ on different layers} \\ \log^{K\cdot d} m \cdot (\mathbf{M}^{-1} \cdot \mathbf{f}'_{\mathbf{y},\mathsf{C}})[11\ldots 1], & \text{otherwise} \end{cases} \tag{4.16}$$

Figure 4.4 shows how HIO-JOIN works with two services, each collecting one attribute: at collection, each tuple of a service is perturbed on one the 3 layers of the hierarchy of the attribute; at aggregation, the predicate C with range conditions on attributes of the two services is first decomposed into pairs of nodes across the two hierarchies. For one pair of nodes, *i.e.*, the green and purple ones, HIO-JOIN calculates $\mathbf{f}'_{\mathsf{C}}$ by counting the numbers for indices 11, 10, 01 and 00, which represent the four cases where the joined tuple $\mathbf{y}_i$ has part from service 1 (not) in the green node and part from service 2 (not) in the purple node. For instance, $\mathbf{y}_1$ has its two attributes inside the green and purple nodes, which increments $\mathbf{f}'_{\mathsf{C}}[11]$ by 1. And $\mathbf{y}_2$ has its first attribute inside the green node, and the second attribute outside the purple node, which increments $\mathbf{f}'_{\mathsf{C}}[10]$ by 1. Joined values not on the same layers as the decomposed nodes, *e.g.*, $\mathbf{y}_5$, do not contribute to any of the classes. Since there are 3×3 layers across the two hierarchies, $\frac{1}{9}$ of the collected tuples are expected to be on the same layer as the decomposed nodes. Thus, after multiplying $\mathbf{f}'_{\mathsf{C}}$ by $\mathbf{M}^{-1}$, HIO-JOIN multiplies the estimation by 9. HIO-JOIN repeats the estimation for the other decompositions of predicate C, and adds their estimations together as the aggregation for C.

We state the error bound of HIO-JOIN in Lemma 6:

Lemma 6. *Answering the frequency query for predicate* C *on* $K \cdot d_q$ *attributes against* $T = \bowtie_{i=1}^{K} T_i$ *using* HIO-JOIN *has error bound:*

$$\mathrm{MSE}(\bar{\mathbf{P}}^{\mathsf{C}}_{\mathsf{HIO\text{-}JOIN}}(\mathcal{R}(T))) = \mathrm{O}\left(\frac{n \log^{K(d+d_q)} m}{(\epsilon/K)^{2K}}\right). \tag{4.17}$$

Proof Sketch: First, the number of joined values sampled on the same layer as C is $\mathrm{O}\left(\frac{n}{\log^{K\cdot d} m}\right)$. For these joined values, since the parameter ϵ is only splitted by K, estimating the range fre-

quency on each of the $\mathrm{O}\left(\log^{K\cdot d_q} m\right)$ hierarchy nodes, using model inversion, incurs error bound of $\mathrm{O}\left(\frac{n}{\log^{K\cdot d} m(\epsilon/K)^{2K}}\right)$. Finally, we multiply the aggregations by factor of $\mathrm{O}\left(\log^{K\cdot d} m\right)$. □

Note that here the effect of joint estimation, which is manifested as the power of ϵ, is at the scale of K, which is smaller than the $K \cdot d_q$ in SC-JOIN. The difference will be considerable when d_q is large and ϵ is small, which we evaluate in Section 4.7.3

Security. HIO-JOIN splits ϵ by K, and, by sequential composition, the overall privacy guarantees for each user is ϵ-LDP. Furthermore, since users have the same number of tuples collected by each service, *i.e.*, 1, it is ϵ-uLDP. This argument, however, does not generalize to the one-to-many relation, where users of the i-th service can generate arbitrary numbers of tuples. We will address this problem in Section 4.5.

4.5 Handling One-to-Many Join

In this section, we focus on the joint aggregation over tuples with one-to-many relation. In particular, we focus on the primary-foreign-key joint aggregation with two services. Each user u generates exactly one tuple with user id as the primary key in service 1, *i.e.*, $\forall u : |T_1^u| = 1$, and up to $\tau_{\max}$ tuples with user id as a foreign key in service 2, *i.e.*, $\forall u : 0 \leq |T_2^u| \leq \tau_{\max}$. We assume that $\tau_{\max}$, the maximum number of tuples that can be generated by a user in service 2 is public. We want to guarantee ϵ-uLDP (Definition 4), and the target analytical task is the same as query (4.12).

4.5.1 Frequency-based Attack

We introduced SC-JOIN and HIO-JOIN for joint aggregation under one-to-one relation. A straightforward extension to one-to-many relation is to split the privacy parameter ϵ for a user u among all the tuples collected. That is, each tuple in services 1 and 2 is reported under $\epsilon/(|T_1^u| + |T_2^u|)$-LDP. And the overall privacy guarantees for u is ϵ-LDP, based on the sequential composition.

Unfortunately, this approach is not private, as it enables the simple yet effective *frequency-*

based attack: After collecting perturbed tuples from service 2, for each user, the adversary can count the number (*frequency*) of tuples that can be joined with some tuple from service 1 on user id, by grouping the tuples by user id. With such frequency information available, one can immediately distinguish between users with different usage patterns in service 2. With extra prior knowledge, based on these frequencies, the adversary can launch even more devastating attacks to infer sensitive attributes about users (see Example 4.1.1). In fact, the presence of some reported tuples or absence of any tuple already reveals information about whether the user is using a service, which by itself is sensitive especially when the service targets certain group of users.

More formally, we can show that the above approach does not provide uLDP (Definition 4) to individual users. To show this for HIO-JOIN and two users u and u', *s.t.*, $|T_2^u| \neq |T_2^{u'}|$, we take $\mathbf{y} = \mathcal{R}_{\text{HIO-JOIN}}(T^u)$ in Definition 4, and, thus, $|\mathbf{y}| = |\mathcal{R}_{\text{HIO-JOIN}}(T^u)| \neq |\mathcal{R}_{\text{HIO-JOIN}}(T^{u'})|$. Thus, the ratio between the probabilities of the perturbed tuples from u and u' being equal to $\mathbf{y}$ are unbounded (∞-uLDP), as $\mathbf{Pr}[\mathcal{R}(T^u) = \mathbf{y}] > 0,$ and $\mathbf{Pr}[\mathcal{R}(T^{u'}) = \mathbf{y}] = 0.$

The problem of the above approach is that it outputs a perturbed value for each input tuple of the user, and breaks the privacy guarantees when users have different numbers of tuples. Next, we close such security loophole by hiding the real number of tuples with user id as the foreign-key, *i.e.*, from service 2, of a user, and achieve ϵ-uLDP under one-to-many relation.

4.5.2 Hiding Existence and Frequency with τ-Truncation

To prevent the frequency-based attacks that exploit the distinct numbers of collected tuples on service 2 among users, we propose the τ-truncation mechanism that outputs the fixed number, *i.e.*, τ, of perturbed tuples for each user's tuples on service 2.

Suppose each user u generates a maximum of $\tau_{\max}$ tuples in service 2, *i.e.*, $0 \leq |T_2^u| \leq \tau_{\max}$. Here, $|T_2^u| = 0$ means that u is not using this service. The goal of τ-truncation is to hide each user's presence/absence information as well as the true frequency.

If $|T_2^u| > 0$, we want to hide the frequency $|T_2^u|$. The mechanism samples τ tuples from

T_2^u with replacement, and attach a *truncation weight* (inversed sampling ratio) $r = |T_2^u|/\tau \in [r_{\min}, r_{\max}]$, where $r_{\min} = 0$ and $r_{\max} = \tau_{\max}/\tau$, to each of these tuples. This weight is used to obtain an unbiased estimate of the aggregation.

If $|T_2^u| = 0$, we want to hide the absence of u. The mechanism randomly draws a sample of τ tuples from the domain $\mathbb{D}_2$ of T_2, and attach a weight $r = 0$ to each of them (so that we know these dummy tuples have no contribution in any aggregation).

Formally, the τ-truncation procedure Trunc is defined to be:

$$\mathsf{Trunc}(\tau, T) = \begin{cases} \langle t_i \xleftarrow{\$} T, r = \frac{|T|}{\tau} \rangle_{i=1}^{\tau} & \text{if } |T| > 0 \\ \langle t_i \xleftarrow{\$} \mathbb{D}, r = 0 \rangle_{i=1}^{\tau} & \text{if } |T| = 0 \end{cases}, \tag{4.18}$$

where $t_i \xleftarrow{\$} T$ (or the domain $\mathbb{D}$ of T) means that we use uniform random sampling to draw a tuple from T (or $\mathbb{D}$) as t_i.

In both cases, each user generates exactly τ tuples $\langle t_i, r \rangle_{i=1}^{\tau}$. In order to completely hide the existence and frequency information as well as the content in t_i, we apply an LDP perturbation mechanism, *e.g.*, $\mathcal{R}_{\mathsf{AHIO}}$ in Section 4.3.3, on both the value of r and t_i.

In terms of service 1, since user id is the primary key, each user generates exactly one tuple in T_1^u, and thus, we only need to apply the perturbation mechanism on this tuple.

For each user u, there are a total of $\tau + 1$ tuples to be perturbed, namely, one tuple in T_1^u and τ tuples in $\mathsf{Trunc}(\tau, T_2^u) = \langle t_i, r \rangle_{i=1}^{\tau}$. The privacy budget is evenly partitioned among these $\tau + 1$ tuples. Putting them together, we are going to release

$$\mathcal{R}_{\mathsf{Trunc}}^{\epsilon}(T_1^u, T_2^u) = \mathcal{R}_{\mathsf{AHIO}}^{\epsilon/(\tau+1)}(T_1^u) \oplus \langle \mathcal{R}_{\mathsf{AHIO}}^{\epsilon/(\tau+1)}(\langle t_i, r \rangle) \rangle_{i=1}^{\tau} \tag{4.19}$$

where each instance of $\mathcal{R}_{\mathsf{AHIO}}^{\epsilon/(\tau+1)}$ with privacy parameter $\frac{\epsilon}{\tau+1}$ runs independently. We can show that $\mathcal{R}_{\mathsf{Trunc}}^{\epsilon}$ is ϵ-uLDP.

Lemma 7. *Collecting a user's tuples from service 1 (with user id as the primary key) and service*

2 (with user id as a foreign key) using $\mathcal{R}^{\epsilon}_{\mathsf{Trunc}}$ above satisfies ϵ-uLDP.

Proof Sketch: For users u and u', they both have one tuple on service 1, and $|T_2^u|$ and $|T_2^{u'}|$ tuples on service 2, respectively. With τ-truncation, both users have τ tuples collected by service 2. In addition, all collected tuples of a user is perturbed with $\frac{\epsilon}{1+\tau}$-LDP. In the definition of uLDP, any possible value $\mathbf{y}$ from τ-truncation consists of one perturbed tuple for service 1 and τ perturbed tuples for service 2. Thus, we have $\frac{\mathbf{Pr}[\mathcal{R}(T^u)=\mathbf{y}]}{\mathbf{Pr}\left[\mathcal{R}(T^{u'})=\mathbf{y}\right]} \leq (e^{\epsilon/(1+\tau)})^{1+\tau}$. □

Both the aggregating attribute A and the truncation weight r are randomly rounded (to $\{A_{\min}, A_{\max}\}$ and $\{r_{\min}, r_{\max}\}$, respectively) during the perturbation process $\mathcal{R}_{\mathsf{AHIO}}$. In the next subsection, we will introduce our *double-rounding mechanism and estimation* technique to recover the answer to the original aggregation query from truncated and perturbed data in an unbiased way.

4.5.3 Double Rounding: Recovering Aggregation from Truncated Tuples

At aggregation, the contribution of each perturbed value need to be multiplied by its truncation weight to compensate for tuples truncated away by Trunc. In particular, we need to answer the following queries for $\mathsf{COUNT}(*)$ and $\mathsf{SUM}(A)$, respectively:

$$\texttt{SELECT}\ \mathsf{SUM}(r)\ \texttt{FROM}\ T_1\ \texttt{JOIN}\ T_2\ \texttt{WHERE}\ \mathsf{C},\ \text{and} \tag{4.20}$$

$$\texttt{SELECT}\ \mathsf{SUM}(r \cdot A)\ \texttt{FROM}\ T_1\ \texttt{JOIN}\ T_2\ \texttt{WHERE}\ \mathsf{C}, \tag{4.21}$$

where r is the truncation weight in the collected tuples. Queries (4.20) and (4.21) require two sensitive-weight frequency oracles, *i.e.*, one for $\mathsf{Q} = (r, \mathsf{C})$ and the other for $\mathsf{Q} = (r \cdot A, \mathsf{C})$, and directly applying the mechanisms in Section 4.3.3 would degrade the utility for the frequency aggregation (query (4.20)) by $\mathrm{O}(d+1)$, due to partitioning.

To preserve the utility for frequency aggregation, we do not partition the tuples for frequency aggregation, and only partition the tuples among the d attributes for attribute aggregation, via what we call *double rounding*: for each truncated tuple output by Trunc, we group it for one of the

the d attributes, and derive two rounding values, one for the partition attribute A, and the other for the truncation weight r; thus, the rounding value for r enables query (4.20) for all tuples, *i.e.*, $\bar{\mathbf{P}}^{(r,\mathsf{C})}_{\mathsf{Trunc}}(\mathbf{y}) = r_{\max} \cdot \bar{\mathbf{P}}^{\mathsf{C}\wedge r_{\max}}_{\mathsf{HIO\text{-}JOIN}}(\mathbf{y}) + r_{\min} \cdot \bar{\mathbf{P}}^{\mathsf{C}\wedge r_{\min}}_{\mathsf{HIO\text{-}JOIN}}(\mathbf{y})$; Note that, since $r_{\min} = 0$, the contributions for predicate $\mathsf{C} \wedge r_{\min}$ is zero, and, thus, ignored; similarly, the rounding values for A and r together enable query (4.21) for tuples partitioned for A, and, for one such $\mathbf{y}$, we have $\bar{\mathbf{P}}^{(r\cdot A,\mathsf{C})}_{\mathsf{Trunc}}(\mathbf{y}) = d \cdot \sum_{b_r} \sum_{b_A} b_r \cdot b_A \cdot \bar{\mathbf{P}}^{\mathsf{C}\wedge b_r \wedge b_A}_{\mathsf{HIO\text{-}JOIN}}(\mathbf{y})$, where $b_r \in \{r_{\min}, r_{\max}\}$ and $b_A \in \{A_{\min}, A_{\max}\}$.

Concretely, we pack the two rounding values as a pair, and augment the tuple with attribute $X_{r,A} \in \{\langle r_{\min}, A_{\min}\rangle, \langle r_{\min}, A_{\max}\rangle, \langle r_{\max}, A_{\min}\rangle, \langle r_{\max}, A_{\max}\rangle\}$ for it. The augmented attribute is perturbed, together with the original d attributes, for data collection. For $\bar{\mathbf{P}}^{(r\cdot A,\mathsf{C})}_{\mathsf{Trunc}}(\mathbf{y})$, if $\mathbf{y}$ is partitioned for A, we estimate

$$\bar{\mathbf{P}}^{\mathsf{C}\wedge r_{\min}\wedge A_{\min}}_{\mathsf{HIO\text{-}JOIN}}(\mathbf{y}) = \bar{\mathbf{P}}^{\mathsf{C}\wedge (X_{r,A} = \langle r_{\min}, A_{\min}\rangle)}_{\mathsf{HIO\text{-}JOIN}}(\mathbf{y}), \tag{4.22}$$

as well as for other pairs: $\langle r_{\min}, A_{\max}\rangle$, $\langle r_{\max}, A_{\min}\rangle$ and $\langle r_{\max}, A_{\max}\rangle$; otherwise, $\bar{\mathbf{P}}^{(r\cdot A,\mathsf{C})}_{\mathsf{HIO\text{-}JOIN}}(\mathbf{y}) = 0$. And we have:

Lemma 8. $\bar{\mathbf{P}}^{(r\cdot A,\mathsf{C})}_{\mathsf{Trunc}}(\mathcal{R}_{\mathsf{Trunc}}(T_1, T_2))$ *is unbiased for query* (4.21).

Proof Sketch: First, since $\bar{\mathbf{P}}_{\mathsf{HIO\text{-}JOIN}}$ is unbiased, and r and A are independently rounded, we have

$$\mathbf{E}[\bar{\mathbf{P}}^{\mathsf{C}\wedge (X_{r,A} = \langle r_{\min}, A_{\min}\rangle)}_{\mathsf{HIO\text{-}JOIN}}(\mathbf{y})] = \frac{1_{\{\mathbf{t}\in\mathsf{C}\}} \cdot (r_{\max} - \mathbf{t}[r])(A_{\max} - \mathbf{t}[A])}{d(r_{\max} - r_{\min})(A_{\max} - A_{\min})}. \tag{4.23}$$

And we have similar results for $\langle r_{\min}, A_{\max}\rangle$, $\langle r_{\max}, A_{\min}\rangle$ and $\langle r_{\max}, A_{\max}\rangle$. Hence, we have
$\mathbf{E}[\bar{\mathbf{P}}^{(r\cdot A,\mathsf{C})}_{\mathsf{Trunc}}(\mathcal{R}_{\mathsf{Trunc}}(\langle T_1, T_2\rangle))]$
$= \sum_{\mathbf{t}\in T_1 \bowtie T_2} \frac{d \cdot 1_{\{\mathbf{t}\in\mathsf{C}\}} \cdot \mathbf{t}[r] \cdot \mathbf{t}[A] \cdot (r_{\max} - r_{\min})(A_{\max} - A_{\min})}{d \cdot (r_{\max} - r_{\min})(A_{\max} - A_{\min})}$
$= \sum_{\mathbf{t}\in T_1 \bowtie T_2} 1_{\{\mathbf{t}\in\mathsf{C}\}} \cdot \mathbf{t}[r] \cdot \mathbf{t}[A]$. □

Lemma 9. *For n users, whose tuples on service 1 and 2 are collected via $\mathcal{R}_{\textsf{Trunc}}$, answering query* (4.21) *using* $\bar{\mathbf{P}}_{\textsf{Trunc}}$ *has error bound:*

$$\mathrm{O}\left(\frac{nd\tau(1+\tau)^4\log^{2(d+d_q)}m}{\epsilon^4}(r_{\min}^2+r_{\max}^2)(A_{\min}^2+A_{\max}^2)\right). \tag{4.24}$$

Proof Sketch: First, since the ϵ is splitted evenly for the $1+\tau$ tuples of a user, the tuples are each perturbed with $\frac{\epsilon}{1+\tau}$-LDP. In addition, we have $r_{\min}=0, r_{\max}=\tau_{\max}/\tau$. Thus, estimating the contribution of one pair of joined tuples from the two services to the aggregation on A, using $\bar{\mathbf{P}}^{(r\cdot A,\mathsf{C})}_{\textsf{HIO-JOIN}}(\mathbf{y})$, has error bound

$$\mathrm{O}\left(\frac{d(1+\tau)^4\log^{2(d+d_q)}m}{\epsilon^4}(r_{\min}^2+r_{\max}^2)(A_{\max}^2+A_{\min}^2)\right). \tag{4.25}$$

We need to add up the contributions from $n\cdot\tau$ pairs of joined tuples, which sums up the error as Equation (4.24). □

Based on Lemma 9, setting $\tau=1$ achieves the optimal error bound. We evaluate the effects of τ on utility in Section 4.7.3.

4.6 Range Consistency Optimization

For single dimensional range aggregation, [37, 97] show that post-processing the hierarchy for *consistency*, *i.e.*, equality between aggregation on every node and the aggregations on its children, improves the aggregation utility. For multi-dimensional case, however, it is challenging to achieve the consistency without exponentially (to the number of attributes) increasing the post-processing overhead. In this section, we propose a cost-effective consistency optimization technique based on optimal range decompositions.

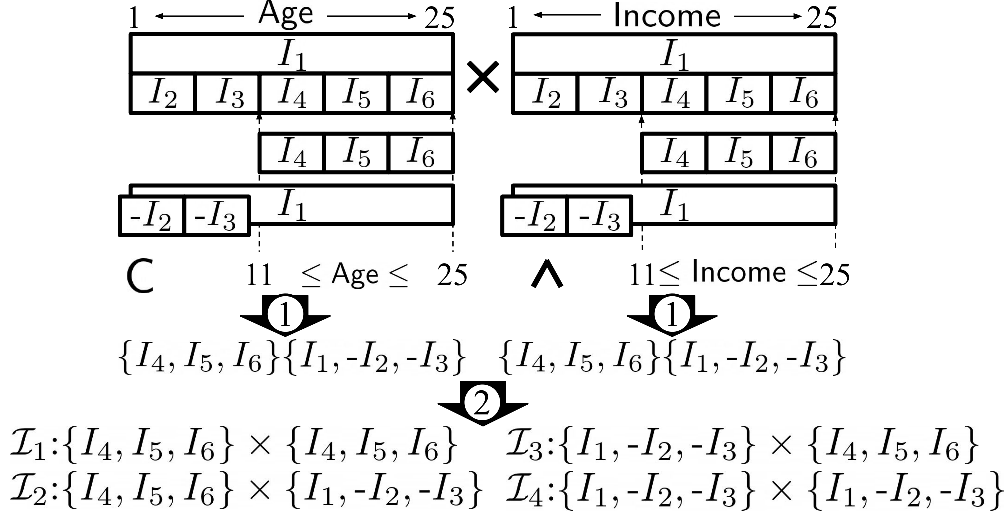


Figure 4.5: Optimal range decompositions for predicate C on attributes Age and Income, with fan-out of 5.

4.6.1 Optimal Range Decomposition

Given the multi-dimensional hierarchy, there are multiple ways to decompose the predicate C into the (hierarchy) nodes. For instance, in Figure 4.5, the predicate asks for $\mathsf{C} = \mathsf{Age} \in [11, 25] \wedge \mathsf{Income} \in [11, 25]$, and we can decompose it into minimum nine nodes in four different ways, *i.e.*, $\mathcal{I}_1, \ldots, \mathcal{I}_4$, without overlapping.

For decomposition $\mathcal{I}^{\mathsf{C}}$ of C, with $|\mathcal{I}^{\mathsf{C}}|$ nodes, since the frequencies of these nodes are added together for C, and the error bound for each node is the same, we have:

$$\mathrm{MSE}(\bar{\mathbf{P}}^{\mathsf{C}}(\mathbf{y})) \;\propto\; |\mathcal{I}^{\mathsf{C}}|. \tag{4.26}$$

Thus, the optimal decomposition of C is the one with the minimum number of nodes among all possible decompositions. The possible decompositions include the ones that only add up the nodes, as in previous work [37, 158], as well as those that subtract some nodes to achieve the equivalent range conditions, *e.g.*, $\mathcal{I}_4$ in Figure 4.5.

To find the decomposition with minimum number of nodes, we can first find the optimal decomposition for each single-attribute condition of C, and cross product them into the optimal de-

composition of C. For the example in Figure 4.5, ① we find the top-2 decompositions for attributes Age and Income separately, and ② cross-product them to get the top-4 decompositions for C. For ①, we can solve for the top-k decompositions for all possible ranges of each individual attribute using dynamic programming offline, with complexity polynomial to the domain size.

4.6.2 Consistency Optimization

The top-k decompositions of C are *consistent*, *i.e.*, they are all unbiased estimations of C, and we can combine them for better utility. For the top-k decompositions of C with $|\mathcal{I}_1^{\mathsf{C}}|, \ldots, |\mathcal{I}_k^{\mathsf{C}}|$ nodes, respectively, we assign w_i as the weight for $\mathcal{I}_i^{\mathsf{C}}$, and the weighted average of the k decompositions has $\mathrm{MSE}(\bar{\mathbf{P}}^{\mathsf{C}}(\mathbf{y})) \;\propto\; \sum_{i=1}^{k} w_i^2 |\mathcal{I}_i^{\mathsf{C}}|$. We can minimize the error bound using the KKT condition, under the constraint that $\sum_{i=1}^{k} w_i = 1$, and the optimal weights are $w_i = (1/|\mathcal{I}_i^{\mathsf{C}}|)/(\sum_{j=1}^{k} 1/|\mathcal{I}_j^{\mathsf{C}}|)$. Hence, the optimal error bound of the weighted average of the top-k decompositions is

$$\mathrm{MSE}(\bar{\mathbf{P}}^{\mathsf{C}}(\mathbf{y})) \;\propto\; 1/(\textstyle\sum_{i=1}^{k} 1/|\mathcal{I}_i^{\mathsf{C}}|), \tag{4.27}$$

which can be as small as $\frac{1}{k}$ of Equation (4.26), *i.e.*, when the top-k decompositions all have the same number of nodes. We state the utility improvement in Lemma 10

Lemma 10. *For range frequency aggregation of* C*, weighted averaging its top-*2^{d_q} *decompositions can improve the utility by* 2^{d_q}.

Proof Sketch: Let's first consider the simple case where each hierarchy of the d_q range attributes is single layer with B leaves. Then, suppose the optimal decomposition has l_i leaves on the i-th attribute, and the total number of nodes of the decomposition is $\prod_{i=1}^{d_q} l_i$. For the i-th attribute, we can substitute its l_i leaves with $B - l_i + 1$ intervals, *i.e.*, the parent interval minus the other $B - l_i$ leaves. In addition, the substitution does not overlap with the original optimal decomposition. Thus, we have 2^{d_q} consistent decompositions by choosing either the optimal or its substitution on

each attribute. The error bound of the optimal decomposition is $\propto \prod_{i=1}^{d_q} l_i$. And the error bound of the weighted average of the 2^{d_q} consistent decompositions is

$$\propto \prod_{i=1}^{d_q} l_i/(1+\sum_{j=1}^{d_q}\sum_{1\le i_1<\cdots<i_j\le d_q}\prod_{k=1}^{j}\frac{B-l_{i_k}+1}{l_{i_k}}). \tag{4.28}$$

When $l_i = B - l_i + 1$, *i.e.*, $l_i = \frac{B+1}{2}$, the error bound is 2^{-d_q} of that of the single optimal decomposition.

For hierarchies with more than one layers, the above argument applies, and the improvement reaches 2^{d_q} when there is a non-overlapping substitution with equal number of intervals for the optimal decomposition on each individual attribute in C. □

We evaluate the effect of k on the aggregation utility in Section 4.7.

4.7 Evaluation

We evaluate the utility of our end-to-end framework using synthetic and real-world datasets and queries. We conduct the evaluation on an Intel Xeon Platinum 8163 2.50GHz PC with 64GB memory. We set up a single node Spark cluster on the machine, using Hadoop 2.7.1, Spark 2.4.3 and Hive 2.3.5. We register the $\mathcal{R}$'s and $\bar{\mathbf{P}}$'s of our mechanisms as UDF's of SparkSQL. Each dataset is first loaded as table into Spark SQL, and we perturb each row, as a tuple, using the perturb UDF and collect them as the table to be released. At query time, ordinary SQL statement is issued from the user interface, and automatically rewritten using the proper $\bar{\mathbf{P}}$ UDF's, which will be executed by the underlying SparkSQL engine on the released table. For joint aggregation, multiple released tables are first joined on the join key by SparkSQL, and the SQL engine applies the joint frequency oracles over joined tuples and adds up their contributions into the result. We will open-source the code and the platform setup as a docker container.

Dataset We evaluate with two synthetic and two real-world datasets:

- SYN-1: Single-table synthetic data with 4 ordinal and 4 categorical attributes, and 1 non-

sensitive ordinal attributes. The number of tuples ranges from $2^{-2} \times 10^6$ to $2^2 \times 10^6$ (default 1×10^6). The domain size m for ordinal ranges from 5^2 to 5^4 (default 5^3), and that for categorical is 500. We simulate correlations among ordinal attributes by sampling from normal distribution, with $(\mu = \frac{m}{2}, \sigma = \frac{m}{4})$ for one of them, and adding it with Gaussian noises $(\mu = 0, \sigma = 10)$ for others.

- SYN-2: Synthetic data of two tables, both with one ordinal and one categorical attributes. For ordinal, $m = 5^3$, and, for categorical, $m = 500$. The two tables can be joined by the forged id, with two configurations: i) *one-to-one*; and ii) *one-to-many*, where a tuple in table one has $[1, 10]$ matching tuples in table two. And we test with 1×10^6 and 2×10^6 joined tuples.
- PUMS-P *[138]*: 3×10^6 census records of US citizens in 2017, with attributes AGE, MARST and UHRSWORK.
- PUMS-H *[138]*: 1940 census records of US households, with attributes STATE and CITY, and citizens, with attributes SEX, AGE, RACE and INCOME. Both records have HID as the join key, and it is primary-key for household and foreign-key for citizens. We focus on two samples: *IN*): 3.42×10^6 joined records from Indiana; and *IL*): 7.56×10^6 joined records from Illinois. For both samples, each household has up to $\tau_{\max} = 10$ citizens.

Mechanisms. We consider six mechanisms in the evaluation:

- HIO: The mechanism proposed in [158] for single-table MDA.
- SC-JOIN: Joint aggregation mechanism using SC (Section 4.4.1).
- AHIO: The augment-then-perturb instantiation of PRP (Section 4.3.3).
- EHIO: The embed-then-perturb instantiation of PRP (Section 4.3.3).
- HIO-JOIN: Joint aggregation over HIO (Section 4.4.2), with aug-ment-then-perturb and τ-truncation (Section 4.5.2).
- *-RDC: Mechanism * with utility optimization (Section 4.6).

We set the fan-out as 5 for hierarchies in both HIO and SC.

Queries. We evaluate the utility of aggregation queries for COUNT, SUM and AVG, as well as their sensitivity to factors: i) m: the domain size of the aggregation attribute; ii) vol: the volume of the predicate, *i.e.*, the ratio of the predicate space over the entire domain; and iii) d_q: the number of attributes in the predicate.

Metrics. We use two metrics to evaluate the estimation utility of $\bar{\mathbf{P}}$ over a set $\mathcal{Q}$ of queries of the same characteristics,

- *Normalized Mean Squared Error.* It measures how large the errors are relative to the maximum possible answer, *i.e.*, $\mathrm{NMSE}(\bar{\mathbf{P}}(\mathcal{Q})) = \frac{1}{|\mathcal{Q}|}\sum_{\mathsf{Q}\in\mathcal{Q}}\left((\bar{\mathbf{P}}^{\mathsf{Q}}(\mathcal{R}(\mathrm{T})) - \mathbf{P}^{\mathsf{Q}}(\mathrm{T}))/\Sigma_{\mathrm{T}}\right)^2$ where $\Sigma_T = |T|$ for COUNT, and $\Sigma_T = \sum_{t\in T}|t[A]|$ for SUM, are the upper bounds of aggregation. Note that this metric is identical to the mean square error used in [37].
- *Mean Relative Error.* It measures how large the errors are relative to the true answers, *i.e.*, $\mathrm{MRE}(\bar{\mathbf{P}}(\mathcal{Q})) = \frac{1}{|\mathcal{Q}|}\sum_{\mathsf{Q}\in\mathcal{Q}}\left|(\bar{\mathbf{P}}^{\mathsf{Q}}(\mathcal{R}(\mathrm{T})) - \mathbf{P}^{\mathsf{Q}}(\mathrm{T}))/\mathbf{P}^{\mathsf{Q}}(\mathrm{T})\right|$.

We use NMSE for COUNT and SUM, and MRE for AVG.

4.7.1 Attribute Aggregation

We first evaluate aggregation on attribute, with COUNT, SUM and AVG queries, using the HIO, AHIO and EHIO mechanisms.

Benchmark with SYN-1. First, we evaluate the effect of the volume of the aggregation query. We sample queries with range volume from 0.04 up to 0.30, and fixed $\epsilon = 2.0$, $d = 2$, $m = 125$ and $d_q = 1$. Figures 4.6a, 4.6d, and 4.6e show the aggregation errors for HIO, AHIO and EHIO. We observe that, for COUNT, HIO performs better than AHIO and EHIO, and the reason is that both AHIO and EHIO spare privacy budget for the rounding value to support attribute aggregation. For SUM and AVG, AHIO and EHIO outperform HIO consistently. The volume does not have much effect on COUNT or SUM, and the relative error of AVG decreases as the volume increases.

Second, we evaluate the effect of m, and test with $m = 5$, 25, 125 and 250 for the aggregation

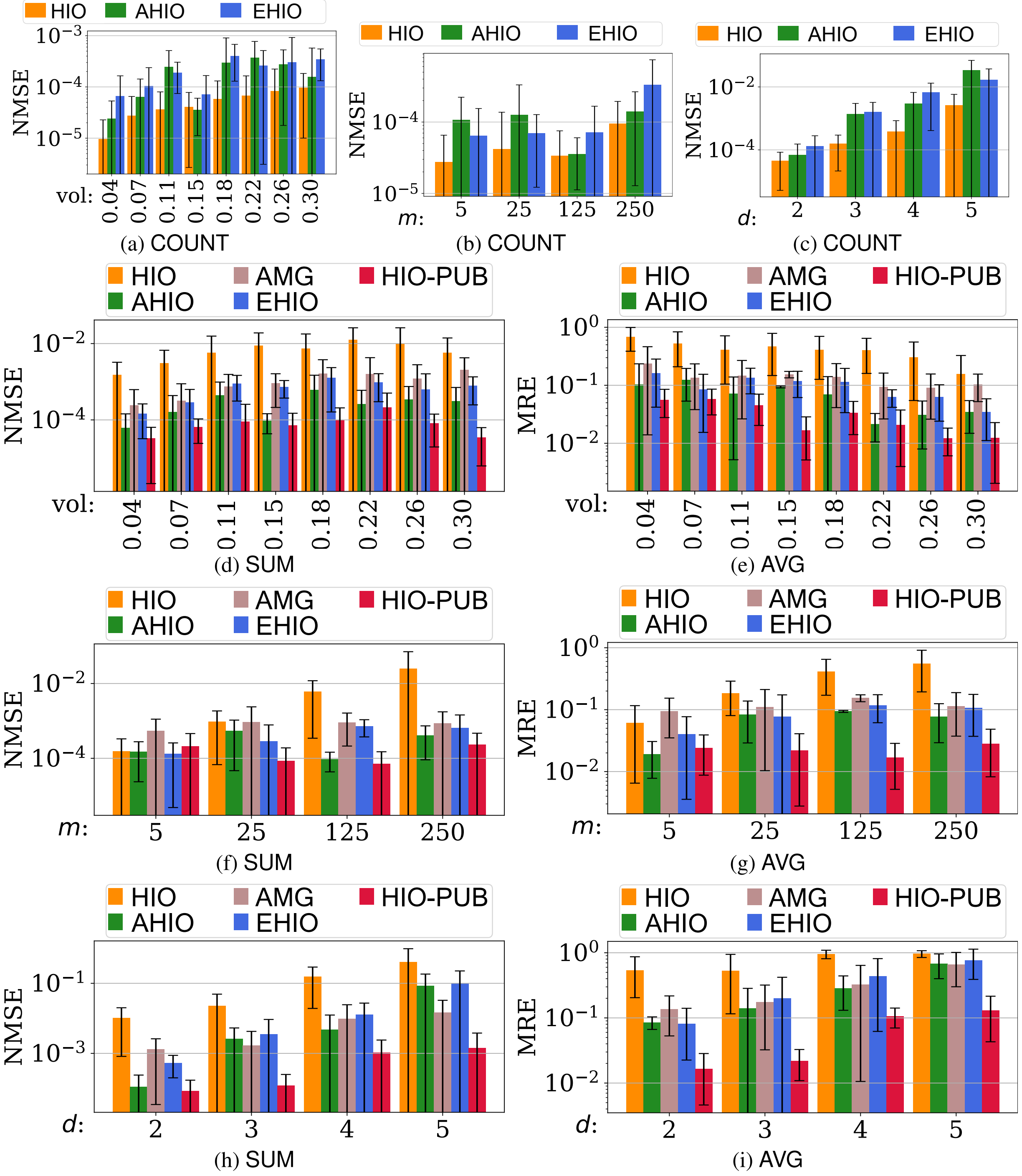


Figure 4.6: Aggregation on sensitive attribute using SYN-1.

attribute, with $\epsilon = 2.0$, $d = 2$, vol $= 0.15$, and $d_q = 1$ (Figures 4.6b,4.6f and 4.6g). We first observe that the domain size has slight effects on the absolute error of COUNT since there is an logarithmic relation between the hierarchy height and the domain size. As for SUM and AVG, the

	$\epsilon =$	0.5	1.0	2.0	5.0	True
Q1	HIO	32.74 ± 20.03	28.25 ± 6.46	27.49 ± 4.07	26.6 ± 0.69	26.62
	AHIO	26.88 ± 5.15	27 ± 2.82	26.71 ± 1.2	26.23 ± 0.75	
	EHIO	$\mathbf{26.93 \pm 4.07}$	$\mathbf{26.76 \pm 2.06}$	$\mathbf{26.07 \pm 1.05}$	$\mathbf{26.7 \pm 0.27}$	
	AHIO-RDC	26.88 ± 5.15	27 ± 2.82	26.71 ± 1.2	26.23 ± 0.75	
Q2	HIO	44.6 ± 47.24	35.95 ± 28.81	33.91 ± 10.08	30.73 ± 3.36	29.98
	AHIO	24.53 ± 12.08	24.33 ± 5.79	29.63 ± 3.43	29.55 ± 1.25	
	EHIO	28.60 ± 18.71	29.16 ± 7.2	29.33 ± 2.89	30 ± 0.7	
	AHIO-RDC	$\mathbf{31.06 \pm 10.37}$	$\mathbf{29.64 \pm 4.71}$	$\mathbf{29.58 \pm 1.82}$	$\mathbf{29.95 \pm 0.69}$	
Q3	HIO	-110.27 ± 320.65	22.76 ± 121.81	54.35 ± 73.6	26.32 ± 8.48	30.82
	AHIO	4.15 ± 133.45	37.5 ± 67.08	26.21 ± 20.34	29.57 ± 4.91	
	EHIO	42.58 ± 243.41	26.58 ± 36.65	29.18 ± 14.62	30.78 ± 1.76	
	AHIO-RDC	$\mathbf{25.13 \pm 98.24}$	$\mathbf{30.7 \pm 25.03}$	$\mathbf{29.18 \pm 9.12}$	$\mathbf{30.83 \pm 1.74}$	

Figure 4.7: Q1-Q3 on PUMS-P.

domain size does not affect the errors for AHIO or EHIO much, and the error for HIO increases fast as the domain size increases.

Third, we evaluate the effect of the number of attributes, *i.e.*, d, in the data, and test with $d = 2$, 3, 4 and 5 (Figures 4.6c, 4.6h and 4.6i). As d increases, all the errors increase.

To better understand the effect of the rounding technique, we evaluate using HIO when the aggregation attribute is released as non-sensitive. We show its results as HIO-PUB in Figure 4.6, and we only report for SUM and AVG since its utility on COUNT is the same as HIO. We observe consistent drop on utility for AHIO and EHIO, compared to HIO-PUB.

We also evaluate the PRP framework when the underlying multi-dimensional analytical frequency oracle is *marginal release* (MG) [45], and we report the results for augment-then-perturb using MG (AMG) in Figure 4.6. As the volume increases, the aggregation error of AMG increases much faster than those of AHIO and EHIO (Figure 4.6d and 4.6e) because MG estimates a large range predicate with many point conditions. When the volume is not large, *e.g.*, ≤ 0.15, the errors of AMG are comparable to those of AHIO and EHIO.

Sample Queries on PUMS-P. We test on the PUMS-P dataset with the following three queries:

Q1: `SELECT` AVG(UHRSWORK) `FROM` PUMS-P `WHERE` MARST = Married;

Q2: `SELECT` AVG(UHRSWORK) `FROM` PUMS-P `WHERE` MARST = Married `AND` $31 \leq$ AGE $\leq$ 70;

Q3: `SELECT` AVG(UHRSWORK) `FROM` PUMS-P `WHERE` MARST = Single `AND` $31 \leq$ AGE ≤ 50.

We set $\epsilon = 0.5, 1, 2, 5$, and test with HIO, AHIO, EHIO and AHIO-RDC. For each setting, we release the data 10 times, evaluate Q1-Q3 on the 10 releases, and report the mean aggregation results, together with the standard deviations, in Figure 4.7. We include the true results for the three queries in the right-most column. For all mechanisms and settings, *i.e.*, the pair of query and ϵ, we have the true aggregation results in the standard deviation intervals. For each setting, we highlight the mechanism with the smallest standard deviation because it provides the best confidence interval. We observe that AHIO, EHIO and AHIO-RDC consistently out-performs HIO, and AHIO and EHIO are comparable. AHIO-RDC performs better than AHIO, especially for the more selective queries, *e.g.*, Q3. In addition, the standard deviation increases when the range predicate gets more selective, and decreases as ϵ increases.

4.7.2 Range Consistency Optimization

We next evaluate the effect of the range consistency optimization on attribute aggregations using the AHIO and AHIO-RDC mechanisms. We use SYN-1 with 4 ordinal sensitive attributes, and fix $\epsilon = 2$. We evaluate the effects of d_q and the number of top decomposition k. In particular, we evaluate aggregation queries with range predicate on d_q = 1, 2, 3 and 4 of the attributes, and vary k to be 2, 4, 8 and 16. Figures 4.8a, 4.8c and 4.8e show the results. We observe that HIO-RDC improves the utility for COUNT and SUM, as the number of sensitive attributes in the range predicate increases. In addition, larger k performs better when the number of attributes in the predicate is larger, which is consistent with Equation (4.27). Note that the improvement on utility is not exactly k because the analysis in Lemma 10 is the upper bound when the k decompositions

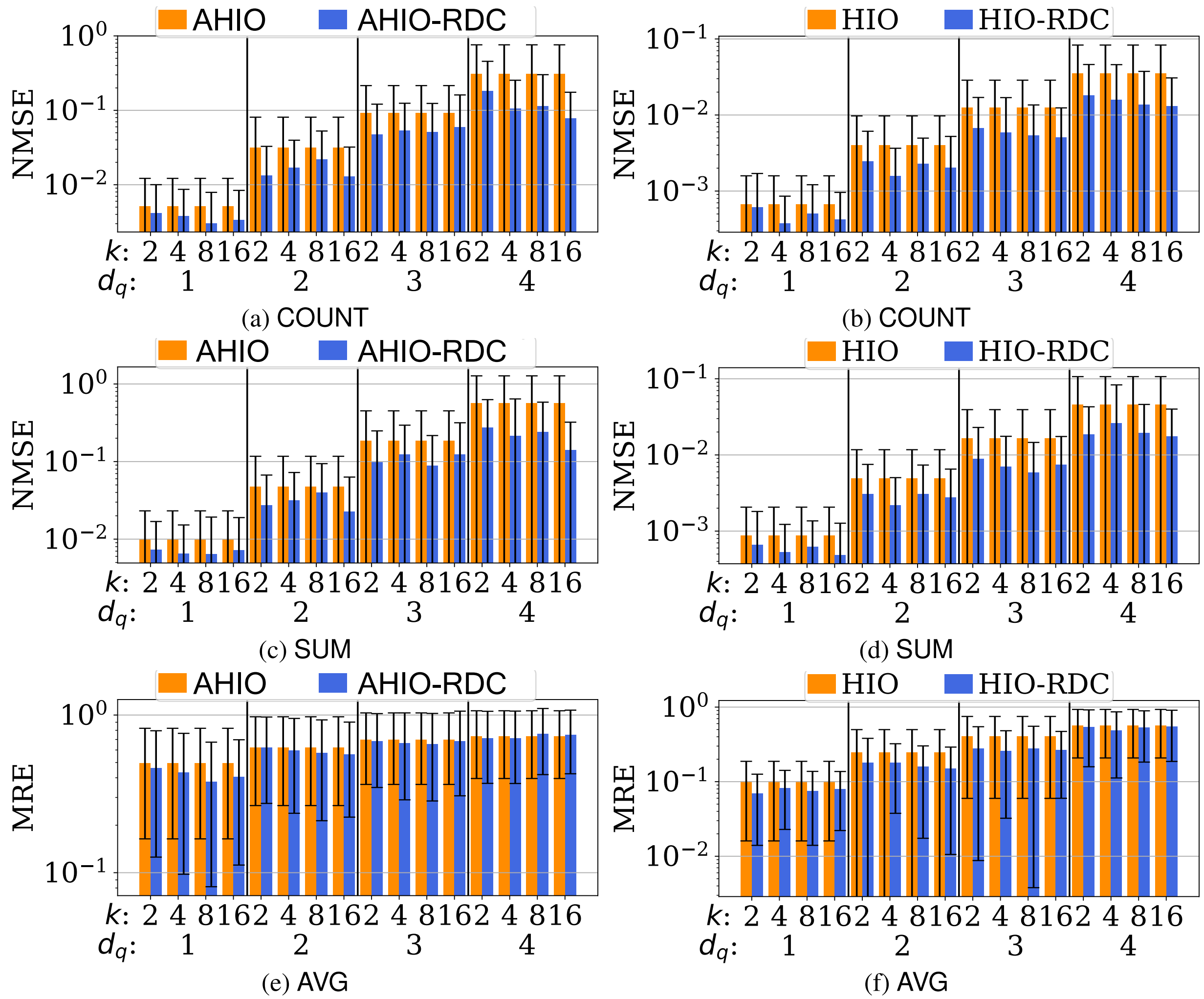


Figure 4.8: Effect of range consistency optimization using SYN-1. Left: aggregation on sensitive attribute. Right: aggregation on non-sensitive attribute. k indicates the number of top decompositions and d_q indicates the number attributes in the range predicate.

are of the same error bound, which is not guaranteed for randomly selected range predicates in our experiments.

To benchmark the effectiveness of the range consistency optimization against the state of the art, we compare the utility of HIO-RDC against that of HIO, using queries that aggregate on the non-sensitive attribute of SYN-1 with range predicate on 1, 2, 3 and 4 of the ordinal attributes, and $k \in \{2, 4, 8, 16\}$. Figures 4.8b, 4.8d, and 4.8f show the results, and HIO-RDC consistently outperforms HIO for all the three types of aggregations.

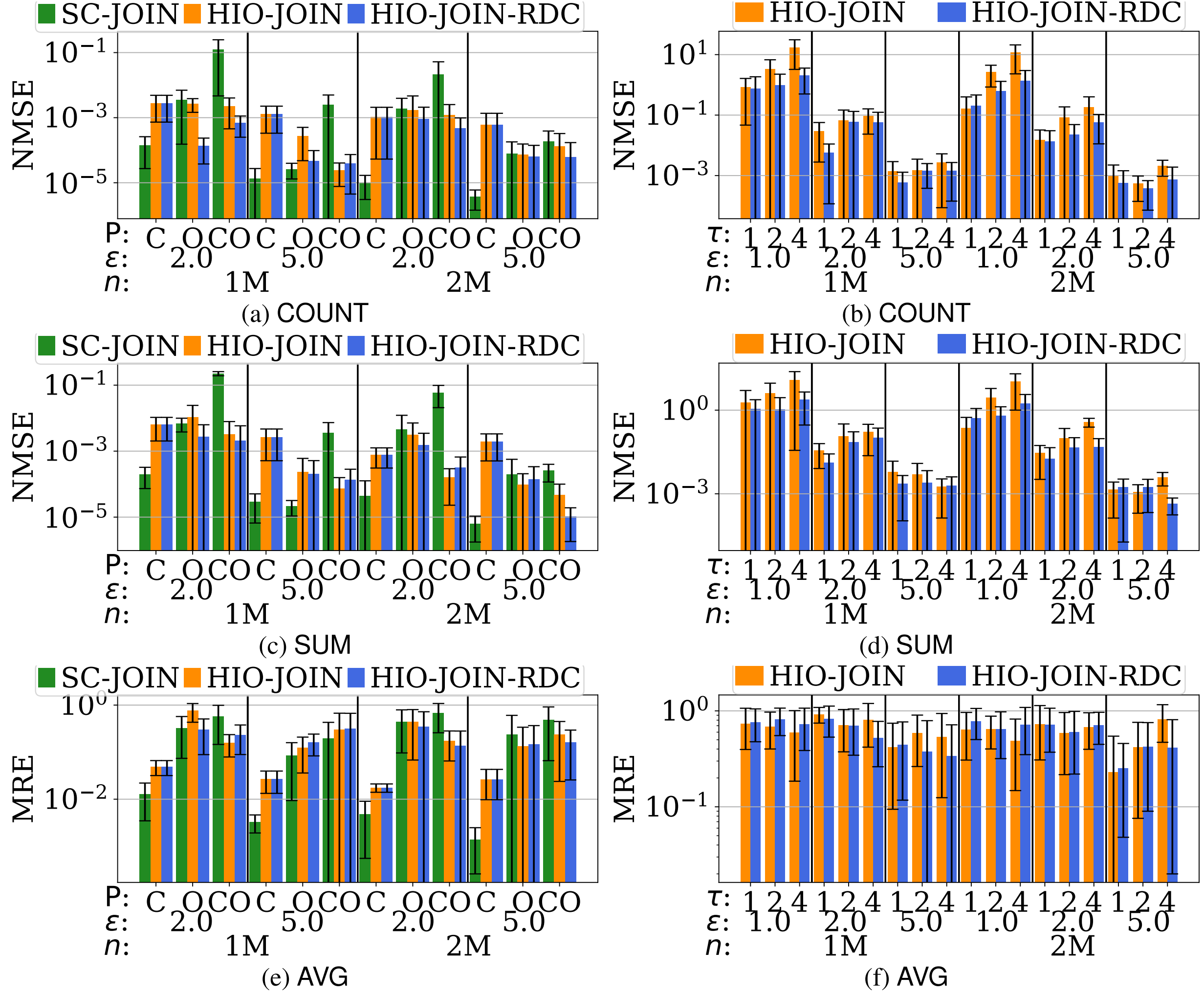


Figure 4.9: Joint aggregation over SYN-2. Left: one-to-one, C, O, CO indicate one categorical, one ordinal, one categorical and one ordinal attributes in the predicate, respectively. Right: one-to-many, τ is the truncation number.

4.7.3 Joint Aggregation

Finally, we evaluate joint aggregation across two tables. We first evaluate with SC-JOIN, HIO-JOIN and HIO-JOIN-RDC using SYN-2 for sensitivity analysis. Then we evaluate with HIO-JOIN and HIO-JOIN-RDC on PUMS-H as case-study.

One-to-one. We test with queries of different range predicates on the attributes: i) C: one point condition; ii) O: one range condition; and iii) CO: one point and one range conditions. The volume for the range condition is fixed at 0.12. If the predicate involves only one attribute, then

the aggregation attribute is in the other table. Figures 4.9a, 4.9c and 4.9e show the results. First, the overall estimation utility improves as either ϵ or the table size n increases. Second, as the predicate gets complicated, HIO-JOIN outperforms SC-JOIN. In addition, HIO-JOIN-RDC consistently improves the aggregation utility over HIO-JOIN.

One-to-many. We test with queries that aggregate on the attribute with the CO range predicate of volume 0.12. We evaluate the HIO-JOIN and HIO-JOIN-RDC schemes because, as we show above, SC-JOIN is worse than HIO-JOIN for this kind of range predicate. For the one-two-many setting, we enforce the user-level LDP using τ-truncation, and we evaluate the same aggregation with $\tau = 1, 2, 4$. Figures 4.9b, 4.9d and 4.9f show the results. First, larger ϵ or table size n leads to better estimation utility for all types of queries. Second, as τ increases, the estimation utility on COUNT and SUM decreases, and the effect on AVG is not substantial. Third, the range optimization technique improves the overall utility consistently.

PUMS-H Case Study. We conduct a case study using the PUMS-H dataset, for Indiana and Illinois, respectively. The study answers the following question: *how many people living in Indiana (Illinois) are in the city Indianapolis (Chicago) and in the specific age group?* In addition to the count, we include in the analysis the sum and average on AGE. The age groups are $[1\text{-}20]$, $[21\text{-}45]$, $[46\text{-}70]$, $[71\text{-}95]$ and $[96\text{-}120]$. We evaluate HIO-JOIN and HIO-JOIN-RDC, and set $\epsilon = 2$ and 5. The truncation number τ is fixed at 1 for this study. Figure 4.10a, 4.10c and 4.10e show the results for Indiana, and Figure 4.10b, 4.10d and 4.10f for Illinois.

4.7.4 Handling Group-by Queries

Our solution can be extended for *group-by queries* to perform aggregation analysis over joins of relations from different services and summarize the results by some *group-by attribute*. To answer the group-by queries, we share the same assumption, as previous works [5, 51, 131] do, that the dictionary (*i.e.*, the set of all possible values) of the group-by attribute is known to public. Note that this assumption does not affect the privacy of each individual's data, as the dictionary is

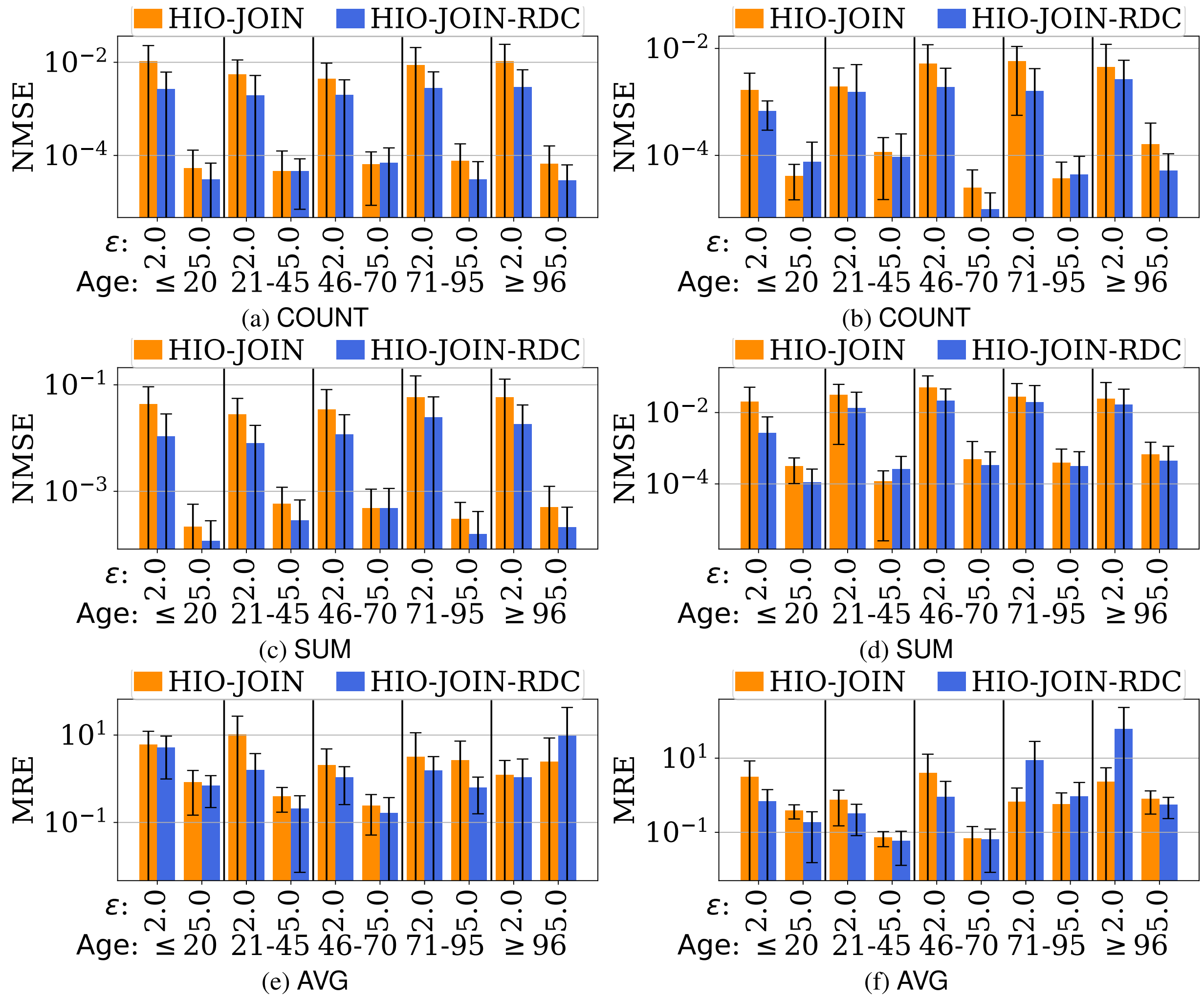


Figure 4.10: Joint aggregation over PUMS-H. Left: Indiana. Right: Illinois. The predicate is "CITY $= x$ AND AGE $\in [l, r]$".

independent on the content of the data and is usually public knowledge (*e.g.*, group by CITY or RACE).

Algorithm HIO-GROUP-BY**.** Consider a query $\mathsf{Q} = (A, \mathsf{C}, T)$ aggregating A under predicate C on a relation or join of relations T, with a group-by attribute G. Abusing the notation, we also use G to denote the dictionary of attribute G. With the perturbation algorithms (τ-Truncation and Partition-Rounding-Perturb framework to ensure ϵ-uLDP) unchanged, our query estimation algorithm HIO-GROUP-BY is outlined in Figure 4.11. Let consider two cases:

If G is a nonsensitive attribute and known to public, we can partition the perturbed relation

Aggregation query $\mathsf{Q} = (A, \mathsf{C}, T)$ group by G:

1: **for** $v \in \mathsf{G}$ **do**
2: **if** G is nonsensitive **then**
3: $T(v) \leftarrow \sigma_{\mathsf{G}=v}(\mathcal{R}(T))$ ($\sigma.(\cdot)$ is *selection*)
4: $S_v \leftarrow \bar{\mathbf{P}}^{(A,\mathsf{C})}_{\textsf{HIO-JOIN}}(T(v))$
5: **else** $S_v \leftarrow \bar{\mathbf{P}}^{(A,\mathsf{C}\wedge\mathsf{G}=v)}_{\textsf{HIO-JOIN}}(\mathcal{R}(T))$
6: **return** S

Figure 4.11: HIO-GROUP-BY: for group-by queries

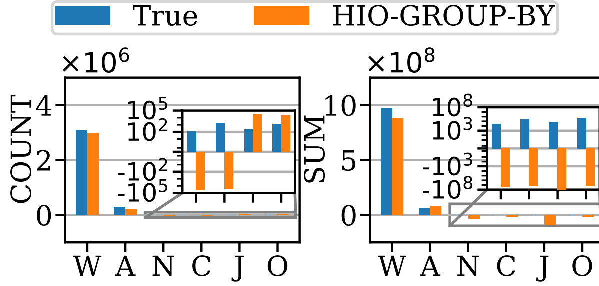


Figure 4.12: One run of HIO-GROUP-BY estimation versus the ground truth.

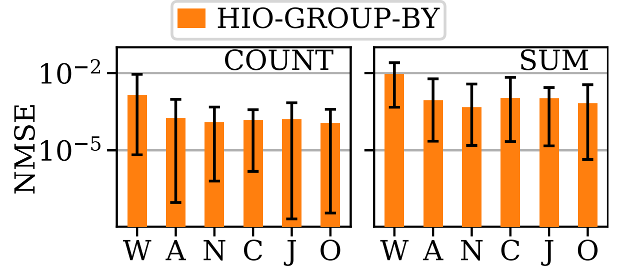


Figure 4.13: Average errors (over 15 runs) of HIO-GROUP-By by group.

$\mathcal{R}(T)$ by G. For each group $v \in \mathsf{G}$, we read the perturbed tuples of this group into $T(v)$ (line 3) and apply HIO-JOIN on $T(v)$ to estimate the aggregation value (line 4).

If G is a sensitive attribute, we can still enumerate all possible values of G with the public dictionary, but cannot access its true value for each perturbed tuple in $\mathcal{R}(T)$. To estimate the answer to Q, we rewrite it a bit: for each group $v \in \mathsf{G}$, we extend the predicate C to be "$\mathsf{C} \wedge \mathsf{G} = v$" and apply HIO-JOIN to process Q with the extended predicate on $\mathcal{R}(T)$ (line 5). In this way, we obtain an unbiased estimate of the aggregation value for each group.

Error bounds. If G is non-sensitive, the error bound of estimated aggregation in each group follows from Lemmas 5, 6, or 9 for different join types. As each group is processed independently, the mean squared error (MSE) is proportional to the size of the group (n in the lemmas is equal to the number of tuples in each group).

The case when G is sensitive is more difficult. The error bound for different join types again

follows from Lemmas 5, 6, or 9. However, since the aggregation value is recovered from all the perturbed tuples with an additional constraint "$\mathsf{G} = v$", the MSE is proportional to the total number of tuples in T (n in the lemmas is equal to $|T|$) as well as the aggregation value. Note that MSE is the "squared" error, the above error bound implies that, for groups with large aggregation values, the relative estimation errors are smaller.

Case study and empirical evaluation. We test a group-by query on the PUMS-H dataset: what are COUNT and SUM(Income) of people whose households are in Chicago, grouped by RACE[1]? All the attributes, including RACE, are sensitive. We compare the estimates obtained by HIO-GROUP-BY in one run ($\epsilon = 5$) with the ground truth (Figure 4.12), and further report the average error (NMSE) of each group for 15 runs of the mechanism (Figure 4.13). The key observation is that, our HIO-GROUP-BY preserves the trend across groups very well especially for large groups (*e.g.*, groups 'W' and 'A'), which enables us to identify the "top groups" and support accurate decision making on them. NMSE for all the groups are very close, which is consistent with the theoretical error bounds. For small groups (*e.g.*, groups 'J' and 'O'), the error is relatively large because of small aggregation values, which is an inherently difficult case for DP-based estimations.

1. W: White; A: African American; N: American Indian or Alaska Native; C: Chinese; J: Japanese; O: Other Asian or Pacific Islander.

CHAPTER 5

SEARCHING ENCRYPTED DATA WITH SIZE-LOCKED INDEXES

5.1 Introduction

Client-side encryption protects data stored at untrusted servers, but deploying it poses both usability and security challenges. Off-the-shelf file encryption disables server-side data processing, including features for efficiently navigating data at the request of the client. And even with well-designed special-purpose encryption, some aspects of the stored data and user behavior will go unprotected.

This work concerns text searching on encrypted data, and targets replicating, under encryption, the features provided in typical plaintext systems efficiently and with the highest security possible. Diverse applications are considered, but a concrete example is a cloud storage service like Dropbox, Google Drive, and iCloud. These systems allow users to log in from anywhere (e.g., from a browser) and quickly search even large folders. The search interface accepts multiple keywords, ranks the results, and provides previews to the user. To provide such features, these storage services retain access to plaintext data. In contrast, no existing *encrypted* storage services (e.g., Mega, SpiderOakOne, or Tresorit) supports keyword search.

The problem of implementing practical text search for encrypted data was first treated by Song, Wagner, and Perrig [142], who described several approaches to solutions. Subsequently a primitive known as *dynamic searchable symmetric encryption (DSSE)* was developed over the course of an expansive literature (c.f., [23–25, 30, 32, 40–42, 56, 86, 88, 94, 110, 143, 171]). But DSSE doesn't provide features matching typical plaintext search systems, and more fundamentally, all existing approaches are vulnerable to attacks that recover plaintext information from encrypted data. The security of DSSE is measured by leakage profiles which describe what the server will learn. *Leakage abuse attacks* [22, 29, 61, 64, 79, 116, 130, 152, 162, 176] have shown that DSSE schemes can allow a server to learn substantial information about the encrypted data and/or queries. Even more

damaging have been *injection attacks* [29,176], where adversarially-chosen content (documents or parts of documents) are inserted into a target's document corpus. After inserting a small amount of content, an attacker can identify queried terms accurately by observing when those documents are returned, which is revealed by the so-called *results pattern* leaked by DSSE schemes.

Contributions. This work returns to a simple, folklore approach to handling search over encrypted documents: simply encrypt a standard search index, storing it remotely and fetching it to perform searches. In fact this idea was first broached, as far as we are aware, by the original Song, Wagner and Perrig [142] paper, but they offered no details about how it would work and there has been no development of the idea subsequently. While the approach has many potentially attractive features, including better security and the ability to provide search features matching plaintext search, it has not received attention perhaps because it seems technically uninteresting and/or because the required bandwidth was thought impractical — indexes can be very large for big data sets.

We initiate a detailed investigation of encrypted indexes. Our first contribution is to show the insecurity of naively encrypting existing plaintext search indexes, such as those produced by the industry-standard Lucene [1]. The reason is that Lucene and other tools use compression aggressively to make the index — a data structure that allows fast ranking of documents that contain one or more keywords — as compact as possible. Compression before encryption is well known to be dangerous, and indeed we show how injection attacks would work against this basic construction.

We therefore introduce what we call *size-locked indexes*. These are specialized indexes whose representation as a bit string has length that is a fixed function of information we allow to leak. We show a compact size-locked index whose length depends only on the total number of documents indexed and the total number of postings handled. By coupling our size-locked index with standard authenticated encryption, we are able to build an encrypted index system that works with stateless clients and provides better search functionality (full BM25-ranked search) than prior approaches, while resisting both leakage abuse and injection attacks. We provide a formal security model and

analysis.

Our encrypted size-locked index already provides a practical solution for moderately sized document sets. But for larger document sets it can be prohibitive in terms of download bandwidth, for example providing a 217 MB index for the full 1.7 GB classic Enron email corpus. Here prior techniques like DSSE require less bandwidth to perform searches. We therefore explore optimizations to understand whether encrypted indexes can be made competitive with, or even outperform, existing approaches.

We show two ways of partitioning our size-locked indexes to reduce bandwidth. Our vertical partitioning technique exploits the observation that, in practice, clients only need to show users a page of results at a time. We therefore work out how to securely partition the index so that the top ranked results are contained within a single (smaller) encrypted index, the next set of results in a second-level index, and so on. Handling updates is quite subtle, because we must carefully handle transferring data from one level to another in order to not leak information in the face of injection attacks. We provide an efficient mechanism for updates. We show formally that vertical portioning prevents injection attacks and only leaks (beyond our full index construction) how many levels a user requested. Because most users are expected to most often need only the first level, vertical partitioning decreases average search bandwidth by an order of magnitude.

We also consider horizontal partitioning which separates the space of keywords into a tunable parameter P of partitions, and uses a separate vertically partitioned size-locked index for each. This gives us a finely tunable security/performance trade-off, since now performing searches and updates can be associated by an adversarial server to certain partitions. We formally analyze the security achieved, and heuristically argue that for small P is less dangerous than the result patterns revealed by prior approaches. In terms of performance, horizontally plus vertical partitioning enable us to almost match the bandwidth overheads of DSSE. For Enron fetching the first page of results requires just 0.95 MB.

5.2 Problem Setting

As mentioned in the introduction, we target building efficient encrypted search for cloud services such as Dropbox, Google Drive, Mega, and others. Some already offer client-side encryption (without search), and for others it is straightforward to add it. We will therefore leave the actual file storage out of our modeling, and assume that filenames and content are encrypted, and focus on adding search capabilities in a practical way. We target search functionality that matches that of plaintext services that handle search by directly accessing plaintext data on the server side.

To understand search in contemporary storage services, we briefly surveyed several widely used ones. Some features are not precisely documented, in which cases we performed simple experiments to assess functionality. We summarize our findings in Figure 5.1. We do not include in the table encrypted services like Tresorit, Mega, Sync.com, and SpiderOakOne which provide client-side encryption but only currently allow search of (unencrypted) filenames (for the first three) or no search at all (for SpiderOakOne). As can be seen, search features vary across plaintext services. Most support conjunctions of keywords (but not disjunctions), perform relevance ranking of some small number k of returned results, and update search indices when keywords are added to documents. Interestingly, none of these services appear to update search indices when a word is removed from a document (i.e. once a word is added, that document will contain that word forever).

All of these services supported both a full-fledged application that mirrored data on a client as well as *lightweight* portals for accessing data via the web and mobile apps. When the data is locally held by the client, search is easy (say, via the client system's OS tools). For lightweight clients, search queries are performed via a web interface with processing at the server. Previews of the top matching documents are returned, but documents are not accessed until a user clicks on a returned link for the matching document. A user can also request the subsequent pages of results.

In summary, our design requirements include the following.

- Lightweight clients with no persistent state should be supported. Users should be able to log in

App	Type	Rank	Preview	Top-k	Update
Dropbox	$\wedge$	rel	n,d,s,p	10	✓
Box	$\wedge, \vee$	rel	n,d,s,p	6	✓
Google Drive	$\wedge$	rel	n,d,s	8	✓
Microsoft OneDrive	$\wedge$	rel	n,d,s	8	✓
Amazon Drive	$\wedge$	date	n,d,s	8	✗

Figure 5.1: Search features in popular storage services. Services support either just conjunctions ($\wedge$) or additionally disjunctions ($\vee$) over keywords. The top-k results are ranked according to relevance (rel) or just date. Previews may of search results may include name (n), modification date (d), file size (s), and/or the parent directory (p). Search indices may be updated due to edits within a file (✓) or only when documents are added or deleted (✗).

with a password and search from anywhere.

- Multi-keyword, ranked queries should be supported. Most services support conjunctive multi-keyword queries ranked by relevance.
- Query results may be presented in pages of k ($k \approx 10$) results with metadata previews, including name, date and size.
- The addition and deletion of entire documents should be supported. Updating indices in response to deletion of words from documents is optional.

Summary. In summary, we do not currently have systems for searching encrypted documents that (1) come close to matching the functionality of contemporary plaintext search services; (2) that work in the required deployment settings, including lightweight clients; and (3) that resist attacks.

5.3 Insecurity of Encrypting Standard Indexes

In this section we describe and analyze how modern search indices, should they be encrypted, are vulnerable to various threats, especially file injection attacks. The basic problem is that the length of industry-standard indices varies significantly based on the plaintext content, and lengths are visible even after encryption to an adversary. Despite the suggestion of index encryption being around for close to twenty years [142], we are unaware of any prior investigations into the security

of this basic approach.

We recall the approach of Song, Wagner and Perrig [142] of storing encrypted indexes at a server, fixing some details that were not discussed. We then show that using this approach with a standard tool like Lucene can result in a system vulnerable to document injection attacks (and possibly more). The key observation is that changes in the byte-length of the encrypted index blob will depend on sensitive data in an exploitable way.

Naive encrypted indexing. A simple approach to adding search to an outsourced file encryption system, such as those discussed in Section 5.2, is to have a client build a Lucene (or other standard) index, encrypt it using their secret key, and store it with the service. To perform a search, a lightweight client can download the entire encrypted index, decrypt it, initialize an in-memory index, and then use it to perform searches. Should client state be dropped (e.g., due to closing a browser or flushing its storage), the next search will require fetching the encrypted index again.

To perform updates to the index, because a new file is added or new keywords are added to a file, the system can work in two different ways. The most obvious is to download the encrypted index (if not already downloaded), update it directly, re-encrypt, and upload. In an update-heavy workload, where the frequency of document updates is greater than the frequency of searches, it would be more efficient to defer updating the index until the next search. More specifically, an update delta d describing the changes to the index (i.e., the new keyword and/or new postings, as well as changes to document frequencies) can be encrypted separately and appended to an update list stored in encrypted form at the server. The next time a search is perform, the δ values can be recovered and merged into the full index. This *lazy merging* avoids having to download the full index each time an update occurs.

Prior work has observed that indices can be large, which may make this basic approach inefficient for settings where clients or network bandwidth are constrained and the dataset is large. We will explore efficiency issues in more detail in later sections. A more critical issue is that the security of this approach is far from certain.

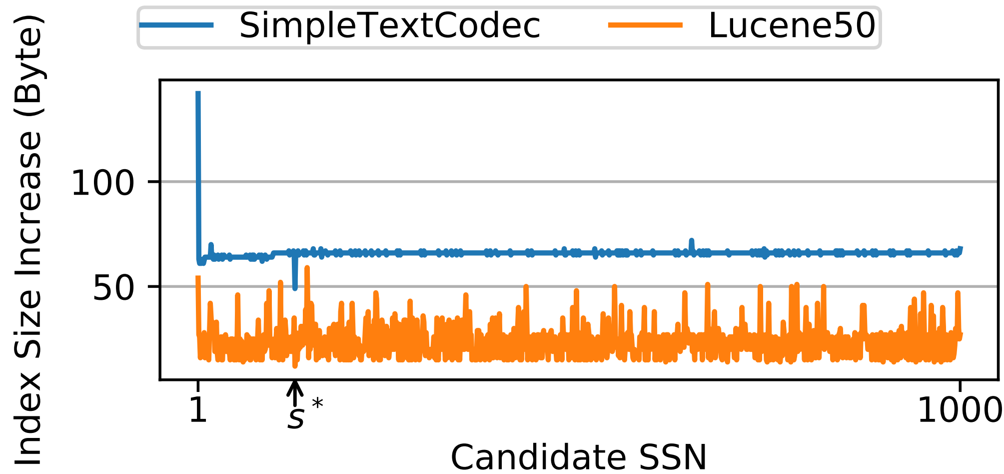


Figure 5.2: Document-injection attack on Lucene to recover indexed term. The term s^* results in a noticeably smaller change in byte-length than other terms.

Security case study: Lucene. Assuming encryption meets a standard notion like CPA security, a standard reduction would prove that this basic encrypted index approach reveals only the byte-lengths of index and (in the case of lazy updates) the update messages.

While simple to express, this leakage depends on plaintext data in a complicated way. We thus performed a simple experiment to highlight the possibility of practically-exploitable issues with naive encrypted indexing using Lucene. The basic question is: Does the length of encrypted ciphertexts observed by an adversary enable attacks? Compression (delta encodings and variable-byte encodings), and data structure overheads mean that the size of the index is related to various sensitive pieces of information, such as the number of unique keywords present in the index. Factors like periodic merging and optimization introduce noise variation, obscuring the channel. We show however that enough information survives in a controlled setting, and conclude that this system is unlikely to be secure in practice.

We simulated a naive encrypted index using Lucene version 7.7.3. For simplicity, we configured Lucene to immediately merge segments after each update and also disabled the skip list feature, which accelerates lookup while slightly increasing the index size.[1] We also configure

1. We see no reason why these features would prevent attacks, even though they may add extra noises.

Lucene not to include the positions in the postings because such information are not useful for the kinds of search queries we target supporting. We use two different built-in Lucene encodings, the naive `SimpleTextCodec` and `Lucene50` (the default). The adversary is given the byte-length of the encoding after updates: In the case of `SimpleTextCodec` only one file is output, and we used that length. In the case of `Lucene50` several files are output, so we used the total sum of their lengths. This captures the assumption that encryption leaks the exact length of the plaintext data, which is true of most popular symmetric encryption schemes.

We considered the following file-injection attack setting: An index has been created that contains a single document containing exactly one term which is a random 9-digit numerical, e.g., a social security number (SSN) that we denote s^*. An adversary is given a list of 1,000 random SSNs $s_1, \ldots, s_{1000}$, one of which equals s^*. Its goal is determine which s_i equals s^*. Our attacker is allowed to repeatedly inject a document of its choosing and observe the new byte-length of the index. A secure system should keep s^* secret, even against this type of adversary.

We ran simulated document injection attacks. Our attacker works as follows: It records the initial byte-length of the index. Then for each of the 1,000 SSNs in its list, the attacker requests that a document consisting of exactly that SSN be injected. It then records the change in byte-length of the index. (Documents are not deleted here, so at the conclusion of this attack, the index contains 1,001 documents.) Finally, the attack finds the injected SSN that resulted in the smallest change in byte-length, and uses that as its guess for s^*.

The intuition for this attack is that if a term in already in the Lucene index, then adding a posting with that term should require fewer bytes than a "fresh" term. This is because adding fresh terms will result in extra overhead, like new head nodes in hash table linked-list which are slightly larger than other nodes.

We plot two example runs in Figure 5.2, one for each encoding. The horizontal axis corresponds to the injected terms in order, and the vertical axis is the change in byte-length after each injection. We observe first that our attack works for both encodings, since the smallest change

corresponded to s^* (as is visible in the plot). We also observe that this worked despite quite a bit of noise, especially in case of `Lucene50`, where the variation in changes due to the internal function of Lucene is visible. We repeat the attack 100 times, each with a different s^* from the 1000 candidate SSNs, and the attack succeeded every single time.

Discussion. While a toy example, the highlighted weakness appears to be only the beginning of an exploration of the size-side-channel in this construction. For instance, an index may compress its term dictionary, so injecting terms similar to existing terms may result in a different byte-length change than injection terms that are far from the existing dictionary. We have also not exploited the variable-byte encoding used in postings lists. And while implementation noise issues may confound some attacks, we conclude that a well-controlled approach to sizing indexes is required to have confidence in the security of an encrypted search system.

5.4 Basic Encrypted Size-Locked Indexes

We now describe an approach to encrypted indexes that ensures security from injection and leakage-abuse attacks. The key technique is what we call a *size-locked index*, which is an index whose byte-length encoding is a fixed function of features we are willing to leak, particularly the total cumulative number of postings N and the number of documents n.

One potential approach for a size-locked index would be to use a standard index tool like Lucene as a black box, resulting in a (variable length) encoded string, and padding to a target fixed length. But it is unclear how to do this with any confidence in security as it is hard-to-predict the length after padding, and adversarial injections may interfere with predicted bounds on length. We instead take a direct approach to constructing a size-locked index.

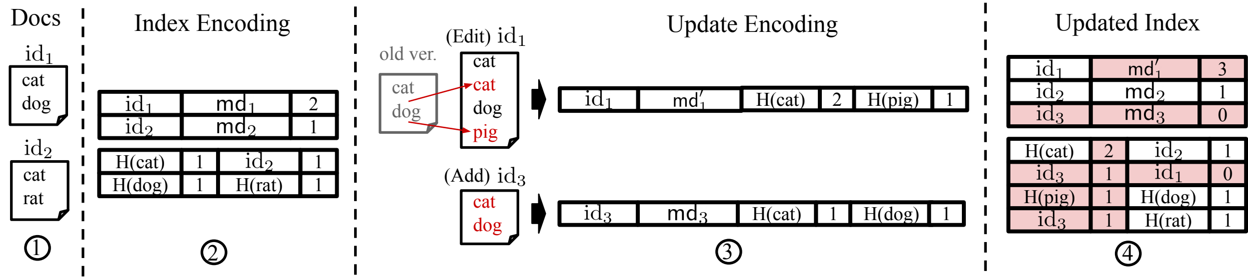


Figure 5.3: Four stages of basic encrypted size-locked indexing. ①: documents to be indexed; ②: primary index on the documents; ③: document updates and their encoding; ④: primary index merged with the encoded updates, with new entries shown in red.

5.4.1 A Size-Locked Index

Primary index construction. Our encoder takes as input a standard forward index that includes document identifiers, the keywords that appear in them, their term frequencies for that document, and metdata about each document. The encoder operates deterministically and outputs a byte string

$$\langle n \rangle_4 \parallel \texttt{fwd} \parallel \texttt{inv},$$

where $\parallel$ denotes string concatenation, $\langle n \rangle_4$ is an encoding of the number of documents (we use a four byte representation in our implementation), `fwd` is an encoded forward index byte string, and `inv` is an encoded inverted index byte string. Here and below, for a value v we use the notation $\langle v \rangle_k$ to mean some canonical k-byte encoding of v. We use two configurable parameters: W is the number of bytes used to encode identifiers and M is the number of bytes of per-file metdata allowed.

We explain `fwd` and `inv` next, starting with `inv`. The goal is to ensure that they together: encode the metadata and postings, allow efficient reconstruction of an in-memory inverted index, and have total byte-length equal to a fixed function of n and N. Because we support updates, meaning document insertions and edits, the values of n and N will change over time, and we will make them monotonically increasing as explained further below. Our primary index will achieve size

$$(W + W/2 + M) \cdot n + (W + 1) \cdot N$$

bytes. (We assume that W is a multiple of two.) The first term is the size of `fwd` and the second term is the size of `inv`.

The inverted index. To start, consider a standard inverted index which would consist of a sequence of posting lists each of the form

$$w \; , \; \mathsf{df}(w, D) \; , \; (\mathsf{id}_1, \mathsf{tf}_1) \; , \; \ldots \; , \; (\mathsf{id}_\ell, \mathsf{tf}_\ell)$$

where $\ell = \mathsf{df}(w, D)$ is the document frequency and $\mathsf{tf}_i = \mathsf{tf}(w, d_i)$ is the term frequency of w in document d_i whose id is id_i. We start by compactly encoding keywords w as short, W-byte hashes, denoted by $H(w)$. We use a cryptographic hash for H and then truncate appropriately. This replaces variable-length keywords with fixed length hashes. Of course for small W there can be collisions in which case we simply merge the colliding posting lists. This happens rarely enough to not degrade search accuracy much.

Next we make term frequencies compactly represented by one-byte values. This will make BM25 calculations coarser, but still enable sufficiently accurate ranking. Specifically we let $\widetilde{\mathsf{tf}}_i$ be a rounding of tf_i to the nearest value expressible in the form $a2^b$, where a, b are four-bit non-negative integers. We can thus encode each $\widetilde{\mathsf{tf}}_i = a \| b$ as a single byte. We dispense with storing document frequency $\mathsf{df}(w, D)$ entirely, as this is equal to the length of the posting list.

This is still not yet sufficient, because concatenating together the resulting posting lists of the form

$$H(w) \; , \; (\mathsf{id}_1, \widetilde{\mathsf{tf}}_1) \; , \; \ldots \; , \; (\mathsf{id}_\ell, \widetilde{\mathsf{tf}}_\ell)$$

would give a byte string of length dependent on the number of keywords in the search index. We therefore perform a trick to make a posting list of length ℓ have exactly $(W+1)\ell$ bytes. We enforce domain separation between hashes and document identifiers by fixing the top bit of hashes to be one, i.e., replacing $H(w)$ with $H(w) \vee 10^{8W-1}$ and fixing the top bit of all document identities to zero. Then, we remove id_1 in each posting list, making it implicit. We will store information in

`fwd` to be able to recover it during decoding. Thus our posting list ends up encoded as

$$(H(w) \vee 10^{8W-1}) \parallel \widetilde{\mathsf{tf}}_1 \parallel \mathsf{id}_2 \parallel \widetilde{\mathsf{tf}}_2 \parallel \cdots \parallel \mathsf{id}_\ell \parallel \widetilde{\mathsf{tf}}_\ell$$

and `inv` is the concatenation of the individual posting lists.

The forward index. We now turn to the forward index `fwd`. This includes an entry for each document that includes the document identifier, the metadata, and the number of keywords that first appear in that document. Let New_i be the set of terms newly introduced by document id_i and let $ct_i = |\mathsf{New}_i|$, encoded as a $W/2$-byte integer. For this to make sense, imagine the n documents are processed one by one, and those terms in the i-th document but not in the previous $i-1$ documents are in New_i. Then for each document we include in the forward index the byte string $\mathsf{id}_i \parallel \mathsf{md}_i \parallel ct_i$. Then to enable the decoder to fill-in the omitted first identifiers, we sort the posting lists within `inv` so that the ct_1 posting lists associated to keywords in New_1 appear first, then the ct_2 lists for New_2, and so on. Thus during decoding we know that the first $ct_1 = |\mathsf{New}_1|$ posting lists should have added id_1, the next $ct_2 = |\mathsf{New}_2|$ have added id_2, and so on.

The ① and ② stages in Figure 5.3 illustrate how the full primary index looks with two documents

Update encoding. We want to support adding documents as well as editing documents in a way that changes the keywords (including repetitions). Updates need to be handled carefully, as otherwise subtle injection attacks arise like those we described against Lucene in the previous section. Concretely, suppose we handle in distinct ways adding a new keyword "cat" to a document versus adding another repetition of an existing keyword "dog" to that document. A natural approach here would be, in the former case, to add a new posting for "cat" and, in the latter case, just update the term frequency for "dog". Consider if our index, via its size, leaks the exact number of postings in the search index before an update (N) and after an update (N'). Then an adversary can perform an injection attack to infer plaintext contents by inserting different values until they see $N' = N$, indicating they inserted a keyword matching an existing one.

We therefore make the total number of postings in the index monotonically increasing as up-

dates occur. Thus we will have an invariant that $N' = N + m$ where m is the number of modifications made (additions or edits). This has performance implications (storage and bandwidth), but is required to avoid injection attacks.

We also use a "lazy" update merging approach to make updates fast. Whenever a document is added or edited, our construction will encode and encrypt the changes so that they can be appended to a list of outstanding updates on the server. The updates will be downloaded and merged into the main index upon the next search. Lazy merging means we can perform updates without downloading the search index, which is good for performance in when workloads are update heavy.

In detail, an update consists of a document identified by id, metadata md, and list of m updated term, term-frequency pairs. We use the encoding

$$\mathsf{id} \parallel \mathsf{md} \parallel H(w_1) \parallel \widetilde{\mathsf{tf}}_1 \parallel \cdots \parallel H(w_m) \parallel \widetilde{\mathsf{tf}}_m,$$

where $\widetilde{\mathsf{tf}}_i$ is the rounding of $\mathtt{tf}_i$ to fit into a single byte as described earlier. For an update with m postings, this encoding will be exactly $W + M + (W + 1)m$ bytes. Thus the size leaks the number of changes to the postings list. It does not leak whether a new keyword was added or an existing keyword updated. The stage ③ in Figure 5.3 shows how to encode the update on existing or new documents.

To perform a merge, the existing posting list in `inv` is located, and we add all m postings to it. Note that we do *not* delete prior postings: in the case that there are two postings for the same keyword and document pair, we set one of these to the correct term frequency and make the remaining term frequencies zero. These latter postings are now essentially padding to mask whether a new term was added to the document. After processing an update with m postings, we have that `inv`'s size increases by exactly $(W + 1) \cdot m$ bytes. The stage ④ in Figure 5.3 show the primary index merged with updates on both existing and new keyword and document pairs.

$\mathsf{Update}(K, \mathsf{id}, \delta; \vec{C}_{up})$:

1: Client generates size-locked `inv` and `fwd` on id and δ
2: Client sends $\mathsf{Enc}_K(\langle 1 \rangle_4 \| \mathtt{fwd} \| \mathtt{inv})$ to server, and server appends it to $\vec{C}_{up}$

$\mathsf{Search}(K, q, i; \mathrm{EDB}, \vec{C}_{up})$:

4: Client downloads EDB and $\vec{C}_{up}$ from server
5: Client merges $\mathsf{Dec}_K(\vec{C}_{up})$ and $\mathsf{Dec}_K(\mathrm{EDB})$ into DB
6: Client sends $\mathsf{Enc}_K(\mathrm{DB})$ to server, and server stores it as EDB
7: Client returns the ranked results in the i page for q in DB

Figure 5.4: Basic encrypted size-locked index construction. EDB and $\vec{C}_{up}$ denote the encrypted index and the outstanding encrypted updates. The semicolon in protocol input separates the input to client (left) and to server (right).

5.4.2 Security Analysis

With our index encodings fixed, converting to an encrypted index is straightforward: just apply a standard authenticated encryption (AE) scheme to the concatenation of the index byte strings. We provide detailed pseudocode in Figure 5.4.

At any point in time, EDB at the server will consist of a encoded and encrypted index, along with possibly several appended update ciphertexts. To upload an update, the client uses our update encoding, encrypts the result, and uploads it to the server, where the ciphertext is appended to EDB. To perform a search, the current encrypted primary index as well as encrypted updates are downloaded, decrypted, and merged. The query is run against the index, and a new primary index is encrypted and uploaded. A key benefit of this approach is that we can use standard, fast authenticated encryption tools, such as AES-CCM [53].

We analyze security in two steps: In the first we use cryptographic analysis to show that the security of the underlying symmetric encryption together with size-locking guarantees that a server cannot learn more information than is provided via a simple *leakage profile*. Second, we discuss what the leakage profile reveals to adversarial servers.

Formal cryptographic analysis. We give an informal overview here and the details in Ap-

pendix B.1. The definitions follow closely prior formal models from those of symmetric searchable encryption (c.f., [40]) and we adopt their terminology, calling our security goal $\mathcal{L}$-adaptive security. Here $\mathcal{L}$ refers to a leakage profile that captures what is allowed to be leaked to the adversarial server by client messages to it. An update is only allowed to reveal the number of postings included and a search is only allowed to reveal the total number of documents and postings added thus far when a search occurs.

Our security model captures an adaptive adversary that observes all interactions with the server. (In our case this amounts to update ciphertexts and the main primary index ciphertext.) The model allows the adversary to adaptively choose inputs to insert into the index, thereby capturing active injection attacks. The only secret is the encryption key. The formalisation asks that the adversary cannot distinguish the server's view when receiving real encryption indexes from ones generated by a simulator that only receives the leakage specified by $\mathcal{L}$.

We prove in Appendix B.1 that our encrypted size-locked index achieves $\mathcal{L}$-adaptive security assuming the underlying encryption scheme is secure (in the standard sense of indistinguishability under chosen plaintext attack), for an $\mathcal{L}$ that intuitively reveals the number of postings in updates, and the number of documents added to the system. A benefit of our approach is that ours proof is very simple, and in fact most of the work is done in the encoding analysis.

Leakage-abuse analysis. The formal cryptographic analysis implies that only the leakage specified by $\mathcal{L}$ is revealed to an adversary. This leakage is strictly less, in a formal sense, than efficient DSSE schemes (which do not achieve the same level of functionality as our scheme, see Appendix B.2 where we discuss adapting existing SSE schemes to enable ranking). In particular $\mathcal{L}$ does not reveal the results patterns exploited by prior leakage abuse attacks [29, 79, 130, 162, 176]. We even hide the number of search results, completely preventing other attacks that get by with just that leakage including [22, 29]. Our leakage does reveal some information about updates (as do prior DSSE schemes), and in theory an adversary can use this to infer something about the magnitude of changes made to documents. But we are aware of no practical attacks known that

exploit this type of leakage.

Our leakage also is crafted to ensure resistance to injection attacks [29, 176]. The reason is that the leakage revealed via a sequence of updates is independent of whether those updates touch on keywords already present in the index.

Finally we note that in a deployment, a user may click on documents after a search, leading to the revelation of the *access pattern*. If a user requests every page of results and clicks on every result, then in principle the results pattern would leaked by prior DSSE schemes would also be revealed by our schemes. In practice users may be unlikely to actually do this, and whenever they opt not to click a result, it is not revealed in the access pattern, leading a false negative for an attack. Users may also click on additional documents unrelated to the search, further confusing attacks.

5.5 Partitioning Size-Locked Indexes

While providing stronger security than previous encrypted search systems, the size-locked index from the previous section may result in high bandwidth requirements for initializing a light client. In this section we introduce techniques to improve bandwidth at the cost of some additional, tunable leakage. The high-level idea is to *partition* indexes vertically (i.e., slicing postings lists) or horizontally (i.e., placing postings lists into buckets on a per-term basis), or both.

Vertical partitioning nicely matches the paginated nature of user-facing search while saving bandwidth, and introduces almost no additional leakage (concretely, once implemented properly it only leaks how many pages are being requested). Horizontal partitioning reflects the fact that not every posting list is needed to process a particular query, and provides additional bandwidth reduction, though at the cost of some leakage about keywords being searched.

Naive attempts at partitioning result in damaging leakage that enable attacks of the sort we identified against the Lucene-based solution earlier. Thus we introduce a size-locked versions of partitioning that carefully pad or truncate partitions according to fixed functions of the number of postings. As with our first construction from the previous section, our priority is security via

small leakage, and our constructions will allow for violations in correctness if necessary. We can however argue in a principled way that correctness is unlikely to be violated in practice due to typical text data distributions.

5.5.1 Vertical Partitioning

An insecure attempt. We begin with a straw proposal to highlight the subtleties and motivate our solution. Consider a modification of our scheme from the previous section which works by dividing every posting list into pages of k postings each, and then encodes all of the first pages together in one encrypted blob B_1, the second pages in another encrypted blob B_2, and so on. For a search that requests page pg, one downloads blob B_{pg} and looks up the corresponding term. Let us generously assume that updates are done by downloading all of the blobs, re-ordering lists, and then re-encoding and uploading all of the blobs. (This is impractical but good for security, which is our concern at the moment.)

This strategy is strictly more leaky than before, since the size of each blob is revealed. Even if the encoding size-locked to only reveal the number of postings in the blob, this revelation is exploitable by injection attacks. For example, an adversary can inject a document containing w and observe which level grows to learn the rounded document frequency of w.

This attack allows an adversary to recover statistics about the plaintext, such as if a particular term appears anywhere (i.e., if the postings list has positive length), so we conclude that this approach provides unacceptable security for our setting.

Overview of size-locked vertical partitioning. We propose a secure vertical partitioning algorithm that results in index partitions whose sizes are "locked" by the features of the dataset we are willing to leak.

Our strategy is to fix a capacity function $\mathsf{Cap} : \mathbb{Z}^+ \to \mathbb{Z}^+$ that determines a partition size $\mathsf{Cap}(N)$ for a corpus with N postings; We defer how this is done for the moment. We then maintain an invariant on our size-locked partitions, requiring that each vertical partition except the last one

hold exactly $\mathsf{Cap}(N)$ postings. This will completely mitigate the size leakage issue because the sizes are all determined by N only, and not the number of terms or how postings are distributed amongst the terms.

We now overview our construction. The vertical partitioning strategy works by splitting the index across a number $L = \lceil N/\mathsf{Cap}(N) \rceil$ of levels. Each level has associated to it an encrypted index $B_1, \ldots, B_L$ and an encryptd update cache $C_1, \ldots, C_L$, both stored at the server. Updates are performed lazily, by appending encoded and encrypted information to the first level cache C_1.

To search, one begins by downloading just the first level (B_1, C_1), decrypting and decoding them, and performing the search using whatever postings are available. To process any outstanding updates in C_1, the client uses a policy function (to be fixed below) which dictates the postings that should remain in the first level and the postings that will be evicted to the next level. These evicted updates are encrypted and appended to C_2. The number of evicted postings is exactly the number of new postings in the updates included in C_1. (Recall that we leak the number of postings included in an update.) This ensures the invariant for B_1.

If more results are requested, then additional levels are accessed as needed, and the process above is repeated. Thus an update of size m eventually results in what looks like m postings being moved from each level to the next.

To realize this approach, we need to fix a number of details, including specifying Cap and an eviction policy, as well as how to encode both the primary indexes (which require some changes due to levels) and the evictions.

Capacity function. Our goal is to select Cap so that the top-k postings for the entire corpus sum to about $\mathsf{Cap}(N)$ total postings. If Cap overshoots this target then extra postings can be opportunistically packed into the level, so the space is not wasted, but we would prefer not to in order to reduce bandwidth. If Cap undershoots the target, then not all of the top-k postings will fit, resulting in some searches not being properly handled. In any case, security does not depend on the accuracy Cap.

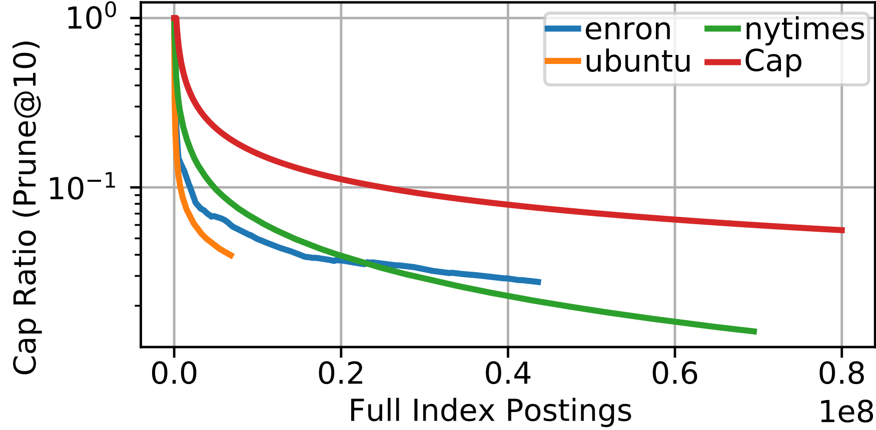


Figure 5.5: Comparing $\mathsf{Cap}(N)$ to N_{10}, the total number of postings in the top-10 of postings-lists across three datasets and for varying . We added documents in random order, measuring the ratio N_{10}/N after each addition. Heap's Law is apparent in the shape of each curve, and our choice of Cap proves conservative.

We would like to simply set $\mathsf{Cap}(N) = kt$, where k is the page size (e.g., $k = 10$) and t is the number of terms in the corpus. But we do not want to leak t which varies as new keywords are added to the document corpus. Fortunately, on typical text t behaves in a predicable fashion known as *Heap's Law* (c.f., [103], Chapter 5.1.1) which predicts that $t \approx \alpha \cdot T^{\beta}$, where T is the total number of occurrences of terms in the corpus (i.e., the sum of all of the term frequencies across all document/term pairs) and α, β are constants satisfying $10 \leq \alpha \leq 100$ and $\beta \approx 0.5$. Following this, we conservatively take $\alpha = 20$ and $\beta = 0.5$, and use the bound $N \leq T$, and set the capacity function to be

$$\mathsf{Cap}(N) = \min\{N \ , \ 20 \cdot k \cdot N^{0.5}\} \ .$$

To estimate how this will perform in practice, we compare this function to the size required to store the top-k postings for $k = 10$ and the datasets we use for evaluation in Section 5.7. See Figure 5.5. We find that the estimate provides a large margin above what is required. This slack is both from our conservative choice of constants α, β, and the fact that many postings lists contain fewer than 10 postings.

Primary vertical partition encoding. We encode the first partition differently from the rest, which

are all encoded the same. The difference will be that the first partition will hold the entire forward index associating identifiers to metadata, as well as document frequencies for each term, stored in a size-locked way. The high-level structure of the first partition is again $\langle n \rangle_4 \,\|\, \texttt{fwd} \,\|\, \texttt{inv}$, where fwd is encoded exactly as before. We change the encoding of inv in order to explicitly store document frequencies (recall these were implicit before; now, since we do not have entire postings lists they must be stored explicitly). We can not naively write the document frequencies into a separate table due to size-locking, as we should not reveal the number of terms.

We encode inv by processing postings lists in order as before, with the same encoding, except that each posting includes an extra byte. We process the list for term w to compute its document frequency df, and encode it in as many bytes as needed, in little-endian order; call these bytes $\mathsf{df}_1, \mathsf{df}_2, \ldots$, where the unused bytes are implicitly zero. We then encode the posting list for w as

$$(H(w) \vee 10^{8W-1}) \,\|\, \widetilde{\mathsf{tf}}_1 \,\|\, \mathsf{df}_1 \,\|\, \mathsf{id}_2 \,\|\, \widetilde{\mathsf{tf}}_2 \,\|\, \mathsf{df}_2 \,\|\, \cdots \,\|\, \mathsf{id}_\ell \,\|\, \widetilde{\mathsf{tf}}_\ell \,\|\, \mathsf{df}_\ell.$$

Note that while the document frequency byte encoding varies in length, it will not be leaked, because we pad each posting regardless. We will also always have enough space for the df encoding: In a list of length ℓ we will have[2] ℓ bytes to encode $\mathsf{df} = \ell$. This completes the description of encoding the first partition. For an index with W-byte identifiers, M-byte metadata, n documents, and N total postings, the encoding will have length

$$(W + W/2 + M) \cdot n + (W + 2) \cdot \min\{N, \mathsf{Cap}(N)\}.$$

We next describe how the other partitions are encoded. The input here is partial posting lists of the form

$$H(w),\ (\mathsf{id}_1, \widetilde{\mathsf{tf}}_1),\ \ldots,\ (\mathsf{id}_\ell, \widetilde{\mathsf{tf}}_\ell)\ .$$

Our encoder sorts the terms by their document frequency (which is available as we assume the first partition has been loaded). It then encodes this a posting list as

$$(\mathsf{id}_1 \vee 10^{8W-1}) \,\|\, \widetilde{\mathsf{tf}}_1 \,\|\, \mathsf{id}_2 \,\|\, \widetilde{\mathsf{tf}}_2 \,\|\, \cdots \,\|\, \mathsf{id}_\ell \,\|\, \widetilde{\mathsf{tf}}_\ell.$$

2. Formally, we use that $256^\ell - 1 > \ell$ for all positive integers ℓ.

In words, the first identifier has its top bit set, and rest are encoded with their rounded term frequencies (and top bits cleared) exactly as before. This encoding does not include the hashed terms themselves. The decoder can infer this association using the document frequencies from the top partition (the first posting list corresponds to the term with highest document frequency and so on). The byte-length of this encoding is $(W+1) \cdot \mathsf{Cap}(N)$. The last partition may contain fewer than $\mathsf{Cap}(N)$ postings, in which case the length is appropriately adjusted.

Updates and evictions. Updates are encoded exactly as in the basic scheme before they are appended to the top-level update cache C_1.

We opted for a simple greedy policy to determine which postings are stored in each tier. First, fix an ordering over keywords that matches the ordering in which they were added to the corpus, and breaks ties (due to being added by the same document) arbitrarily. Then we loop over keywords in that order, adding to their posting list the next highest posting by BM25 for that keyword. This round robin approach ends when $\mathsf{Cap}(N)$ postings have been processed. To get the next level's postings, remove all the postings from the first level from consideration, and otherwise repeat the process. BM25 ties within a list are broken arbitrarily.

When processing updates incrementally, the eviction policy can be evaluated with just the current level and the outstanding updates for that level. In case of two postings for the same document and term, we keep one with (updated) term frequency and zero out the term frequency of the other. Then we combine the new and old postings to get a set of $\mathsf{Cap}(N) + m$ eligible postings, and perform the round-robin approach using those postings. The remaining m postings will then become the postings shifted to the next lower level. We encode each of those postings as a $(2W+1)$-byte string $H(w) \parallel \mathsf{id} \parallel \widetilde{\mathsf{tf}}$, and concatenate them together to form a single string of length $(2W+1) \cdot m$.

We provide the pseudocode for the vertically partitioned encrypted size-locked index construction in 5.6.

Leakage profile. The leakage profile $\mathcal{L}$ of this construction is exactly the same as before except the

$\mathsf{Update}(K, \mathsf{id}, \delta; (\vec{C}_1, \ldots))$:

1: Client generates size-locked `inv` and `fwd` on id and δ
2: Client sends $\mathsf{Enc}_K(\langle 1 \rangle_4 \| \mathtt{fwd} \| \mathtt{inv})$ to server, and server appends it to $\vec{C}_1$

$\mathsf{Search}(K, q, i; (B_1, \ldots), (\vec{C}_1, \ldots))$:

4: Client initializes DB as empty
5: **for** $l = 1, \ldots$ **do**
6: Client downloads B_l and $\vec{C}_l$ from server
7: Client merges $\mathsf{Dec}_K(\vec{C}_l)$ and $\mathsf{Dec}_K(B_l)$ into DB
8: Client looks up for ranked results in the i page for q using DB
9: Client vertically partitions DB: encrypts and sends (1) the top $\mathsf{Cap}(N)$ postings as B_l, and (2) the remaining postings as update to $\vec{C}_{l+1}$, to server
10: **if** no word in DB has more results in the i-th page **then** break
11: Client returns the search results

Figure 5.6: Vertically partitioned encrypted size-locked index construction. B_l and $\vec{C}_l$ denote the encrypted vertical partition and outstanding updates at level l.

requested number of levels is also leaked. The sizes of all of the levels B_i are determined precisely by the total number of documents and postings, via the capacity function and the size-locked encodings. As described above, the update caches reveal only the number of postings included in updates. Notably, all this means that injection attacks fail.

In deployment, a natural approach will be to request more levels when a user requests further results. The server will therefore learn that more results were desired, which could reveal some information about the number or quality of postings in the previous tiers. We do not know how this could be exploited by an adversary (it would require a model of user behavior), and it is not a threat model that has been treated in prior leakage abuse attack research. Even so, implementations could potentially obfuscate even this information by automatically fetching subsequent levels automatically regardless of user behavior.

5.5.2 *Horizontal Partitioning*

Our next extension is simple: we just horizontally partition the keyword space and build separate size-locked indexes for each of partition. At a high level, we will assign each term to one of P buckets via a pseudorandom function (PRF), and run P parallel versions of either our basic or vertically-partitioned constructions. This roughly reduces the search bandwidth by a factor of P (for single-term queries), at the cost of extra leakage (because touching the buckets limits the possibilities for the query or update). Techniques similar to this have been studied in the information retrieval literature [113, 173] for load balancing and parallel index lookup, and also recently for DSSE schemes [42].

We implement this by having the user derive an additional key K' for a PRF (e.g. HMAC-SHA256), and assigning a term to a bucket via $p = \mathrm{PRF}(K', w) \bmod P$. For updates, the new postings are assigned to their respective partitions, and an update is issued to each partition with the assigned postings. A query for a single term is run against one partition, and a multi-term query is run by executing the query against the relevant partitions and then ranking the results together (once the partitions are downloaded, we can compute BM25 on the entire query).

Leakage analysis. The leakage profile here is more complicated to describe formally, but intuitively simple. Via the PRF, there is a random-looking association between terms and partitions. When an update is issued, the server learns how many postings and documents are being added to the partitions determined by the random mapping. A search reveals which partitions are being touched along with the number of levels requested (if also using vertical partitioning). The result pattern is otherwise hidden. A formal version of this leakage is given in Appendix B.1.

The possibility of leakage-abuse and injection attacks against this construction depends on the number of partitions. For a single partition, it reduces to the previous construction. If we set the number of partitions so large that they all contain a single term, then the construction will be vulnerable to document injection attacks: the adversary can insert documents with single terms and learn which term is contained in each partition. Subsequently it can identify searches.

With a modest number of (say, 10) partitions, it is less clear how to attack the system in a practical setting. Via document injection, an adversary can eventually learn the mapping of some chosen terms to partitions, so let us pessimistically assume that the map is fully known. Then for single-term queries, the adversary can determine which bucket the query was in. For a small number of partitions, the buckets will be large. If an adversary however knows that the query takes one of a few values (say querying “april” versus ”june”), then it can identify which query was issued with good probability. But if the query is not from a small set, then it may be difficult to identify it amongst the large number of terms assigned to the bucket.

5.6 Secure Binary Index Lookup

5.6.1 Secure Binary Index Lookup without Partitions

The straight-forward approach to search with the encoded index is to first decode it into the in-memory index, consisting of the inverted index and the forward index, and then to lookup the postings of the query keywords in the inverted index to identify the documents. Fully decoding the encoded index file, however, takes time linear in the total number of postings in the index, and, therefore, such an approach cannot scale as the size of the document collection increases.

An optimized approach is to store a lookup table, mapping from the word hashes to the offsets of the bytestrings for their posting lists in the encoded index. Thus, we can first identify the offsets of the posting lists of the query keywords, and then skip to those offsets to decode only the necessary postings. Even better, we can order the word hashes in the lookup table by their values, and use binary search to quickly identify the offsets at query time. The decoding cost will be linear to the summation of document frequencies of the query keywords, which is asymptotically much smaller than the index size. The extra space overhead is linear to the number of words covered in the lookup table.

The major problem with naively applying this optimization is that including all the words in the

index in the lookup table will leak the total number of words in index, and, with file-injection capability, the attacker can learn whether a word is in the index or not. To resolve such size leakage, we construct the lookup table based on the estimated number of words in the document collection, instead of the exact number, following Heap's law. Heap's law states that, for a document collection with N postings, the number of unique words is roughly $\alpha \cdot N^{\beta}$, and we conservatively choose $\alpha = 50$ and $\beta = 0.5$ to let the estimated number of words be an overestimate. In the rare case in which the estimated number of terms is smaller than the exact number of terms, some words will not appear in the lookup table, and we fall back to the straight-forward approach, *i.e.*, decoding the entire index. Hence, the size of the lookup table is $50(2W + 4) \cdot N^{0.5}$ because we store, for each word hash of W bytes, the W-byte identifier of the first document that introduces the word and 4 bytes offset, and the size only leaks total number of postings in the index.

5.6.2 Secure Binary Index Lookup with Partitions

The optimization for fast keyword search using a size-locked binary lookup table (Section **??**) can be adapted for both vertical and horizontal partitioning.

For vertical partitioning, we use the same estimated number of words from Heap's Law over the entire document collection to construct the binary lookup tables for all vertical partitions: for each partition, we fill in the lookup table with the word hashes, together with their offsets, from the partition, as well as dummy entries, up to the estimated number. The first partition requires $W + 4$ bytes per word to help identify both the first document for the word and the offset of the posting list, while the subsequent ones only need 4 bytes per word for the offset of the posting list.

For horizontal partitioning, the expected number of words in each partition is $\frac{1}{P}$ of the total number of words, and we divide the estimated number of words from Heap's Law over the entire document collection by P to determine the lookup table size for each horizontal partition. Within each horizontal partition, the construction of the lookup table across vertical partitions is the same as described above.

5.7 Evaluation

We experimentally evaluate our proposed constructions, particular to answer the questions: (1) Are size-locked index-based constructions feasible in practice, in terms of bandwidth and end-to-end processing time? (2) Do they provide accurate ranked search results, compared to plaintext searches?

Experimental setup. We implemented the schemes from Sections 5.4 and 5.5 in Python, and plan to release our prototype as a public open-source project. It uses the PyCryptodome library's implementation of AES-CCM-128 for authenticated encryption, and HMAC-SHA256 for a PRF. For all symmetric cryptographic tools, we fix the key size to be 128 bits. We use BLAKE2b for the hash function. As for the encoding parameters, we fix identifier/term hash with $W = 4$ bytes (giving a term collision rate less than 10^{-4} for our target datasets), the document metadata consisting of $M = 14$ bytes (6 bytes for the name; 2 bytes for encoding the length of the document; 2 bytes for encoding the number of words in the document; 4 bytes for a timestamp).

We refer below to our full index scheme as FULL, the vertically partitioned scheme as VPART, and the scheme using a combination of vertical and horizontal partitioning as VHPART. We vary the number of horizontal partitions between 10 and 1,000, and utilize the Cap function described in Section 5.5.1 to bound the sizes of all vertical partitions.

We run experiments with one machine as client and the Amazon S3 as the storage service provider. We test as the client with a variety of machines: (i) iMac Mojave with Intel quad-core 4.00GHz CPU and 16 GB RAM; (ii) MacBook Pro with Intel dual-core 2.70GHz CPU and 8 GB RAM; and (iii) AWS EC2 instance with Intel dual-core 2.30GHz CPU and 8 GB RAM. We test under different network bandwidth conditions: LOW: home wifi router with 12-25Mbps downstream and 2Mbps upstream bandwidth; MED: department wifi router with 80-160Mbps downstream and 250Mbps upstream bandwidth; and HIGH: AWS data-center network with 700Mbps downstream and upstream bandwidth.

Dataset	Size (MB)	$|D|$	$|W|$	$|DB|$	Lucene
Ubuntu	116	1,038,324	95,120	6,674,297	16.8
Enron	1,700	517,400	252,904	43,358,907	101.8
NYTimes	846	300,000	101,470	69,665,021	115.1

Figure 5.7: The datasets used for our experiments. The first column is size of the datasets in MB, followed by: the number of documents $|D|$, the vocabulary size $|W|$, the number of unique word/document pairs $|DB|$, and the size in MB of a standard Lucene index for the corpus.

Dataset	FULL	VPART		VHPART-10		VHPART-100	
		L1	L2	L1	L2	L1	L2
Ubuntu	45.7	16.3	3.67	5.76	0.34	0.90	0.04
Enron	217.7	17.3	9.10	6.50	0.90	2.19	0.09
NYTimes	340.4	17.8	11.50	4.85	1.15	3.05	0.12

Figure 5.8: Search bandwidth cost (in MB) after adding all documents in corpus. L1/2 stands for level 1 and 2 in vertical partitioning.

Datasets. We evaluate with three public datasets: the public Enron email collection (Enron) [2], the Ubuntu IRC dialogue corpus (Ubuntu) [4], and a collection of New York Times news articles (NYTimes) [3]. All of these have been used in previous works on either searchable encryption [29, 79, 176] or information retrieval [101, 155]. Summary statistics for the datasets appear in Figure 5.7. Note that the Enron and Ubuntu datasets are raw textual data without preprocessing, and, for indexing, we preprocess the text with stop word removal [103] and Porter stemming [129]. The NYTimes dataset has already been preprocessed into terms, which we directly use for indexing.

Search bandwidth cost. We start by measuring the search bandwidth cost of the constructions, after adding every document and merging them into the primary indexes. We focus on the *cold-start* setting, where the client starts with no state locally.

The search bandwidth costs of the constructions are shown in Figure 5.8. For FULL, the entire index needs to be downloaded for a fresh search, and our specialized encoded index leads to bandwidth cost smaller than the (uncompressed) document corpus sizes by a substantial amount,

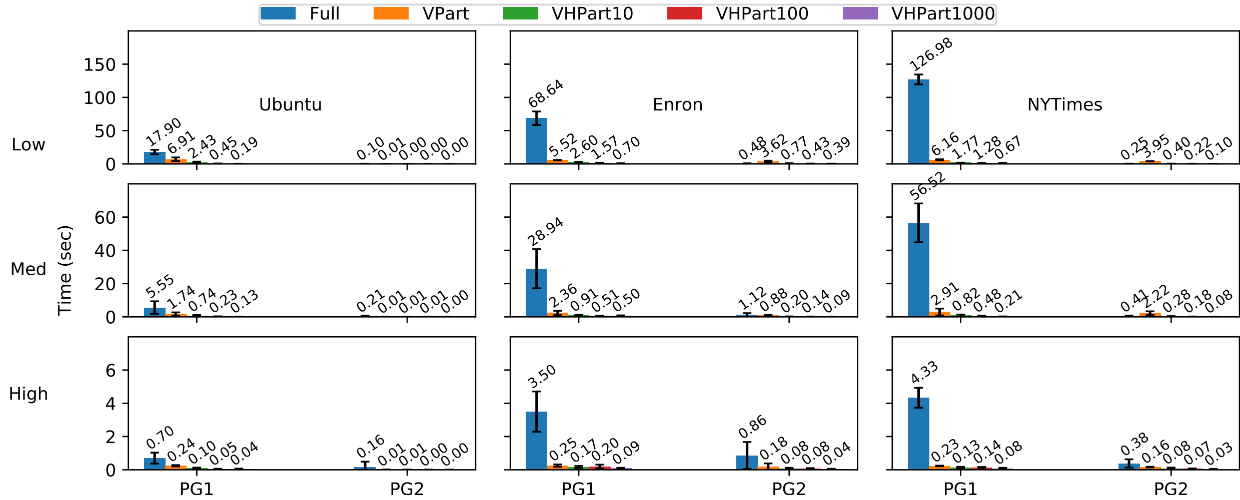


Figure 5.9: End-to-end search time. The rows LOW, MED and HIGH indicate downstream network bandwidth at levels of 12-25Mbps, 80-160Mpbs and 700Mbps, respectively. PG-1 and PG-2 indicate performance of searching for the first and second pages. The error bar indicates the standard deviation.

i.e., 12.8% (Enron), 39.4% (Ubuntu), and 40.2% (NYTimes) of the size of the respective corpus. For small-to-moderate datasets, e.g., 10% of the datasets, the bandwidth cost would be 22.6MB (Enron), 4.6MB (Ubuntu) and 34.1MB (NYTimes), respectively. For very large datasets, e.g., the full corpus, FULL would require too much bandwidth for many clients.

Vertical partitioning substantially reduces bandwidth (Figure 5.8) for searches handled by the first page or two of results (the common case). The bandwidth cost for searching the first page, which can be handled by the first level, is smaller by 12.6x (Enron), 2.8x (Ubuntu), and 19.1x (NYTimes), compared to that of FULL. Since all levels beyond the first are of the same size, we just report on the sizes of the second partition, which is substantially smaller than the first since it contains just postings and not the forward index.

Horizontal partitioning can easily be used to reduce bandwidth costs further, trading off increased leakage for performance. In Figure 5.8 we show the average size of the first and second levels with $P = 10$ and $P = 100$, respectively. In particular, for $P = 100$ and Enron, searching for the first page only downloads 2.19MB data, and query for subsequent pages at most downloads 0.09MB data per page.

End-to-end search time. We now turn to analyzing the end-to-end search performance. We focus on the end-to-end search time in the steady state, *i.e.*, after all documents of the corpus have been added and merged into the primary index, and we use 30 single-keyword queries (Appendix B.3) covering a wide range of document frequencies for each corpus, and report the averages over the 30 queries. We evaluate the search performance with one local client connected to the remote AWS S3 service that holds encrypted search indices, over three levels of network conditions: (i) LOW (12-25Mbps); (ii) MED (80-160Mbps); and (iii) HIGH (700Mbps).

The results are shown in Figure 5.9. First, when the network bandwidth is very limited, *i.e.*, LOW, FULL has very large end-to-end search time for the first page results, *i.e.*, averaging at 68.64 (Enron), 17.9 (Ubuntu) and 126.98 (NYTimes) seconds, respectively. VPART alone can reduce the search time to 5.52, 6.91 and 6.16 seconds for the three datasets, respectively, and VHPART with 1000 partitions, can further reduce the search time to 0.7, 0.19 and 0.67 seconds. When the network bandwidth is abundant, *i.e.*, HIGH, the end-to-end time of FULL reduces to 3.5, 0.7 and 4.33 seconds for the three datasets, and the time for VPART and VHPART are all way below 1 second.

When searching for results in the second page, the client does not need to download the index that has been downloaded already for the first page. Therefore, FULL only needs to lookup for the second page results locally without downloading anything, and, even under LOW network conditions, the end-to-end time are small, *i.e.*, 0.48, 0.1 and 0.25 seconds, respectively. VPART and VHPART, on the other hand, may need to download subsequent vertical levels to identify enough results for the requested page, and, under the LOW network conditions, the end-to-end times of VPART are 3.62, 0.007 and 3.95 seconds for the three datasets, while those of VHPART with 1000 partitions are 0.39, 0.002 and 0.1 seconds.

Effect of dataset size on the end-to-end search time. It is not uncommon for real-world users to upload a much smaller amount of documents to the cloud, than the three datasets that we adopted. To fully understand the performance of these constructions for datasets of smaller scale, we eval-

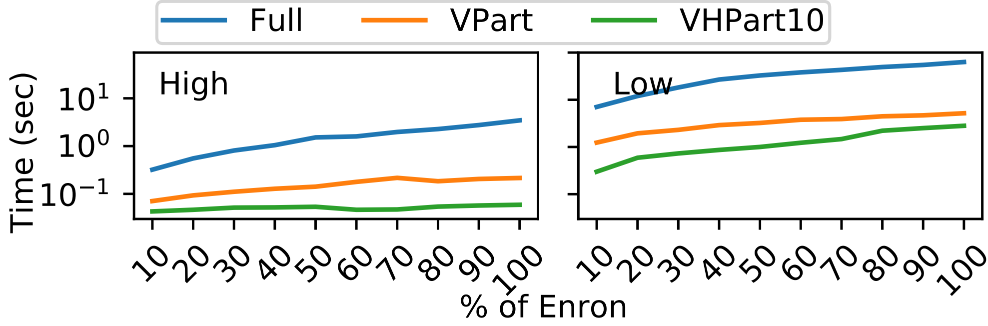


Figure 5.10: End-to-end search time for the first result page, with various percentage of the Enron dataset added to the index. LOW and HIGH indicate downstream network bandwidth at levels of 12-25Mbps and 700Mbps, respectively.

uate the end-to-end search time when a variety of percentage of Enron gets added to the search index. In particular, for each target percentage, we randomly select that percentage of documents from the Enron dataset, and add them to the empty index. Finally, we merge the index updates into the steady-state index for the end-to-end search evaluation.

The results are shown in Figure 5.10. The key takesway is that: when network bandwidth is abundant, *i.e.*, HIGH, the FULL construction can achieve acceptable end-to-end search time, *i.e.*, < 1 second, for datasets of small scale, *i.e.*, less than 30% of Enron; when network bandwidth is limited, *i.e.*, LOW, we can switch to VHPART with mild partition number, *i.e.*, 10, to achieve acceptable performance for up to 50% of Enron.

Search quality Recall that our constructions provide approximate BM25 scoring, and also have occassional hash collisions which could degrade search quality. More broadly we can evaluate search quality by comparing to a baseline of using BM25 [136] as per plaintext search. For a query q with result being an ordered list of documents $\mathcal{R}$, we measure the overall search quality using *normalized discounted cumulative gain* (NDCG) [80, 103], which aggregates the scores of the results, with more weights on the earlier ones, *i.e.*,

$$\mathsf{NDCG}(q, \mathcal{R}) = \frac{1}{\mathsf{IDCG}(q, |\mathcal{R}|)} \sum_{i=1}^{|\mathcal{R}|} \frac{2^{\mathsf{BM25}(q, \mathcal{R}_i)} - 1}{\log(i+1)},$$

where $\mathsf{IDCG}(q, k)$ is the normalization factor, calculated from the optimal ranking of the top-$|\mathcal{R}|$ results for q, to make $\mathsf{NDCG}(q, \mathcal{R}) \in [0, 1]$. Higher NDCG indicates better search quality, relative

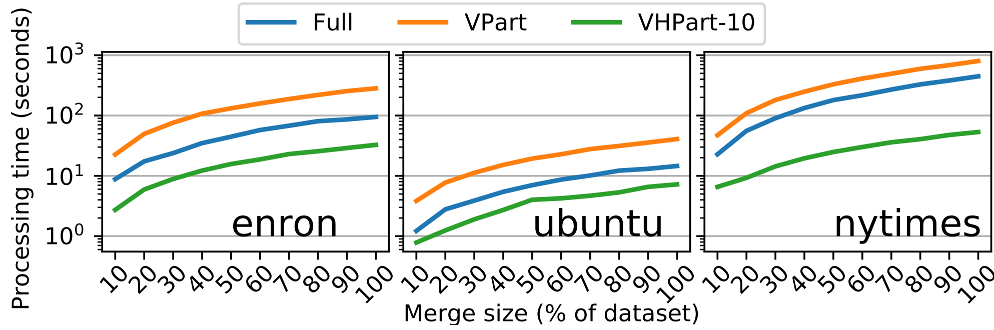


Figure 5.11: Processing time to lazily merge as a function of the fraction of the document corpus inserted before performing the merge.

to BM25 plaintext search. We can average the NDCG's of multiple queries as the overall search quality of a system.

Considering just the top 10 results, the NDCG over 50 random searches for all our techniques never drops below 0.9985. This means we match the search quality of state-of-the-art plaintext search systems for the first page of results. In our experiments only less than 10^{-4} term collisions arose, and these never impacted search quality calculations given how unlikely it is to choose such terms.

Update performance. We now turn to assessing the performance of our lazy merging technique for updates. We first measured the total amount of data transferred between client and server triggered by adding a new document to the dataset. For FULL, VPART and VHPART, the update size is just a small fixed overhead plus the document size (in terms of the number of keywords), and the average update bandwidth costs of three constructions are roughly the same, *i.e.*, around 0.1KB, 0.5KB and 1.2KB for Ubuntu, Enron and NYTimes, respectively. The average end-to-end time varies based on the network upstream bandwidth, and it is around 30ms and 80ms for HIGH and LOW, respectively. Horizontal partitioning requires more time in general because the extra computations on dividing the update into P buckets.

Searches that occur with outstanding updates will require first merging the update before completing the search. We are unaware of good datasets indicating how often searches occur relative to

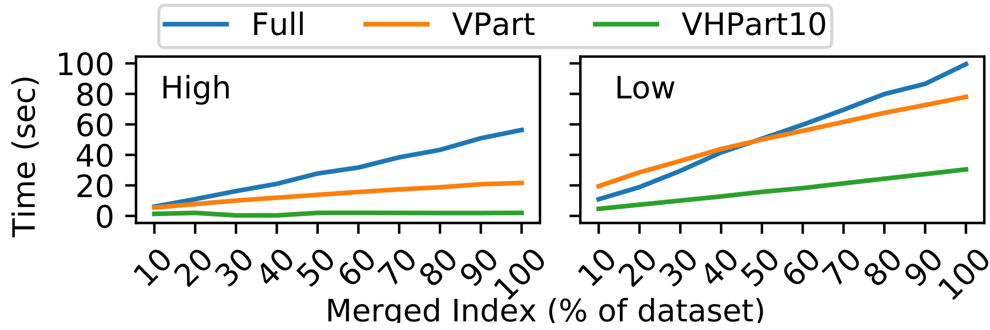


Figure 5.12: End-to-end search time, right after adding 10 documents, when various percentages of the document corpus have been added and merged.

document updates, so we consider two distinct cases: (I) the very first search after adding a large number of documents to an empty index; and (II) the very first search after adding a small number of documents to an merged index.

For case (I), we measure the total processing time on the very first search after different percentages of the dataset have been updated without merging. That is, we insert some fraction of documents before the first search is performed. The results are shown in Figure 5.11. This shows that merge times can be signficant (on the order of several seconds) when searches only occur after adding tens of thousands of documents. In practice, this can be easily mitigated at the cost of some bandwidth by periodically having the client perform a merge to ensure the backlog of updates does not get so large.

For case (II), we measure the total processing time on the very first search, right after adding 10 documents to the index that has already merging different percentages of the dataset. That is, we insert some fraction of documents to the index, perform a search (to have all the updates merged into the steady state), add 10 new documents to the index, and, finally, perform a search, whose time we measure and report. The results are shown in Figure 5.12. Compared to those in case (I), the search times in case (II) are smaller because the amount of updates to be merged is smaller, but downloading the merged index will bottleneck the performance as the index size increases, *i.e.*, merged with more documents, especially when the network bandwidth is LOW.

Comparison against DSSE. For comparison, we also implement a state-of-the-art forward-private DSSE based on the Diana construction [25]. We extend it with metadata and BM25 scoring, as well as making it work for stateless clients. As the scheme relies on client-side counters, we encrypt that client state and store it at the server with optimizations. We call the resulting scheme CTR-DSSE. Details are given in Appendix B.2. We note that this approach achieves much weaker security than our scheme, but it will serve to highlight trade-offs in performance.

As expected CTR-DSSE mostly outperforms our approach on search bandwidth and computation, as DSSE schemes target search costs that scale with just the size of the posting list for the queried keyword, with the average and maximum (for the most common keyword) bandwidth cost around 0.01MB and 0.5MB for the three datasets. Note that to show the first page of results, CTR-DSSE must download the entire posting list because the latter cannot be stored in sorted order in the dynamic case. Because posting list sizes vary greatly, the result is higher variance in performance for CTR-DSSE compared to our size-locked approach. The key takeaway here is that VHPART can easily achieve comparable performance to CTR-DSSE, while leaking less to the server, simply by a modest increase in the number of horizontal partitions.

The update performance of CTR-DSSE is more complicated as the bandwidth is dominated by the size of the client states, which increases as more documents are added. Updates early on will be fast, for example the best case update bandwidth usage for the three corpuses are 1.1KB, 0.2KB, and 2.4KB for Enron, Ubuntu, and NYTimes. The worst case occurs when adding the final document to the dataset, which results in update costs of 4 MB, 1.5 MB, and 1.5 MB respectively. In general the update bandwidth will be significantly better for encrypted indexing schemes, and in update-heavy workloads this will make a significant difference in overall bandwidth utilization. While in theory one might try some kind of lazy merging for CTR-DSSE, it's unclear how to do so while preserving search efficiency.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, we aim at improving the privacy-utility trade-offs in three popular cloud applications.

The first one is the federated SQL processing, and we presented a principled approach to closing the four most critical side channels: memory, instructions, timing, and output size. Our approach relies on a new primitive, hardware-assisted oblivious execution environments, new query operators that leverage differential privacy, and a novel privacy-aware query planner. Our experimental evaluation shows that Hermetic is competitive with previous privacy-preserving systems, even though it provides stronger privacy guarantees.

The second application is cloud data collection and analytics. We generalize the setting in previous work from single service to multiple services under local differential privacy. And we address two challenges in this more general setting: first, how to prevent privacy guarantee from being weakened during the joint data collection; second, how to analyze perturbed data jointly from different services. We introduce the notation of user-level LDP to formalize and protect the privacy of a user when her joined tuples are released. We propose mechanisms and estimation methods to process multi-dimensional analytical queries, each with attributes (in its aggregation and predicates) collected and perturbed independently by multiple services. We also introduce an online utility optimization technique for multi-dimensional range predicates, based on consistency in domain hierarchy.

The third application is end-to-end encrypted search. We show that the simple, folklore approach to end-to-end encrypted search: simply encrypt a standard search index, storing it remotely and fetching it to perform searches, is actually a promising direction, in terms of security, usability and efficiency. In particular, it reduces the leakage at search and update to the level where the file-injection attacks, which break almost all existing work, would not work. Moreover, it is

straight-forward to extend the index blob with necessary information for arbitrary query support, and, with proper optimizations, the end-to-end performance is practical in real-world setting.

6.2 Future Work

6.2.1 Shuffled LDP

A recent model of LDP, *i.e.*, *shuffled LDP* [14, 35, 58], introduces an extra data anonymization operation between the user and the service, and proves its amplification effects on the privacy of each user. In our setting, assuming all attributes, excluding the join keys, of the tuples are sensitive, and thus obfuscated, we can apply the shuffling operation on the perturbed tuples, *i.e.*, via a 3rd party shuffler, before forwarding them to each service. Note that the join keys do not neutralize the effects of shuffling because they themselves are anonymous. Thus, according to the results in [14, 35, 58], we can achieve the same level of user-level privacy with larger perturb budget, *i.e.*, ϵ, which will improve the aggregation utilities of all queries on the collected tuples.

6.2.2 Join with Star Schema

We focus on two-table primary-foreign-key join in Section 4.5, and can extend to join with the more complex star schema, where one service collects tuples with foreign keys to multiple primary-key tuples collected by different services. Such schema is common in business data warehouse. For instance, in Example 4.1.1, a third service could collect tuples on products, with attributes Price and Country, and a unique product id PID. The collected transaction tuples further contain the foreign key PID of the corresponding product. And an analyst wants to know the total sales from certain users on products from certain country, *e.g.*,

```
SELECT SUM(Amount) FROM Transaction
JOIN Product ON Transaction.PID = Product.PID
JOIN User ON Transaction.UID = User.UID
WHERE Country = "China" AND Age ∈ [20, 30].
```

We can extend τ-truncation to handle such relation, under ϵ-uLDP. The major adaptation is that we need to enforce that the same number of tuples with foreign-key, *i.e.*, transaction, match each pair of primary-key tuples, *i.e.*, $\langle$user,product$\rangle$. Thus, for n users and n products, with τ-truncation, we need to collect $n^2\tau$ transaction tuples. Thus, for joint aggregation, we can join the perturbed values from the three services, and aggregate on the joined values. It is possible to optimize such straight-forward extension for better efficiency and privacy management, which we leave as future work.

6.2.3 Adaptive Index Management System

In Chapter 5, we introduce three separate constructions to handle encrypted search, namely, FULL, VPART, and VHPART. They have different trade-offs on security, *i.e.*, resilience to file injection attacks, and efficiency, *i.e.*, end-to-end bandwidth and processing cost. Despite of the asymptotic trade-offs, we realized that, under various settings, certain construction might dominate the others. For instance, when the data corpus is very small, the security benefits of FULL would be more appealing than the marginal efficiency benefits from VHPART. On the other hand, when the data corpus is extremely large, the severity of the leakage of VHPART might diminishes because of the increased entropy of the corpus itself, while the efficiency benefits of VHPART would shine.

Hence, we envision an adaptive index management system that could switch between the different constructions under various conditions. The first key factor would be the scale of the ingested data: as the index size increases, the system would switch from FULL to VPART, then to VHPART, and increase the number of partitions of VHPART on-demand. The second key factor is the target performance requirement, and it determines when the system should switch to meet the

requirements. The third factor is the overheads of switching. Each switching would incurr certain amount of extra computations at the client, and it would increase as the data scale increases. The system should minimize the impact of the switching operation on the performance exposed to the end-user.

6.2.4 Multi-user End-to-end Encrypted Search

In Chapter 5, we focus on end-to-end encrypted search for one single user who uploads her documents to the cloud, and searches with keyword queries. This enables the fundamental use case for real-world cloud storage services, *e.g.*, Dropbox. For broader adoption, we might need to consider the more general setting where multiple users share a directory of documents in the cloud, and each of them can upload and search for documents in the shared directory. There are challenges in both security and efficiency under such setting. First, if some users, with access to the shared directory, collude with the service provider, what privacy guarantees can we provide to the query and documents of other users sharing the same directory? Second, when some users are removed from the sharing list, what kind of privacy can we achieve for queries and documents of other active users? In terms of efficiency, how to design the constructions so that the local updates by a user can be immediately searchable by other users sharing the same directory?

References

[1] Apache lucenetm 7.7.3 documentation. https://lucene.apache.org/core/7_7_3/index.html.

[2] Enron email dataset. https://www.cs.cmu.edu/~enron/.

[3] New york times news. https://archive.ics.uci.edu/ml/datasets/Bag+of+Words.

[4] Ubuntu dialogue corpus. https://www.kaggle.com/rtatman/ubuntu-dialogue-corpus.

[5] Learning with privacy at scale. *Apple Machine Learning J.*, 2017.

[6] J. Acharya, Z. Sun, and H. Zhang. Hadamard response: Estimating distributions privately, efficiently, and with little communication. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1120–1129, 2019.

[7] J. Allen, B. Ding, J. Kulkarni, H. Nori, O. Ohrimenko, and S. Yekhanin. An algorithmic framework for differentially private data analysis on trusted processors. *CoRR*, 2018.

[8] AMPlab, University of California, Berkeley. Big data benchmark. https://amplab.cs.berkeley.edu/benchmark/.

[9] Anati, Ittai and Gueron, Shay and Johnson, Simon P. and Scarlata, Vincent R. Innovative Technology for CPU Based Attestation and Sealing (March, 2017). https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing.

[10] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *Proc. IEEE Symp. on Security & Privacy*, 2015.

[11] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *Proc. CIDR*, 2013.

[12] R. K. Arvind Arasu. Oblivious query processing. In *Proc. ICDT*, 2014.

[13] S. Bajaj and R. Sion. Trusteddb: A trusted hardware based database with privacy and data confidentiality. In *Proc. SIGMOD*, 2011.

[14] B. Balle, J. Bell, A. Gascón, and K. Nissim. The privacy blanket of the shuffle model. In *CRYPTO*, 2019.

[15] B. Barak, K. Chaudhuri, C. Dwork, S. Kale, F. McSherry, and K. Talwar. Privacy, accuracy, and consistency too: a holistic solution to contingency table release. In *PODS*, 2007.

[16] R. Bassily, K. Nissim, U. Stemmer, and A. G. Thakurta. Practical locally private heavy hitters. In *NIPS*, 2017.

[17] R. Bassily and A. D. Smith. Local, private, efficient protocols for succinct histograms. In *STOC*, 2015.

[18] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS '68 (Spring)*, 1968.

[19] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. Smcql: Secure querying for federated databases. *Proc. VLDB Endowment*, 10(6):673–684, Feb. 2017.

[20] J. Beekman. Improving cloud security using secure enclaves. Technical Report No. UCB/EECS-2016-219, University of California, Berkeley, 2016.

[21] A. Bittau, U. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnes, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proc. SOSP*, 2017.

[22] L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. Cryptology ePrint Archive, Report 2019/1175, 2019. https://eprint.iacr.org/2019/1175.

[23] J. Blömer and N. Löken. Dynamic searchable encryption with access control. Cryptology ePrint Archive, Report 2019/1038, 2019. https://eprint.iacr.org/2019/1038.

[24] R. Bost. $\Sigma o\phi o\varsigma$: Forward secure searchable encryption. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 1143–1154. ACM Press, Oct. 2016.

[25] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 1465–1482. ACM Press, Oct. / Nov. 2017.

[26] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. Technical Report arXiv:1701.00981, arXiv, 2017.

[27] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: Sgx cache attacks are practical. Technical Report arXiv:1702.07521, arXiv, 2017.

[28] M. Bun, J. Nelson, and U. Stemmer. Heavy hitters and the structure of local privacy. In *PODS*, 2018.

[29] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 668–679. ACM Press, Oct. 2015.

[30] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*. The Internet Society, Feb. 2014.

[31] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for Boolean queries. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373. Springer, Heidelberg, Aug. 2013.

[32] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili. New constructions for forward and backward private symmetric searchable encryption. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 1038–1055. ACM Press, Oct. 2018.

[33] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou. Detecting covert timing channels with time-deterministic replay. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Oct. 2014.

[34] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with Déjà Vu. In *Proc. ASIACCS*, 2017.

[35] A. Cheu, A. D. Smith, J. Ullman, D. Zeber, and M. Zhilyaev. Distributed differential privacy via shuffling. In *EUROCRYPT*, 2019.

[36] G. Cormode, T. Kulkarni, and D. Srivastava. Marginal release under local differential privacy. In *SIGMOD*, 2018.

[37] G. Cormode, T. Kulkarni, and D. Srivastava. Answering range queries under local differential privacy. *PVLDB*, 2019.

[38] V. Costan and S. Devadas. Intel sgx explained. Technical Report Report 2016/086, Cryptology ePrint Archive, 2016.

[39] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proc. 25th USENIX Security*, 2016.

[40] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In A. Juels, R. N. Wright, and S. De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 79–88. ACM Press, Oct. / Nov. 2006.

[41] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou. Dynamic searchable encryption with small client storage. Cryptology ePrint Archive, Report 2019/1227, 2019. `https://eprint.iacr.org/2019/1227`.

[42] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, Aug. 2020. USENIX Association.

[43] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In *NIPS*, 2017.

[44] B. Ding, M. Winslett, J. Han, and Z. Li. Differentially private data cubes: optimizing noise sources and consistency. In *SIGMOD*, 2011.

[45] Z. Ding, Y. Wang, G. Wang, D. Zhang, and D. Kifer. Detecting violations of differential privacy. In *CCS*, 2018.

[46] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *Proc. USENIX Security*, 2015.

[47] J. C. Duchi, M. I. Jordan, and M. J. Wainwright. Local privacy and statistical minimax rates. In *FOCS*, 2013.

[48] F. B. Durak, T. M. DuBuisson, and D. Cash. What else is revealed by order-revealing encryption? In *Proc. CCS*, 2016.

[49] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. *Proc. EUROCRYPT*, 2006.

[50] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. TCC*, 2006.

[51] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[52] C. Dwork, G. N. Rothblum, and S. Vadhan. Boosting and differential privacy. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 51–60. IEEE, 2010.

[53] M. J. Dworkin. Sp 800-38c. recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality. Technical report, Gaithersburg, MD, USA, 2004.

[54] Ú. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *CCS*, 2014.

[55] S. Eskandarian and M. Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, 2017.

[56] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans. Efficient dynamic searchable encryption with forward privacy. *PoPETs*, 2018(1):5–20, Jan. 2018.

[57] G. Fanti, V. Pihur, and Úlfar Erlingsson. Building a rappor with the unknown: Privacy-preserving learning of associations and data dictionaries. *Proceedings on Privacy Enhancing Technologies*, 2016.

[58] V. Feldman, I. Mironov, K. Talwar, and A. Thakurta. Privacy amplification by iteration. In *FOCS*, 2018.

[59] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. STOC*, 2009.

[60] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. Technical Report Report 2012/099, Cryptology ePrint Archive, 2012.

[61] M. Giraud, A. Anzala-Yamajako, O. Bernard, and P. Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. Cryptology ePrint Archive, Report 2017/046, 2017. `http://eprint.iacr.org/2017/046`.

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

[63] Google. Encrypted bigquery client, 2016.

[64] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov. Breaking web applications built on top of encrypted data. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 1353–1364. ACM Press, Oct. 2016.

[65] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *Proc. CCS*, 2017. To appear.

[66] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proc. CCS*, 2016.

[67] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+flush: A fast and stealthy cache attack. In *Proc. DIMVA*, 2016.

[68] Gueron, Shay. Intel Advanced Encryption Standard (Intel® AES) Instructions Set (March, 2017). `https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set`.

[69] J. Hansen, S. Hissam, and G. A. Moreno. Statistical-Based WCET Estimation and Validation. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, 2009.

[70] B. Harangsri. *Query result size estimation techniques in database systems*. PhD thesis, The University of New South Wales, 1998.

[71] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the accuracy of differentially private histograms through consistency. *PVLDB*, 3(1), 2010.

[72] J. Hsu, M. Gaboardi, A. Haeberlen, S. Khanna, A. Narayan, B. C. Pierce, and A. Roth. Differential privacy: An economic method for choosing epsilon. In *Proc. CSF)*, July 2014.

[73] Intel. Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developers Manuals (May, 2018). `https://software.intel.com/en-us/articles/intel-sdm`.

[74] Intel Corporation. Intel Software Guard Extensions SDK, Developer Reference (April, 2017). https://software.intel.com/en-us/sgx-sdk/documentation.

[75] Intel Corporation. Intel Trusted Execution Technology: White Paper (April, 2017). http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html.

[76] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[77] Intel Corporation. Speculative execution side channel mitigations, revision 2.0, May 2018. Available at https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[78] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *Proc. VLDB*, 1992.

[79] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*. The Internet Society, Feb. 2012.

[80] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, Oct. 2002.

[81] N. M. Johnson, J. P. Near, and D. Song. Towards practical differential privacy for SQL queries. *PVLDB*, 11(5), 2018.

[82] S. Johnson. Intel SGX and side channels. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels, Mar. 2017.

[83] M. Joseph, A. Roth, J. Ullman, and B. Waggoner. Local differential privacy for evolving data. In *NeurIPS*, 2018.

[84] P. Kairouz, S. Oh, and P. Viswanath. The composition theorem for differential privacy. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 1376–1385, 2015.

[85] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sublinear complexity. In J. Coron and J. B. Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 94–124. Springer, Heidelberg, Apr. / May 2017.

[86] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 2012*, pages 965–976. ACM Press, Oct. 2012.

[87] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Accessing data while preserving privacy. *CoRR*, 2017.

[88] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 1449–1463. ACM Press, Oct. / Nov. 2017.

[89] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[90] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proc. CRYPTO*, 1999.

[91] I. Kotsogiannis, Y. Tao, X. He, M. Fanaeepour, A. Machanavajjhala, M. Hay, and G. Miklau. Privatesql: A differentially private sql query engine. *PVLDB*, 2019.

[92] I. Kotsogiannis, Y. Tao, A. Machanavajjhala, G. Miklau, and M. Hay. Architecting a differentially private SQL engine. In *CIDR*, 2019.

[93] M. G. Kuhn. *Compromising emanations: eavesdropping risks of computer displays*. PhD thesis, University of Cambridge, 2002.

[94] K. Kurosawa, K. Sasaki, K. Ohta, and K. Yoneyama. UC-secure dynamic searchable symmetric encryption scheme. In K. Ogawa and K. Yoshioka, editors, *IWSEC 16*, volume 9836 of *LNCS*, pages 73–90. Springer, Heidelberg, Sept. 2016.

[95] B. W. Lampson. A note on the confinement problem. *CACM*, 16:613–615, Oct. 1973.

[96] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proc. Usenix Security*, Aug. 2017.

[97] Z. Li, T. Wang, M. Lopuhaä-Zwakenberg, B. Skoric, and N. Li. Estimating numerical distributions under local differential privacy. In *SIGMOD*, 2020.

[98] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[99] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proc. HPCA*, 2016.

[100] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proc. CCS*, 2015.

[101] R. Lowe, N. Pow, I. Serban, and J. Pineau. The Ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. In *Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 285–294, Prague, Czech Republic, Sept. 2015. Association for Computational Linguistics.

[102] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, 2005.

[103] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

[104] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. Rote: Rollback protection for trusted execution. Technical Report Report 2017/048, Cryptology ePrint Archive, 2017.

[105] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proc. IEEE Symp. on Security & Privacy*, 2010.

[106] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. EuroSys*, 2008.

[107] R. McKenna, G. Miklau, M. Hay, and A. Machanavajjhala. Optimizing error of high-dimensional statistical queries under differential privacy. *PVLDB*, 11(10), 2018.

[108] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD*, 2009.

[109] Microsoft. Sql server 2016 always encrypted (database engine), 2016.

[110] I. Miers and P. Mohassel. IO-DSSE: Scaling dynamic searchable encryption to millions of indexes by improving locality. In *NDSS 2017*. The Internet Society, Feb. / Mar. 2017.

[111] R. Misener and C. A. Floudas. Piecewise-linear approximations of multidimensional functions. *J. Optim. Theory Appl.*, 145(1):120–147, 2010.

[112] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *Proc. S&P*, 2018.

[113] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'06, page 348–355, New York, NY, USA, 2006. Association for Computing Machinery.

[114] T. Müller, F. C. Freiling, and A. Dewald. Tresor runs encryption securely outside ram. In *Proc. USENIX Security*, 2011.

[115] J. Murtagh and S. Vadhan. The complexity of computing the optimal composition of differential privacy. In *Theory of Cryptography*, pages 157–175. Springer, 2016.

[116] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 644–655. ACM Press, Oct. 2015.

[117] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proc. CCS*, 2015.

[118] Nguyen, Khang T. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[119] Nguyen, Khang T. Usage Models for Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/cache-allocation-technology-usage-models.

[120] NYC Taxi & Limousine Commission. TLC Trip Record Data (April, 2017). http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[121] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in mapreduce. In *Proc. CCS*, 2015.

[122] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *Proc. USENIX Security*, 2016.

[123] G. Paoloni. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[124] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *Proc. OSDI*, 2016.

[125] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Proc. IEEE Symp. on Security & Privacy*, 2014.

[126] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *Proc. USENIX Security*, 2016.

[127] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Record*, 14(2):256–276, 1984.

[128] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. SOSP*, 2011.

[129] M. F. Porter. An algorithm for suffix stripping. In *Readings in Information Retrieval*. 1997.

[130] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 1341–1352. ACM Press, Oct. 2016.

[131] D. Proserpio, S. Goldberg, and F. McSherry. Calibrating data to sensitivity in private data analysis. *PVLDB*, 2014.

[132] W. H. Qardaji, W. Yang, and N. Li. Understanding hierarchical methods for differentially private histograms. *PVLDB*, 6(14), 2013.

[133] Z. Qin, Y. Yang, T. Yu, I. Khalil, X. Xiao, and K. Ren. Heavy hitter estimation over set-valued data with local differential privacy. In *CCS*, 2016.

[134] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Proc. USENIX Security*, 2015.

[135] X. Ren, C. Yu, W. Yu, S. Yang, X. Yang, J. A. McCann, and P. S. Yu. Lopub: High-dimensional crowdsourced data publication with local differential privacy. *IEEE Trans. Information Forensics and Security*, 13(9), 2018.

[136] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *In Proceedings of SIGIR'94*, pages 232–241. Springer-Verlag, 1994.

[137] R. M. Rogers, A. Roth, J. Ullman, and S. P. Vadhan. Privacy odometers and filters: Pay-as-you-go composition. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 1921–1929. Curran Associates, Inc., 2016.

[138] S. Ruggles, J. T. Alexander, K. Genadek, R. Goeken, M. B. Schroeder, and M. Sobek. Integrated public use microdata series: Version 5.0 [machine-readable database], 2010.

[139] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. IEEE Symp. on Security & Privacy*, 2015.

[140] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proc. NDSS*, 2017.

[141] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. In *Proc. CHES*, 2002.

[142] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55. IEEE Computer Society Press, May 2000.

[143] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *NDSS 2014*. The Internet Society, Feb. 2014.

[144] E. Stefanov and E. Shi. Multi-cloud oblivious storage. In *Proc. CCS*, 2013.

[145] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proc. CCS*, 2013.

[146] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, Nov. 2013.

[147] R. Strackx and F. Piessens. Ariadne: A minimal approach to state continuity. In *Proc. USENIX Security*, Aug. 2016.

[148] S. Tople and P. Saxena. On the trade-offs in oblivious execution techniques. In *Proc. DIMVA*, 2017.

[149] I. Trummer and C. Koch. Approximation schemes for many-objective query optimization. In *Proc. SIGMOD*, June 2014.

[150] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Varom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proc. USENIX Security*, 2018.

[151] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic attacks. In *Proc. SOSP*, 2015.

[152] C. Van Rompay, R. Molva, and M. Önen. A leakage-abuse attack against multi-user searchable encryption. *PoPETs*, 2017(3):168, July 2017.

[153] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proc. IEEE Symp. on Security & Privacy*, 2013.

[154] V. Viswanathan. Disclosure of H/W prefetcher control on some Intel processors. `https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors`.

[155] F. Wang, C. Tan, A. C. KÃ¶nig, and P. Li. Efficient document clustering via online nonnegative matrix factorizations. In *Eleventh SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, April 2011.

[156] N. Wang, X. Xiao, Y. Yang, T. D. Hoang, H. Shin, J. Shin, and G. Yu. Privtrie: Effective frequent term discovery under local differential privacy. In *ICDE*, 2018.

[157] T. Wang, J. Blocki, N. Li, and S. Jha. Locally differentially private protocols for frequency estimation. In *USENIX Security*, 2017.

[158] T. Wang, B. Ding, J. Zhou, C. Hong, Z. Huang, N. Li, and S. Jha. Answering multidimensional analytical queries under local differential privacy. In *SIGMOD*, 2019.

[159] T. Wang, N. Li, and S. Jha. Locally differentially private frequent itemset mining. In *SP*, 2018.

[160] T. Wang, N. Li, and S. Jha. Locally differentially private heavy hitter identification. *IEEE Trans. Dependable Sec. Comput.*, 2019.

[161] N. Weichbrodt, A. Kurmus, P. R. Pietzuch, and R. Kapitza. Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In *Proc. ESORICS*, 2016.

[162] C. V. Wright and D. Pouliot. Early detection and analysis of leakage abuse vulnerabilities. Cryptology ePrint Archive, Report 2017/1052, 2017. `http://eprint.iacr.org/2017/1052`.

[163] Q. Xiao, M. K. Reiter, and Y. Zhang. Mitigating storage side channels using statistical privacy mechanisms. In *Proc. CCS*, 2015.

[164] X. Xiao, G. Wang, and J. Gehrke. Differential privacy via wavelet transforms. In *ICDE*, 2010.

[165] M. Xu, B. Ding, T. Wang, and J. Zhou. Collecting and analyzing data jointly from multiple services under local differential privacy. *PVLDB*, 2020.

[166] M. Xu, A. Papadimitriou, A. Feldman, and A. Haeberlen. Using differential privacy to efficiently mitigate side channels in distributed analytics. In *Proceedings of the 11th European Workshop on Systems Security*, EuroSec'18, New York, NY, USA, 2018. Association for Computing Machinery.

[167] M. Xu, A. Papadimitriou, A. Haeberlen, and A. Feldman. Hermetic: Privacy-preserving distributed analytics without (most) side channels.

[168] M. Xu, T. Wang, B. Ding, J. Zhou, C. Hong, and Z. Huang. Dpsaas: Multi-dimensional data sharing and analytics as services under local differential privacy. *PVLDB*, 2019.

[169] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proc. IEEE Symp. on Security & Privacy*, 2015.

[170] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proc. USENIX Security*, 2014.

[171] A. A. Yavuz and J. Guajardo. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In O. Dunkelman and L. Keliher, editors, *SAC 2015*, volume 9566 of *LNCS*, pages 241–259. Springer, Heidelberg, Aug. 2016.

[172] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI*, 2012.

[173] J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

[174] N. Zhang, K. Sun, W. Lou, and Y. T. Hou. CaSE: Cache-assisted secure execution on ARM processors. In *Proc. Oakland*, 2016.

[175] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proc. CCS*, 2012.

[176] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In T. Holz and S. Savage, editors, *USENIX Security 2016*, pages 707–720. USENIX Association, Aug. 2016.

[177] Z. Zhang, T. Wang, N. Li, S. He, and J. Chen. Calm: Consistent adaptive local marginal for marginal release under local differential privacy. In *CCS*, 2018.

[178] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proc. NSDI*, 2017.

[179] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6–es, July 2006.

APPENDIX A
HERMETIC IMPLEMENTATION DETAILS

A.1 Query operators

A.1.1 List of query operators

Query processing in Hermetic is built upon a set of oblivious operators. From the bottom up, we have *OEE operators* that support simple sort and merge, *auxiliary operators* that support relation transformations, including re-ordering, grouping, expansion, etc. and statistics, *relational operators* that support well-known SQL-style query processing. We describe all the operators below, together with their definitions and oblivious constructions. We also mark the one unique in Hermetic as `HMT`, and other from previous work with citations:

OEE Operators

- (`HMT`) `merge-sort` Given an array of tuples that fits inside OEE and the set of order-by attributes, sort them in the order of the given attributes.
- (`HMT`) `linear-merge` Given two arrays of sorted tuples, that together fit inside OEE and the set of order-by attributes, merge them into one sorted array of tuples with one linear scan.

Auxiliary Operators

- (`HMT`) `hybrid-sort` Given an array of tuples, beyond the capacity of OEE, and a set of order-by attributes, sort them in the order of the given attributes. Algorithm A.1 shows the pseudo-code of `hybrid-sort` (Lines 10-17).
- (`HMT`) `hybrid-merge` Given two arrays of sorted tuples, that together are beyond the capacity of OEE and the set of order-by attributes, merge them into one sorted array of tuples. Algorithm A.1 shows the pseudo-code of `hybrid-merge` (Lines 18-31).
- ([12]) `augment` Given a relation, a function and an attribute name, returns a relation whose attributes consist of the given relation's attributes and the given attribute, and each row of which

is the corresponding row in the given relation extended with the value of the function applied on the that row.

- ([12]) `filter` Given a relation and a predicate on a set of attributes, return a relation that consists of all the rows from the given relation that satisfy the predicate. Algorithm A.1 shows how to construct `filter` using `hybrid-sort` (Lines 1-9): first, augment the relation with "mark" equal to 1 if the row satisfy the predicate or 0 otherwise. Second, `hybrid-sort` the relation on "mark" to in descending order so that all the rows with "mark" equal to 1 are at the front. Finally, return all the rows that satisfy the predicate at the front.
- ([12]) `groupId` Given a relation, a set of aggregate-on attributes, group the rows based on the aggregate-on attributes, and `augment` the relation with the group internal ID, starting from `1`. The `groupId` operator could be constructed by first sorting the relation on the aggregate-on attributes using `hybrid-sort`, and then keeping a running counter as the group internal ID while scanning over the sorted relation and extending each row with the group internal ID.
- ([12]) `group-running-sum` Given a relation, a set of aggregate-on attributes, a summation attribute and a new attribute, `augment` the relation with the new attribute whose value is the running sum over the summation attribute, in the reverse order of the group internal ID, within each group of the aggregate-on attributes. The `group-running-sum` operator could be constructed by first applying `groupId`, then sorting the relation on the aggregate-on attributes, plus the group ID attribute in descending order, using `hybrid-sort`, and finally keeping a running sum over the summation attribute while scanning over the sorted relation and extending each row with the running sum.
- ([12]) `semijoin-aggregation` Given two relations and a set of join attributes, `augment` each of the two relations with the number of matches, i.e. join degree, on the given set of join attributes from the other relation. To `augment` the first relation with the join degree, first augment each relation with attribute "src", e.g. 1(0) for the first(second) relation, `union` the two relations, and then `hybrid-sort` it on the join attributes plus the "src" attribute in ascending

order. Then `augment` the relation with running counter of the rows in each group of the join attributes, but only increment the counter if the ”src” is equal to 0. Finally, apply `filter` to keep rows with ”src” equal to 1. Then, repeat on the second relation by switching the order.

- ([12,122]) `expand` Given one relation from the `semijoin-aggregation` output, duplicate each row by the times, equal to the number of matches, in the order of the join attributes. In Hermetic, this operation will leak the differentially private total `join` size.
- ([12]) `stitch` Given two relations from `expand`, return a relation whose i-th row is the concatenation of the i-th rows from the two given relations.
- (`HMT`) `histogram` Given a relation and a target attribute, returns a histogram over the given attribute of the relation. In Hermetic, we first `hybrid-sort` the relation on the target attribute, and then keep a running counter, refreshed to 0 at the beginning of each histogram bucket, and `augment` the relation with the running counter if the row is the last of the bucket, or -1 otherwise. Finally, `hybrid-sort` the relation on the augmented attribute in descending order, and return the rows with non-negative values.
- (`HMT`) `multiplicity` Given a relation and a set of target attributes, calculates the number of times that the most common values of the target attributes appear in the relation. In Hermetic, we first `hybrid-sort` the relation on the set of target attributes, and then keep a counter on the most common values while scanning over the relation.

Relational operators

- `project` Given a relation and a set of attributes, returns a relation, each row of which is a projection, onto the given attributes, of the corresponding row in the input relation.
- `rename` Given a relation and a set of old and new names, returns a relation, whose specified attributes are renamed from old to new given names.
- `union` Given two relations, returns a relation whose attribute set is the union of the two relations and that contains all the rows from the two relations, with new attributes filled by `null`.
- ([12]) `select` Given a relation and a predicate on a set of attributes, return a relation that

```
filter(R = {t_0, t_1, ..., t_n}, p):

1: osize ← 0
2: for t ∈ R do
3:     cwrite(p(t), match, 1, 0)
4:     t ← t ∪ {('mark', match)}
5:     osize ← osize + match
6: hybrid − sort(R, 'mark', desc = 0)
7: return R[0 : osize]
```

Figure A.1: The oblivious `filter` operator

consists of all the rows from the given relation that satisfy the predicate. The `select` operator could be constructed with one `filter` directly.

- ([12]) `groupby` Given a relation, a set of aggregate-on attributes, an accumulate function and a new accumulate attribute name, group the rows in the given relation by the aggregate-on attributes, apply the accumulate function on each group and returns the aggregate-on attributes and the accumulation of each group. The `groupby` operator could be constructed with one `groupId`, followed by a `group-running-sum`, and, finally, a `filter` to keep the aggregate results only.
- `orderby` Given a relation and a set of order-by attributes, order the rows in the input relation by the order-by attributes. The `orderby` operator could be constructed with one `hybrid-sort` directly.
- `cartesian-product` Given two relations, returns a relation that consists of the concatenations of every pair of rows, one from the first relation and the other from the second relation.
- ([12]) `join` Given two relations and a set of join attributes, returns a relation with every pair of matches from the two relations on the join attributes. The `join` operator could be constructed by first deriving the join degree of each row using `semijoin-aggregation`, then expanding each row by its join degree using `expand` and finally joining the two expanded relations using `sticth` to get the join result.

Figure A.2: Every x86 instruction used in Hermetic OEE

add	addl	cmovbe	cmove
cmovg	cmovle	cmovne	cmp
cmpl	imul	ja	jae
jb	jbe	je	jmp
jne	lea	mov	movl
movzbl	pop	push	ret
setae	setb	setbe	sete
setg	setle	setne	shl
shll	sub	test	

A.2 Predictable timing for OEE operators

Hermetic currently performs two operations within OEEs, MergeSort and LinearMerge. As discussed in Section 3.6.2, although OEEs allow these operators to perform data-dependent memory accesses (which is why they are faster than purely data-oblivious operators), they are carefully structured to avoid data-dependent control flow and instructions and to constrain the set of possible memory accesses. The latter makes the number of accesses that miss the L1 cache, and thus must be served by slower caches, predictable. Moreover, their execution time is padded to a conservative upper bound calibrated to a specific model of CPU. This section describes these measures in more detail.

A.2.1 Avoiding data-dependent control flow and instructions

We ensure that OEE operators' code is free from data-dependent branches using techniques similar to [134]. Furthermore, we limit the set of instructions that they use to avoid those with data-dependent timing. Table A.2 lists all of the x86 instructions that are used by the operators that run in an OEE. The instructions marked in dark green have constant execution time, as verified by Andrysco et al. [10]. The instructions in light green were not among the instructions verified by Andrysco et al., but are either variants of them or, as is the case with `cmov*`, are known to be constant time [134].

A.2.2 Making L1 cache misses predictable

By construction, all memory accesses performed by OEE operators are served from the cache. Moreover, they have deterministic control flow, and therefore perform a fixed number of memory accesses for a given input size. For example, Algorithm A.3 shows the pseudocode for OEE MergeSort. Note that the two running pointers that scan over the two sublists will keep accessing the data even if one of the sublists has been completely merged (Lines 9 and 10). This will not affect the correctness due to the modified merge condition (Line 11), but it will make the total number of memory accesses on each input deterministic. Nevertheless, the operators' timing could vary depending on whether accesses hit the L1 cache or whether they have to be served by slower caches.

To address this problem, we could determine an upper bound on an operator's execution time by assuming that all of its memory accesses are served from the L3 cache, but this would be wildly conservative. In particular, it would result in a 43x slowdown for OEEMergeSort and a 33x slowdown for OEELinearMerge (see Figure A.4). Instead, we carefully structure each operator's code to make the number of L1 cache misses predictable. For example, if we examine the code of MergeSort, we can see that its memory accesses adhere to three invariants:

1. *Each merge iteration accesses one of the same tuples as the previous iteration.* Figure A.5 shows the memory traces of MergeSort on 32,768 input tuples that have been sorted, reverse sorted, and permuted randomly. With sorted input, the two running pointers (Lines 9 and 10 in Algorithm A.3) follow the invariant: the second pointer keeps accessing the first tuple of the second sub-list until the first pointer finishes scanning through the first sub-list. Then, the first pointer keeps accessing the last tuple of the first sub-list until the second pointer reaches the end of the second sub-list. The same memory access pattern holds even with reverse sorted and randomly permuted inputs because only one of the two running pointers would advance after each merge iteration. If we assume that a tuple can fit in a single L1 cache line, as is the case in our examples, then one of the two memory accesses in each iteration will be an L1 hit.

2. *Merge iterations access the input tuples sequentially.* The merge loop (Lines 6–15) accesses the tuples in each of the input sub-lists sequentially. Consequently, if each tuple is smaller than an L1 cache line, then accessing an initial tuple will cause subsequent tuples to be loaded into the same cache line. Assuming that the L1 cache on the OEE's CPU is large enough – as is the case in our experiments – these subsequent tuples will not be evicted from the cache between merge loop iterations, and future accesses to them will be L1 hits. If a cache line is of CL bytes and each tuple is of TP bytes, then at least $1 - \frac{TP}{CL}$ of the tuple accesses will be L1 hits.
3. *Local variables are accessed on every iteration.* Since MergeSort is not recursive, the variables local to the merge loop (Lines 7–14) are accessed on every iteration. Furthermore, because more cache lines are allocated to the OEE than local variables and tuples, these accesses should be L1 hits as well.

Given these invariants, it is possible to express the lower bound on L1 hits as a formula. Let N be the number of tuples per OEE input block, FPR as the number of fields in each tuple, and FNO as the number of fields used as keys for sorting. Then, the lower bound is given by:

$$\begin{aligned} L_{ms} &= (77 + 11 * FNO + 12 * FPR) * N * log(N) \\ &+ (1 - \frac{FPR}{16})(12 + FPR) * N * log(N) \\ &+ 5N * log(N) + \frac{15}{4} N * log(N) \end{aligned} \tag{A.1}$$

A similar analysis can be done for LinearMerge, resulting in the following lower bound formula:

$$\begin{aligned} L_{lm} &= (69 + 24 * FNO + 11 * FPR) * N + 14 \\ &+ (1 - \frac{FPR}{16})(12 + FNO + FPR) * N \\ &+ \frac{15}{4} N \end{aligned} \tag{A.2}$$

Plugging in the values for N, FPR, FNO from our experimental data, we can see that the majority of accesses are served by the L1 — approximately 89.06% and 79.73% for MergeSort and LinearMerge, respectively.

A.2.3 Determining a conservative upper bound on execution time

Even though the number of L1 cache misses is predictable regardless of the input, as discussed in Section 3.6.2, we still pad OEE operators' execution time to a conservative upper bound to account for timing variation that might occur in modern CPUs (e.g., due to pipeline bubbles).[1] To determine this upper bound, we could take the lower bound on L1 hits determined above and assume that all other memory accesses were served from the L3 cache (LLC). We could then compute the bound by plugging in the L1 and L3 access latencies from processor manual [76]. As Figure A.4 shows, however, due to the superscalar execution in modern CPUs, the resulting bound is still 10x larger than the actual execution time.

Instead, we achieve a tighter but still conservative bound using worst-case execution time (WCET) estimation techniques [69]. We performed 32 experiments each on random, sorted, and reverse sorted inputs in which we measured the L1 hit and miss rates using the CPU's performance counters. We then used linear regression to learn *effective* L1 and L3 hit latencies l^*_{L1} and l^*_{L3} for the specific CPU model. Since we could not be sure that we have observed the worst case in our experiments, we increased the L1 and L3 latency estimates by 10% and 20%, respectively to obtain bounds $\hat{l}_{L1}$ and $\hat{l}_{L3}$. Table A.6 shows l^*_{L1}, l^*_{L3}, $\hat{l}_{L1}$, and $\hat{l}_{L3}$ estimated for MergeSort, as well as the latencies from the specification. The computed upper bounds were 1.6x the actual execution time for MergeSort and 1.96x for LinearMerge and were never exceeded in our experiments.

The effective L1 and L3 hit latencies would have to be derived on each distinct CPU model. To do so, we envision a profiling stage that would replicate the procedure above and would be performed before Hermetic is deployed to a new processor.

1. We determine execution time using the `rdtsc` instruction. `rdtsc` is available to enclaves in SGX version 2 [73]. Moreover, a malicious platform cannot tamper with the timestamp register because the core is "locked down."

A.2.4 Overhead of time padding

We examine the overheads of padding time for `mergesort` and LinearMerge in the OEE, and how they depend on the size of the un-observable memory.

Analogous to Section 3.7.2, we generated random data and created relations with enough rows to fill up a cache of 1MB to 27MB. On this data, we measured the time required to perform the actual computation of the two primitives, and the time spent busy-waiting to pad the execution time. We collected results across 10 runs and report the average in Figure A.7. The overhead of time padding ranges between 34.2% and 61.3% for MergeSort, and between 95.0% and 97.9% for LinearMerge. Even though the padding overhead of MergeSort is moderate, it is still about an order of magnitude faster than BatcherSort. This performance improvement over BatcherSort is enabled by having an OEE, and it is the main reason why Hermetic is more efficient than DOA-NoOEE , even though Hermetic provides stronger guarantees.

A.3 Oblivious primitives which use dummy tuples

Section 3.4 mentions that we modified the oblivious primitives from prior work [12] to accept dummy tuples. These modifications have two goals: (1) allowing the primitives to compute the correct result on relations that have dummy tuples, and (2) providing an oblivious method of adding a controlled number of dummy tuples to the output of certain primitives.

A.3.1 Supporting dummy tuples

Dummy tuples in Hermetic are denoted by their value in the `isDummy` field. Below we list all the primitives we had to modify to account for this extra field.

`groupid:` This primitive groups the rows of a relation based on a set of attributes, and adds an incremental id column, whose ids get restarted for each new group. In order for this to work correctly in the face of dummy tuples, we need to make sure that dummy records do not get grouped

with real tuples. To avoid this, we expand the set of grouping attributes by adding the `isDummy` attribute. The result is that real tuples get correct incremental and consecutive ids.

`grsum`: Grouping running sum is a generalization of `groupid`, and as such, we were able to make it work with dummy tuples by applying the same technique as above.

`union`: Union expands the attributes of each relation with the attributes of the other relation, minus the common attributes, fills them up with `nil` values, and then appends the rows of the second relation to the first. To make `union` work with dummy tuples, we make sure the `isDummy` attribute is considered common across all relations. This means that the output of unions has a single `isDummy` attribute, and its semantics are preserved.

`filter`: To make filter work with dummy tuples, we need to make sure that user predicates select only real rows. To achieve this, we rewrite a user-supplied predicate p as "(`isDummy` $= 0$) AND p". This is enough to guarantee that no dummy tuples are selected.

`join`: What we want for join is that real tuples from the one relation are joined only with real tuples from the other relation. To achieve this, we include the `isDummy` attribute to the set of join attributes of the join operation.

`groupby`: For the `groupby` primitive, we apply the same technique as for the `groupid` and `grsum` – we expand the grouping attributes with `isDummy`.

`cartesian-product`: Cartesian product pairs every tuple of one relation with every tuple of the other, and this happens even for dummy tuples. However, we need to make sure that only one instance of `isDummy` will be in the output relation, and that it will retain its semantics. To do this, we keep the `isDummy` attribute of only one of the relations, and we update its value to be 1 if both paired tuples are real and 0 otherwise.

`multiplicity` and `histogram`: These two primitives need to return the corresponding statistics of the real tuples. Therefore, we make sure to (obliviously) exclude dummy tuples for the computation of multiplicities and histograms.

A.3.2 Adding dummy tuples to the primitive outputs

To enable the introduction of dummy tuples, we alter the primitives `filter`, `groupby`, and `join`. The oblivious `filter` primitive from previous work involves extending the relation with a column holding the outcome of the selection predicate, obliviously sorting the relation based on that column, and finally discarding any tuples which do not satisfy the predicate. To obliviously add N tuples, we keep N of the previously discarded tuples, making sure to mark them as dummy.

`groupby` queries involve several stages, but their last step is selection. Therefore, dummy tuples can be added in the same way.

`join` queries involve computing the join-degree of each tuples in the two relations.[2] To add noise, we modify the value of join-degree: instead of the correct value, we set the join-degree of all dummy tuples to zero, except one, whose degree is set to N. As a result, all previous dummy tuples are eliminated and N new ones are created.

A.4 Hermetic multi-objective query optimization

Hermetic's query planner uses multi-objective optimization [149] to find the optimal query plan that matches the user's priorities. A query plan is associated with multiple costs, including the overall performance cost and a vector of privacy costs across the involved relations. The user's specification includes a vector of bounds, $\mathbf{B}$, and a vector of weights, $\mathbf{W}$, for the privacy costs on all the input relations. The planner's output is the plan where the weighted sum of all the costs is as close to optimal as possible and where all of the privacy costs are within the bounds. Each plan that the planner considers could be represented as a join tree covering all the input relations, with each noised operator assigned a privacy parameter, ε_i. (The current query planner only considers different join orders and privacy parameters. We leave more advanced query optimization to future work.)

2. In a join between relations R and S, the join-degree of a tuple in R corresponds to the number of tuples in S whose join attribute value is the same with this row in R.

The planner first constructs the complete set of alternative plans joining the given set of relations. Then, for each of the candidate plans, the planner formalizes an optimization problem on the privacy parameters of all the noised operators, and solves it using linear programming. Finally, the plan that both meets the bounds and has the best weighted-sum of costs is selected.

Returning to the query example in Figure 3.4, let p be the plan under consideration, $\mathrm{BM25}\varepsilon[i]$ be the privacy parameter on the i-th operator of the plan, and $f_p(\mathrm{BM25}\varepsilon)$ be the plan's overall performance cost. Then, we could solve the following optimization problem for the privacy parameters:

$$\begin{aligned} \min\ & \mathbf{W} \cdot \mathbf{A} \cdot \mathrm{BM25}\varepsilon + f_p(\mathrm{BM25}\varepsilon) \\ s.t.\ & \mathbf{A} \cdot \mathrm{BM25}\varepsilon \leq \mathbf{B}, \mathrm{BM250} < \mathrm{BM25}\varepsilon \leq \mathbf{B}. \end{aligned} \tag{A.3}$$

Here, the matrix $\mathbf{A}$ is the linear mapping from privacy parameters to the privacy costs on the input relations. For instance, suppose the C, T and P relations are indexed as 0, 1 and 2, and the privacy parameters on the selection on C, the selection on P, the join of C and T and the join of $(C \bowtie T)$ and P are indexed as 0, 1, 2 and 3 respectively. Then, the corresponding $\mathbf{A}$ for the plan is:

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{A.4}$$

Finding an exact solution to this optimization problem is challenging because we cannot assume that the performance cost function $f_p(\mathrm{BM25}\varepsilon)$ is either linear or convex. As a result, Hermetic approximates the solution instead. The planner first partitions the entire parameter space into small polytopes, and then approximates the performance cost function on each polytope as the linear interpolation of the vertices of the polytope. Thus, it achieves a piecewise linear approximation of the performance cost function. Finally, the planner can solve the corresponding piecewise linear programming problem to obtain an approximately-optimal assignment for the privacy parameters. This approach is consistent with existing nonlinear multidimensional optimization techniques in

the optimization literature [111].

Let $f_p(\mathrm{BM25}\varepsilon)$ be the performance cost function, d be the number of operators in the query plan, and K be the number of partitions on each parameter dimension. Then, the entire parameter space could be partitioned into K^d polytopes, each of which has 2^d vertices. We pick one of the vertices, $\mathrm{BM25}\varepsilon_0$, and d other vertices, $\mathrm{BM25}\varepsilon_1, \ldots, \mathrm{BM25}\varepsilon_d$, each of which is different from $\mathrm{BM25}\varepsilon_0$ in exactly one parameter dimension. Then any point, $\mathrm{BM25}\varepsilon$, as well as its performance cost, $f_p(\mathrm{BM25}\varepsilon)$, in such polytope could be represented as:

$$
\begin{aligned}
(\mathrm{BM25}\varepsilon, f_p(\mathrm{BM25}\varepsilon)) = & \sum_{i=1}^{d} u_i * ((\mathrm{BM25}\varepsilon_i, f_p(\mathrm{BM25}\varepsilon_i)) - (\mathrm{BM25}\varepsilon_0, f_p(\mathrm{BM25}\varepsilon_0))) \\
& + (\mathrm{BM25}\varepsilon_0, f_p(\mathrm{BM25}\varepsilon_0)) \\
\text{, where } & 0 \leq u_i \leq 1, \\
& 0 \leq u_i + u_j \leq 2, \\
& 0 \leq u_i + u_j + u_k \leq 3, \\
& \ldots \\
& 0 \leq \sum_{i=1}^{d} u_i \leq d.
\end{aligned}
\tag{A.5}
$$

For each such polytope, we can plug Equations A.5 into Equation A.3 to obtain the following linear

programming problem:

$$
\begin{aligned}
\min\ & \mathbf{W} \cdot \mathbf{A} \cdot \mathrm{BM25}\varepsilon + \\
& \sum_{i=1}^{d} u_i * (f_p(\mathrm{BM25}\varepsilon_i) - f_p(\mathrm{BM25}\varepsilon_0)) + f_p(\mathrm{BM25}\varepsilon_0) \\
s.t.\ & \mathbf{A} \cdot \mathrm{BM25}\varepsilon \leq \mathbf{B}, \\
& \mathbf{0} < \mathrm{BM25}\varepsilon \leq \mathbf{B}, \\
& \mathrm{BM25}\varepsilon - \sum_{i=1}^{d} u_i * (\mathrm{BM25}\varepsilon_i - \mathrm{BM25}\varepsilon_0) = \mathrm{BM25}\varepsilon_0, \\
& 0 \leq u_i \leq 1, \\
& 0 \leq u_i + u_j \leq 2, \\
& 0 \leq u_i + u_j + u_k \leq 3, \\
& \ldots \\
& 0 \leq \sum_{i=1}^{d} u_i \leq d.
\end{aligned}
\tag{A.6}
$$

Solving this linear programming problem on all the polytopes enables the query planner to determine the optimal assignment of privacy parameters for the plan.

The number of partitions of the parameter space, K, affects the optimization latency and accuracy. Larger K leads to more fine-grained linear approximation of the non-linear objective, but requires more linear programmings to be solved. To amortize the optimization overheads for large K, the query planner could be extended with parametric optimization [78] to pre-compute the optimal plans for all possible W and B so that only one lookup overhead is necessary to derive the optimal plan at runtime.

$\texttt{merge} - \texttt{sort}(\{\mathcal{R} = \{t_0, t_1, \ldots, t_n\} : \mathcal{B}\}, \text{REAL}, '', \text{ascend})$:

1: **for** $len \in \{2^0, 2^1, \ldots, 2^{log_2(n)}\}$ **do**
2: **for** $offset \in \{0, 2 \cdot len, \ldots, n - 2 \cdot len\}$ **do**
3: $pos_0 \leftarrow offset$
4: $pos_1 \leftarrow offset + len$
5: `mov eax, len; add eax, eax`
6: mov ebx, pos_0; mov ecx, pos_1
7: lea edx, [ebx]; lea edi, [ecx]
8: LOOP: cmp edx, edi
9: cmovle esi, edx; cmovg esi, edi

// Merge [pos_0] if $pos_1 \geq 2 \cdot len$

```
10: mov esi, len; mov esi, esi
11: cmp esi, ecx; cmovle esi, edx
```

12: // Merge [pos_1] if $pos_0 \geq len$

```
13: mov esi,len; cmp esi, ebx
14: cmovle esi, edi
15: mov [-eax], esi
```

16: // update pos_0

```
17: mov esi, $0; cmp edx, edi
18: cmovle esi,$1; add ebx, esi
19: mov esi, len; cmp ebx, esi
20: cmovg ebx, esi
```

21: // update pos_1

```
22: mov esi, $0; cmp edx, edi
23: cmovg esi, $1; add ecx, esi
24: mov esi, len; add esi,esi
25: cmp ecx, esi; cmovg ecx, esi
```

26: // load the next item

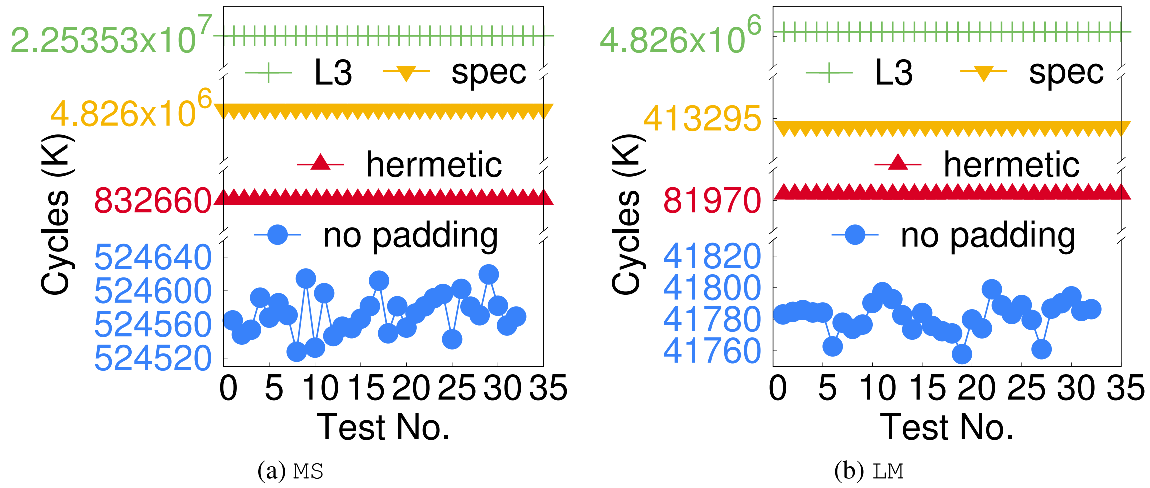
```
27: cmp edx, edi; cmovle esi, ebx
28: cmovg esi, ecx; lea esi, [esi]
29: cmovle edx, esi; cmovg edi,esi
```

30: // decrement the counter

```
31: sub eax,$1; cmp eax,$0;ja LOOP
```

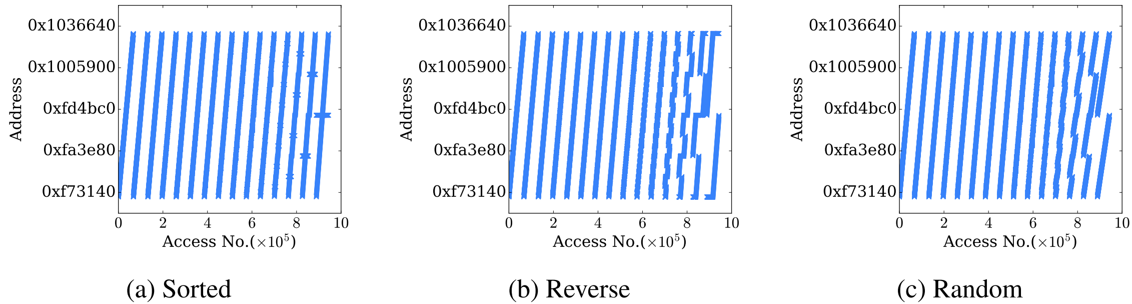
32: $\mathcal{R}[offset : offset + 2 \cdot len] \leftarrow \mathcal{B}[offset : offset + 2 \cdot len]$

Figure A.3: The `merge-sort` supported in OEE.



(a) MS (b) LM

Figure A.4: Cycle-resolution measurements of the actual timing of MergeSort (MS) and LinearMerge (LM) inside the OEE, and their padded timing, respectively.



(a) Sorted (b) Reverse (c) Random

Figure A.5: Memory access patterns of OEEMergeSort on sorted, reverse sorted and random input.

Figure A.6: L1 hit and miss latencies for MergeSort, as reported by Intel's specifications (l_{L1}, l_{L3}), and as measured on different datasets (l^*_{L1}, l^*_{L3}). The last columns show the values we used in our model. All values are in cycles.

Data	l_{L1}	l_{L3}	l^*_{L1}	l^*_{L3}	$\hat{l}_{L1}$	$\hat{l}_{L3}$
Random			0.6800	3.3400		
Ordered	4	34	0.6693	3.8032	0.74	5.0
Reverse			0.6664	4.2630		

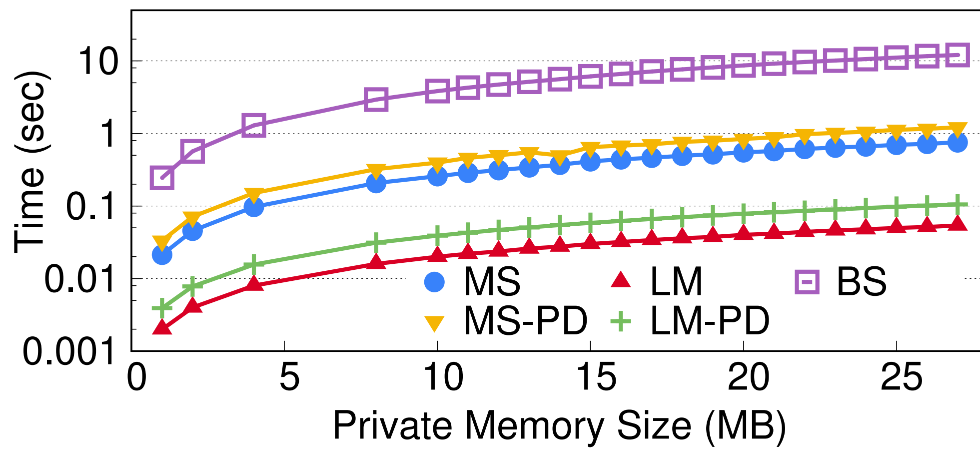


Figure A.7: Latency of MergeSort (MS), LinearMerge (LM) with increasing un-observable memory size, compared to the BatcherSort (BS). -PD indicates time padding.

APPENDIX B

SIZE-LOCKED INDEXING SECURITY AND CONSTRUCTION DETAILS

B.1 Formal Cryptographic Analysis

Security of the full size-locked index scheme. With a formal model in place, we start with arguing the security of the encrypted size-locked index scheme from Section 5.4. For this we are able to prove a very strong result, in the sense that the leakage profile $\mathcal{L}$ is very simple. It has $\mathcal{L}_{up}$ simply output the number of postings in the update, and $\mathcal{L}_{se}$ leak the total number n of documents currently in the index and the aggregate number N of postings that have been added via updates. Note that both numbers are monotonically increasing.

We have the following theorem.

Theorem 11. *Let* Π *be the encrypted size-locked index scheme from Section 5.4 using a symmetric encryption scheme* SE*. Let* $\mathcal{L}$ *be the leakage profile defined above. Then we give a simulator* $\mathcal{S}$ *such that for any adversary* $\mathcal{A}$ *we can build an adversary* $\mathcal{B}$ *such that* $\mathrm{Adv}^{\mathcal{L}}_{\Pi,\mathcal{S}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{cpa}}_{\mathsf{SE}}(\mathcal{B})$ *where* $\mathcal{B}$ *and each algorithm of* $\mathcal{S}$ *run in time that of* $\mathcal{A}$ *plus some small overhead.*

The simulator $\mathcal{S}$ works as follows. The first time either algorithm is executed, it sets $\mathsf{st} \leftarrow\!\!\$\ \mathsf{KeySp}$, i.e., it picks a random key for the underlying encryption scheme SE. On input m algorithm $\mathcal{S}_{up}$ simply outputs the encryption under K of an all zeros string of length $W + M + (W + 1) \cdot m$ bytes. On input n, N algorithm $\mathcal{S}_{se}$ chooses a random byte string of length exactly

$$(W + W/2 + M) \cdot n + (W + 1) \cdot N \ .$$

Then a simple reduction to the security of SE establishes the theorem, because no matter what inputs the adversary chooses, the resulting ciphertexts in $\textsc{Real}_{\Pi}$ will always be encryptions of encodings of a lengths exactly indicated by the formulas above. By the CPA security of SE these ciphertexts are indistinguishable from ones that encrypt zeros.

Security of vertically partitioned scheme. We next analyze the vertically partitioned scheme from Section 5.5.1. The update algorithm is exactly the same as the full scheme, so we only need to recall how the search algorithm works formally. The server state EDB always consists of ciphertexts $(B_1, \ldots, B_L)$ and $(\vec{C}_2, \ldots, \vec{C}_L, \vec{C}_{L+1})$, where $L = \lceil N/\mathsf{Cap}(N) \rceil$ (we identify the top-level update cache $\vec{C}_1$ with $\vec{C}_{up}$ in the experiment, so it is not represented in EDB explicitly to fit our formalism). The scheme maintains the invariant that $B_1, \ldots, B_{L-1}$ will always contain exactly $\mathsf{Cap}(N)$ postings, and B_L contains at most that many.

Search takes input $K, q, i, \mathrm{EDB}, \vec{C}_{up}$; We show how to handle the cases $i < L$, $i = L$, and $i = L + 1$ seperately, starting with the former. In all three cases, it starts by parsing EDB as $(B_1, \ldots, B_L)$ and $(\vec{C}_1, \ldots, \vec{C}_L, \vec{C}_{L+1})$ (where $\vec{C}_1 = \vec{C}_{up}$).

In the case $i < L$, search decrypts and decodes blobs $B_1, \ldots, B_i$, recovering exactly $i \cdot \mathsf{Cap}(N)$ postings. It then decrypts the update caches $\vec{C}_1, \ldots, \vec{C}_i$, and finds another $m \geq 0$ postings. It computes the results R using the postings found so far. Then it applies our policy to assign $i \cdot \mathsf{Cap}(N)$ of the $i \cdot \mathsf{Cap}(N) + m$ recovered postings to the blobs $1, \ldots, i$; These are reencoded and rencrypted to produce $B'_1, \ldots, B'_i$ (note that B'_1 contains the forward encoding information). It then forward encodes the remaining m evicted postings, and appends those to $\vec{C}_{i+1}$, calling the result $\vec{C}'_{i+1}$. Finally search outputs EDB as $(B'_1, \ldots, B'_i, B_{i+1}, \ldots, B_L)$ and $(\varepsilon, \ldots, \varepsilon, \vec{C}'_{i+1}, \vec{C}_{i+2} \ldots, \vec{C}_{L+1})$.

When $i = L$, search works the same, except that it may produce B'_L with fewer than $\mathsf{Cap}(N)$ postings. When $i = L + 1$, search proceeds as before, except that creates a new blob B_{L+1} holding at most $\mathsf{Cap}(N)$ postings, and a new empty update cache C_{L+2}. If there are more than $(L + 1)\mathsf{Cap}(N)$ postings, then it appends the forward-encoded entries to C_{L+1}.

The leakage profile $\mathcal{L}$ is not much more complicated than before: $\mathcal{L}_{up}$ is the same, outputting the number of postings in the update. The search leakage $\mathcal{L}_{se}$ leaks n (the total number of documents) and N (the total number postings added over all updates) and also i, the level requested. We show formally that this is all that is leaked, assuming encryption is secure.

Theorem 12. *Let Π be the vertically-partitioned size-locked index scheme from Section 5.5.1 using*

a symmetric encryption scheme SE*. Let* $\mathcal{L}$ *be the leakage profile defined above. Then we give a simulator* $\mathcal{S}$ *such that for any adversary* $\mathcal{A}$ *we can build an adversary* $\mathcal{B}$ *such that* $\mathrm{Adv}^{\mathcal{L}}_{\Pi,\mathcal{S}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{cpa}}_{\mathsf{SE}}(\mathcal{B})$ *where* $\mathcal{B}$ *and each algorithm of* $\mathcal{S}$ *run in time that of* $\mathcal{A}$ *plus some small overhead.*

We construct a simulator similar to before: The high-level idea is that the leakage allows for the computation of the (size-locked) plaintext lengths, and the simulator simply replaces all of the ciphertexts in EDB with encryptions of the appropriate number of zeros under a key it maintains as state. Concretely, the first time either algorithm is run, it picks $K \leftarrow\$ \mathsf{KeySp}$ and sets $s.t. \leftarrow (K, \varepsilon)$. Looking ahead, the simulator will use the second component of its state to store the last EDB that it output.

The simulator $\mathcal{S}_{up}$ works exactly as in the previous proof. The search simulator $\mathcal{S}_{se}$ takes input n, N, i and its state $s.t. = (K, \mathrm{EDB})$. It parses EDB as $(B_1, \ldots, B_L)$ and $(\vec{C}_2, \ldots, \vec{C}_{L+1})$, and needs to output and updated EDB. The key observation is that all of sizes of the new blobs (including new ones) in the "real" updated EDB are determined by n, N, i and the sizes of the previous EDB update caches (this includes the size of the new blob and update cache that are occasionally created). This follows by construction, using Cap and our size-locked encoding. Thus the simulator can encrypt the appropriate number of zeros under K to produce an indistinguishable EDB.

Security of horizontally partitioned scheme. Our last scheme works by maintaining P databases $\mathrm{EDB}_1, \ldots, \mathrm{EDB}_P$, each maintained as in the previous scheme, except EDB_j only holds postings with $\mathrm{PRF}(K', w) = j \bmod P$.

Updates are still appended to $\vec{C}_{up}$ and encoded exactly as before, except they are now labeled with $\mathrm{PRF}(K', w) \bmod P$. To fit our formalism, we view the first-level update cache as EDB_j as a subset of $\vec{C}_{up}$ that is labeled with j. The search algorithm takes input $K, q, i, \mathrm{EDB}, \vec{C}_{up}$. It parses $q = (w_1, \ldots, w_t)$. It then runs the previous algorithm on partitions determined by $\mathrm{PRF}(K', w_1), \ldots, \mathrm{PRF}(K', w_t)$. The only difference is that results are computed by merging the postings obtained, but this is not relevant for security.

The leakage functions $\mathcal{L}_{up}, \mathcal{L}_{se}$ work as follows. They are stateful, and when either is run for the first time, it picks a random function g mapping term to $\{1, \ldots, P\}$ as its state. On update input, $\mathcal{L}_{up}$ outputs $(m_1, \ldots, m_P)$, where m_j is the number of postings in δ with a term w such that $g(w) = j$. The search leakage $\mathcal{L}_{se}$ outputs the following:

- The set $Z = \{g(w) \bmod P \mid w \in q\}$, where q is the latest query. This is the set of indexes of partitions relevant to the query.
- For each $j \in Z$, it outputs n_j, N_j, where n_j is the number of documents added with a term w such that $g(w) = j$ and N_j is the total number of postings added with term w such that $g(w) = j$.

We have the following theorem.

Theorem 13. *Let Π be the horizontally-partitioned size-locked index scheme from Section 5.5.2 using a symmetric encryption scheme SE and PRF. Let $\mathcal{L}$ be the leakage profile defined above. Then we give a simulator $\mathcal{S}$ such that for any adversary $\mathcal{A}$ we can build an adversary $\mathcal{B}$ such that $\mathrm{Adv}^{\mathcal{L}}_{\Pi,\mathcal{S}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{cpa}}_{\mathsf{SE}}(\mathcal{B}) + \mathrm{Adv}^{\mathrm{prf}}_{\mathrm{PRF}}(\mathcal{C})$ where $\mathcal{B}, \mathcal{C}$ and each algorithm of $\mathcal{S}$ run in time that of $\mathcal{A}$ plus some small overhead.*

This simulator is a straightforward adaptation of the prior one.

B.2 CTR-DSSE Construction

In this section, we introduce the CTR-DSSE construction, using an existing DSSE construction as black-box, for our target search problem.

DSSE primitive. Our CTR-DSSE construction is built with the DSSE.Search and DSSE.Update protocols defined in existing DSSE works (semicolon separates input/output for client and server):

- $(\sigma'; EDB') \leftarrow \mathsf{DSSE.Update}(K, \sigma, w, \mathsf{posting}; EDB)$: the client takes the secret key K, the per-term counters σ, the term w to update and the information posting to be put in the posting;

the server takes the encrypted search index EDB. The protocol outputs the updated counters σ' to the client, and the updated encrypted search index EDB', added with the new posting for w, to the server.

- $(\mathsf{postings}; \perp) \leftarrow \mathsf{DSSE.Search}(K, \sigma, q; EDB)$: the client takes the secret key K, the per-term counters σ and a query q, and the server takes the encrypted search index EDB. The protocol outputs all the postings that match the query q to the client, and nothing to the server.

Almost all existing DSSE constructions only include document identifiers in posting, and they focus on identifying all the documents that contain the query terms. Next, we describe how to achieve our target search syntax based on the DSSE protocols.

Construction. Algorithm B.1 shows the details of the CTR-DSSE. CTR-DSSE executes DSSE. Update with the new postings to update the encrypted search index (Line 4), and DSSE.Search with the query and page number to retrieve all the postings matching the query (Line 8). For the per-term counters σ, CTR-DSSE uploads it in ciphertext to the server (Line 6), and downloads it when necessary (Line 2, 7). For the ranked previews, CTR-DSSE augments the posting in DSSE with the term frequency and document meta-data (Line 4), and retrieves this information, together with the identifiers, to rank and preview the results (Line 11).

We adopt Diana with 8 byte posting label, and 4 byte encoded counters for the underlying DSSE primitives. The identifier, term hashes and metadata are encoded in the same way as in our size-locked encoding.

Security of CTR-DSSE. For CTR-DSSE, $\mathcal{L}_{up}$ outputs the number of words in current, but not any previous, update, and the number of postings in the update. $\mathcal{L}_{se}$ outputs the leakages of the underlying DSSE construction, *i.e.*, query pattern and result pattern for Diana.

Theorem 14. *Let* Π *be* CTR-DSSE *with construction and* $\mathcal{L}$ *above, using a symmetric encryption scheme* SE. *Then we give a simulator* $\mathcal{S}$ *such that for any adversary* $\mathcal{A}$ *we can build an adversary* $\mathcal{B}$ *such that* $\mathrm{Adv}^{\mathcal{L}}_{\Pi,\mathcal{S}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{cpa}}_{\mathsf{SE}}(\mathcal{B})$ *where* $\mathcal{B}$ *and each algorithm of* $\mathcal{S}$ *run in time that of* $\mathcal{A}$ *plus some small overhead.*

$\underline{\mathsf{Update}(K, \mathsf{id}, \delta)}$:

1: $K_1 \| K_2 \leftarrow K$; $\mathsf{md}, (w_i, \mathsf{tf}_i)_{i=1}^m \leftarrow \delta$
2: Download and decrypt (using K_1) σ from server
3: **for** $i = 1, \ldots, m$ **do**
4: $\quad (\sigma'; EDB') \leftarrow \mathsf{DSSE.Update}(K_2, \sigma, w_i, (\mathsf{id}, \mathsf{tf}_i, \mathsf{md}); EDB)$
5: $\quad \sigma \leftarrow \sigma'; EDB \leftarrow EDB'$
6: Encrypt (using K_1) and upload σ to server

$\underline{\mathsf{Search}(K, q, i, \perp, \perp)}$:

7: Download and decrypt σ; Initialize empty tables TFs, Metas
8: $(\mathsf{postings};) \leftarrow \mathsf{DSSE.Search}(K_2, \sigma, q; EDB)$
9: **for** $w \in q$ **do**
10: $\quad$ **for** posting $\in$ postings$[w]$ **do**
11: $\quad\quad \mathsf{id}, \mathsf{tf}, \mathsf{md} \leftarrow \mathsf{posting}$
12: $\quad\quad \mathsf{TFs}[\mathsf{id}, w] \leftarrow \mathsf{TFs}[\mathsf{id}, w] + \mathsf{tf}; \mathsf{Metas}[\mathsf{id}] \leftarrow \mathsf{md}$
13: Use TFs to rank results, and return page i of results with entries from Metas

Figure B.1: CTR-DSSE construction.

Optimization. CTR-DSSE can be optimize to download only the counters for the query terms at Search, without affecting the search leakage. The idea is to store the per-term counters separately, and download the counters only for the query terms at Search. Such optimization does not affect the leakage because it only reveals query pattern on the query terms, which is already included in $\mathcal{L}_{se}$. Note that, at Update, all the separate counters have to be downloaded at the beginning and re-uploaded at the end, whether the associated word is changed or not because, otherwise, it would leak the updated words, and make the construction not forward private.

We can further optimize to decouple the metadata from the postings using a forward index from identifiers to the encrypted per-document metadata. Accessing metadata of matched identifiers only leaks the result pattern, which is included in $\mathcal{L}_{se}$ already.

B.3 Experimental Queries

Here are the single-keyword queries that we used to evaluate the search performance of the constructions in Section 5.7. Each keyword is followed by its document frequency in parentheses. f

- Enron: enron (246814), thank (191266), message (130359), attach (108190), work (97741), meet (76706), market (71063), review (63604), energy (63016), schedule (62857), friday (54685), contract (49902), product (44437), talk (42498), california (38701), credit (35100), financial (34715), transaction (33037), industry (25828), policy (20413), invest (17785), portland (17232), georgia (2138), tokyo (2154), japan (4870), china (2259), chinatown (32), rolex (55), piano (178), slave (178)
- Ubuntu: ubuntu (127005), help (42044), linux (28063), system (19892), driver (15561), version (14018), firefox (8899), debian (4901), mysql (1781), radeon (1100), winxp (715), wikipedia (535), gpg (436), invalid (389), udev (329), httpd (273), warcraft (251), rss (224), checksum (195), askubuntu (164), openssl (142), benchmark (125), tracerout (110), smbpasswd (85), selinux (65), noacpi (56), vsync (44), byobu (39), ipsec (36), openprint (31)
- NYTimes: president (79991), government (65339), research (27026), drug (22189), criminal (13861), terrorism (13959), protest (9902), employee (8054), revolution (6271), diplomatic (5247), girlfriend (4402), firefighter (3768), courtroom (3273), classified (2685), securities (2296), zzz_michael_jordan (2062), diabetes (1847), zzz_lincoln (1636), yankee (1460), heroin (1261), zzz_mit (1146), cereal (1058), zzz_lieberman (983), zzz_starbuck (912), cinnamon (842), zzz_rocky (779), volleyball (730), zzz_gucci (641), nerd (486), tapestry (399)

B.4 Extended Security Analysis

In this section, we present an analysis of the leakage profiles of the basic sizelocked index, vertical partitioning (VP), horizontal partitioning (HP), and vertical horizontal partitioning (VHP) along

with comparisons to the leakage of a standard forward private DSSE scheme. Furthermore, we discuss and analyze the power and practicality of possible attacks against our construction.

B.4.1 Leakage Across Various Settings

DSSE. In all circumstances, DSSE search will leak the length of the posting list associated with a query as well as query equality, that is, the server can tell when two queries are associated with the same keyword. Furthermore, search will leak the result pattern, that is, which documents contain the queried keyword. Note that this result pattern is leaked even if a user does not request the associated document from the server.

Updates will leak the size of the udpate was well as the number of new keywords added.

Vertical Partitioning. In all circumstances, a search in the vertical partitioning scheme will leak the requested page of results and partition cutoffs for page numbers. Updates will leak the size of the update. The leakage of partition cutoffs can open the possibility of a Cap undershooting attatck, which we detail in section D.2. If we assume a user that looks specifically for top honesetly ranked results, then the vertical partitioning scheme is vulnerable to a ranking deflation attack, which we detail in section D.4. Assuming a user that clicks through all pages of results, the number of requested pages can leak the length of the posting list of the queried keyword to the granularity of the page length.

Horizontal Partitioning. Updates and searches for a keyword will leak the partition to which a keyword belongs. Under the circumstance that a keyword is never queried after being added to the index, horizontal partitioning has strictly worse leakage than DSSE because horizontal partitioning will leak the partition to which the keyword belongs. The CTR-DSSE construction will have no such access pattern leakage for updates because it is forward private.

If however a keyword is searched after it is added, then the access pattern leakage of DSSE will be at least as granular as the access pattern leakage at the partition level of horizontal partitioning.

Note that if a user requests every document in the list of results served by a horizontally partitioned index, our leakage becomes the same as that of DSSE because the document requests leak the full result pattern .

B.4.2 Cap Undershooting

Recall from Section 5.1 that the number of postings in each vertical partition is determined using the Cap function and is therefore dependent only on the number of postings in the index, which we are willing to leak. Under ideal circumstances the Cap function outputs a partition size that can accommodate at least the first page of results for all keywords in the index. However, if there are many unique terms relative to the number of postings, formalized by Heap's Law as a heuristic, then the first page of certain keywords may be forced to spill over to the second partition. In particular, the Cap function will undershoot when $\mathsf{Cap}(N) < \sum_{w \in W} \min(l(w), k)$ where W is the full set of keywords, $l(w)$ is the length of the posting list of keyword w and k is the page size. The right hand side of the inequality corresponds to the total number of postings across all of the first pages of terms within the index. Such a scenario would lead to two partitions being fetched instead of just one, upon loading the first page of results. This distinction enables a file-injection attack that can ascertain information about the initial state of the index.

Consider a scenario in which an adversary wishes to distinguish between an index containing a single document containing the single word ”dog” versus an index containing a single document with just the word ”cat”. In order to do so, the adversary can begin by injecting documents to attain the condition that $\mathsf{Cap}(N) < \sum_{w \in W} \min(l(w), k)$ only if an additional posting is added to the index. Of the injected postings, exactly $k - 1$ will be postings for “cat”.

The adversary then injects a final “cat” posting, which will force some “cat” posting to the second partition. If there were a “cat” posting in the original index, then there will be $k + 1$ total “cat” postings, so a single “cat” posting will be evicted to the second partition onto the second page of the posting list. Note that when a user searches for a keyword, the first page of results is

automatically loaded for the user and enough partitions are fetched to accommodate the first page of any keyword in the index.

Therefore when the first page is served to a user, just the first partition will be fetched if and only if “cat” was originally in the index. If “cat” were not originally in the index, then the first page of the posting list of “cat” will be split between the second and first partitions, leading to two partitions being fetched. This enables the adversary to win the security game with non-negligible advantage and infer the contents that the index initially held.

One naturally may wonder if Cap undershooting can be used to mount a query recovery attack. Because the partition fetching policy for loading page i will request enough partitions to have page i of any keyword in the index, the number of partitions fetched by itself leaks no information about the keyword in the query. If one were to modify the partition fetching policy to load enough partitions to load results for page i of just the queried keyword, then a query recovery attack is possible.

B.4.3 Timing Attack on the Lookup Table

As described in Section 5.3, the keyword table contains offsets within the serialized index to the posting lists of keywords, allowing the client to avoid deserializing the full index to answer individual search queries. The size of the keyword table, as with the partition sizes in VP, is determined using the Cap function. When the number of terms is sufficiently large relative to the number of postings, the keyword table cannot accommodate all terms in the index. Therefore, when a keyword is not found in the keyword table, the entire index must be deserialized, which leads to a considerable increase in query response time. In this way, timing information leaks when keywords that aren’t in the lookup structure or in the index at all are searched. This leads to a query recovery attack if the adversary can ensure that particular keywords $\{w_1, \ldots, w_r\}$ do not make it into the lookup table because the lookup table was packed with injected keywords. This leads to at most one bit of leakage (fast vs. slow). It is likely the case that noise from the manner in which

users make queries will mask this timing side channel.

Under the ambitious assumption that the adversary is able to send queries, a timing attack can be used to ascertain what keywords are in the index. If the adversary sees that their queries are answered quickly, the adversary can deduce that the keywords in the queries are in the index. If a query takes a long time to be answered [reword] then the full index must have been deserialized, indicating that the keyword is not in the lookup structure and likely not in the index.

The above two attacks assume that the adversary is able to observe query response times on the client. The frequency with which a user searches for different keywords can be a proxy for these response times, however this is likely to be a noisy signal. In order for an adversary to accurately measure the response time this way, it must be the case that the user immediately searches for a new keyword once obtaining the results for the last keyword.

B.4.4 Ranking Deflation Attack

Posting lists are shown to the user across a number of pages. The page in which a result occurs is determined by a ranking function (e.g. BM25, TF-IDF). On average, users are expected to find what they are looking for among the first pages, ideally on the first page itself. Recall that in vertical partitioning, the number of partitions accessed leaks the page number requested. If an adversary observes that later pages are requested for a particular query, then the adversary may deduce that the queried keyword has low-quality rankings for the first pages of results.

The adversary can inject files to deflate the ranking of the first page of honest high-ranked results and replace them with lower quality results for a keyword w^*. Assuming that users will keep clicking for results until they find at least one honest high-ranked result, then this gives a query recovery attack for w^*.