

Realizing Privacy Design Strategies for Back-end Web Architectures

by

Miti Mazmudar

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

Waterloo, Ontario, Canada, 2025

© Miti Mazmudar 2025

Examining Committee Membership

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

External Examiner: Sébastien Gambs
Professor in Computer Science
Université du Québec à Montréal (UQAM)

Supervisor(s): Ian Goldberg
Professor
Cheriton School of Computer Science

Internal Members: Xi He
Associate Professor
Cheriton School of Computer Science

Bernard Wong
Associate Professor
Cheriton School of Computer Science

Internal-External Member: Mahesh Tripunitara
Professor
Electrical and Computer Engineering

Author's Declaration

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

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

Statement of Contributions

The contents of Chapter ?? were previously published in a paper [MHL+23] and a technical report [MHL+22] with Thomas Humphries, Jiaxiang Liu, Matthew Rafuse and Xi He. Thomas, Xi and I collaboratively wrote this paper. All authors were involved in the development of our MMM and RP modules (?? and ??), and collaborated on other modules as follows. I developed the Strategy Generator (??) and the Proactive Querying (??) modules. Thomas worked on the Strategy Expander module (??), and developed the Full-Rank Transformer (??) with Jiaxiang. Thomas, Jiaxiang and I extended these two modules for workloads with multiple attributes (??). Jiaxiang conducted our evaluation (??), in collaboration with Thomas. Matthew conceptualized and experimented with our MMM and RP modules in the early stages of our work.

The contents of Chapter ?? were also previously published in a paper [MGG21a] and a technical report [MGG21b], both of which were written in collaboration with Stan Gurtler and Ian Goldberg. Stan, Ian and I wrote our background (??), analyzed work in robust DHTs (??) and developed our design (??). Stan and Ian formulated our threat model (??). I performed our security analysis and a simulation to evaluate our security-performance tradeoff (??), in collaboration with Ian. Stan and I collaborated on our performance analysis ??, in collaboration with Ian.

The contents of Chapter ?? were co-authored in collaboration with Shannon Veitch and Rasoul Akhavan Mahdavi [MVM25]. I formulated our background and threat model. Shannon and I developed our routing table normalization algorithm. Shannon conducted a simulation to verify the proof. Shannon and I developed our private peer routing algorithm. All of us collaborated on our private provider routing algorithm. Rasoul and I developed our private content retrieval protocol. Rasoul benchmarked existing schemes and refined existing PIR algorithms for application to each of our algorithms. I conducted the evaluation of our algorithms and developed the security proofs in conjunction with Ian.

All other chapters in this dissertation contain new material developed under the supervision of Ian Goldberg.

Abstract

In today's big data world, companies collect extensive profiles of users' personal information. This kind of extensive data collection risks exposing users' personal information in larger, more severe and frequent data breaches. Furthermore, software development practices lead developers to processing this sensitive information for purposes orthogonal to the core functionality offered by the business. Users' sensitive data has often been shared with business partners and other third parties in violation of the business' privacy policy, and in a breach of the users' trust. Hoepman proposes eight privacy design strategies, in order to guide software developers to incorporate privacy as a central element within the software development process. These privacy design strategies underpin the design of, and draw examples from, various privacy-enhancing technologies (PETs) that are deployed today, including differential privacy, anonymity networks and so on. Nevertheless, software engineers and privacy engineers have found that realizing these privacy design strategies within the back-end web architectures that they maintain remains challenging. Some privacy design strategies, such as the INFORM and CONTROL strategies, are commonly realized, by informing users through privacy policies and equipping them with controls. However, the remaining strategies are not commonly translated into practice.

We consider different back-end web architectures, including both centralized and distributed web architectures, and each contribution in this thesis operationalizes one or more privacy design strategies for the respective back-end architecture. In our first system, namely CacheDP, we examine relational DBMSes, and reduce the granularity of sensitive data revealed to data analysts using these systems, thereby realizing the ABSTRACT strategy. In our second and third systems, namely DHTPIR and Peer2PIR, we delve into peer-to-peer networks and distributed hash tables. Both of these contributions seek to minimize the data revealed to other peers while a peer queries for, and retrieves, files from the network, and thus, they realize the MINIMIZE strategy. However, both systems do so differently; DHTPIR separates the sensitive content of the query across multiple peers in the network, thereby realizing the SEPARATE strategy, while Peer2PIR hides the content of the query using encryption, and thus operationalizes the HIDE strategy. In our final system, namely Prose, we focus on a distributed microservice architecture, conceptualizing technical requirements that cater towards the needs of privacy engineers and software engineers, and implements the ENFORCE and DEMONSTRATE strategies for this architecture. Considering the MINIMIZE strategy as an example, we distinguish between PETs that realize a privacy design strategy once within the underlying architecture, without further interventions as the architecture evolves, as is

the case with DHTPIR and Peer2PIR, with PETs that realize a strategy in a way such that the PET can respond to changes in the architecture, as in Prose. We thus relate our privacy-enhancing technologies to the privacy design strategies that they realize, highlighting combinations of privacy design strategies that are realized simultaneously, and illustrating how some privacy design strategies may be used to indirectly realize others. Importantly, our work shows that PETs can be designed to systematically and continually realize privacy design strategies, such as the ENFORCE, DEMONSTRATE and MINIMIZE strategies. Finally, we evaluate our privacy-enhancing technologies and show that they are effective, in terms of continuing to providing the desired functionality but with the added privacy guarantee afforded by the privacy design strategy, and efficient, in terms of low communication and computation overheads over the original architecture.

Acknowledgements

I would like to thank my supervisor, Ian Goldberg, for his advice, patience and unending support throughout my PhD. I also thank my committee members, Xi He, Bernard Wong, Mahesh Tripunitara and Sébastien Gambs, for their time and feedback.

I am grateful to Nikita for his steadfast support and for reminding me to regularly take a break from my computer! I thank my parents for persistently encouraging me throughout my PhD. I am inspired by their constant willingness to help others. I thank Larisa and Oleg for always providing moral support. I appreciated the fun gaming sessions with Kostia.

I would like to thank all of my collaborators — this thesis would not be possible without their constant support. I thank Thomas and Shane for their unwavering support and for hosting great BBQs. I am inspired by Rasoul’s indefatigable enthusiasm in making Peer2PIR “just a little bit better” and I am grateful for Shannon’s eye for detail. Bailey and I started our graduate school journeys together. I benefited tremendously from her advice, as well as from our many long walks. I thoroughly enjoyed working with Kyle on our project. Cecylia and Justin have been a pillar of support throughout my entire grad school and continue to inspire me to this day. I found a lifelong friend in Lindsey and I thank her for always lending a listening ear. I also enjoyed the fun times at RBG with Lindsey, Ogadi and Kiernan. I appreciated the numerous discussions with Nik on cryptography. Our D&D sessions have been a breath of fresh air and I look forward to them in the near future. I thank Sajin for his help in navigating post-PhD career paths, and I am grateful to Camila and Masoumeh for fun crafts nights. Akshar has kept me in the loop with pop culture and math memes, and I have enjoyed the constant nerdsnipping.

I had the privilege of teaching the CS458 course with Asokan, and I am immensely grateful for this opportunity to learn from him. I am thankful to the staff of the CUT program, Svitlana Taraban, Kristin Brown and Monica Soczewinski, for their timely help and advice.

This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo. My research into Peer2PIR has been funded by the RFP-014: Private retrieval of data grant from Protocol Labs (PL). I am grateful to Will Scott from PL for his support throughout this project. I also thank Dennis Trautwein and Guillaume Michel from PL for their assistance in integrating Peer2PIR algorithms within the LibP2P library. Finally, I am grateful to the entire team at NSD KW!

Table of Contents

List of Figures	xiii
List of Tables	xv
List of Algorithms	xvi
1 Introduction	1
1.1 Back-end web architectures	3
1.2 Data minimization	4
1.3 Contributions	7
1.3.1 CacheDP	8
1.3.2 DHTPIR and Peer2PIR	8
1.3.3 Prose	10
2 CacheDP	12
2.1 Introduction	12
2.2 Background	14
2.3 System Design	18
2.3.1 Cache Structure Overview	20
2.3.2 Strategy Transformer (ST) Overview	21
2.3.3 Cache-aware DP Modules	23

2.4	Modified Matrix Mechanism (MMM)	25
2.4.1	MMM Overview	26
2.4.2	Answer Workload Interface	27
2.4.3	Estimate Privacy Budget Interface	30
2.5	Strategy Modules	36
2.5.1	Strategy Transformer	36
2.5.2	Strategy Expander	43
2.5.3	Proactive Querying	48
2.6	Relax Privacy Mechanism	53
2.6.1	Estimate Privacy Budget Interface	53
2.6.2	Answer Workload Interface	55
2.7	Multiple Attribute Workloads	56
2.8	Evaluation	57
2.8.1	Experimental Setup	58
2.8.2	Privacy Budget Evaluation	60
2.8.3	Ablation Study	62
2.8.4	Overhead Evaluation	65
2.9	Related Work	69
2.10	Conclusion	71
3	DHTPIR	72
3.1	Introduction	73
3.2	Background and Related Work	75
3.2.1	Private Information Retrieval	75
3.2.2	Distributed Hash Tables	76
3.2.3	Censorship-resistant publishing	78
3.3	Robust DHTs	79
3.4	Threat model	82

3.5	Our Design	84
3.5.1	Building blocks	84
3.5.2	Our interface	86
3.6	Security Parameters Analysis	97
3.7	Complexity analysis	100
3.8	Evaluation	102
3.9	Deployment	107
3.10	Conclusion	107
4	Peer2PIR	109
4.1	Introduction	110
4.2	Background	111
4.2.1	IPFS	112
4.2.2	Private Information Retrieval	113
4.3	Related Work	116
4.4	Threat Model	117
4.5	Private Peer Routing	119
4.5.1	Kademlia and the IPFS Routing Table	119
4.5.2	Routing Table with Normalized Buckets	125
4.5.3	Convergence under Normalized Routing Tables	127
4.5.4	Private Algorithm Integration	132
4.6	Private Provider Advertisements	134
4.6.1	IPFS Provider Advertisements	134
4.6.2	Private Algorithm Integration	135
4.7	Private Content Retrieval	139
4.7.1	Bitswap Protocol	139
4.7.2	The Private Bitswap Protocol	140
4.8	PIR Protocols	142

4.8.1	Existing PIR protocols	143
4.8.2	Tailored PIR Protocols: PAILLIERPIR and RLWEPIR	145
4.8.3	Correctness and Security of PAILLIERPIR	151
4.8.4	Security of Private Provider Advertisements	157
4.9	Implementation and Privacy Engineering	160
4.10	Evaluation	165
4.11	End-to-end latency and churn analysis	169
4.11.1	End-to-end latency analysis	170
4.11.2	Churn analysis	172
4.12	Conclusion	174
5	Prose	175
5.1	Introduction	176
5.2	Background	178
5.2.1	Distributed microservices architectures	178
5.2.2	Orchestration frameworks	179
5.2.3	Software Development Lifecycle (SDLC)	180
5.2.4	General requirements	182
5.3	Privacy policy workflow and violations	183
5.3.1	Iterative privacy policy workflow	184
5.3.2	Privacy engineering requirements	189
5.3.3	Privacy policy violations	189
5.4	Threat model	193
5.5	Design components	196
5.5.1	Service proxies	196
5.5.2	PII identification	198
5.5.3	Authorization agents	199
5.5.4	Distributed tracing	199

5.6	Data plane design	201
5.6.1	Mediating data flows for each violation	201
5.6.2	Detecting PII types	202
5.6.3	Detecting and preventing violations	202
5.6.4	Communicating from data to control plane	203
5.7	Control plane design	205
5.7.1	Forming observed profiles	205
5.7.2	Privacy operators	209
5.7.3	Analysis Engine	212
5.8	Evaluation	215
5.8.1	Implementation	215
5.8.2	Evaluating efficiency	217
5.8.3	Evaluating effectiveness	220
5.9	Related work	227
5.9.1	Formal language-based approaches to data minimization	227
5.9.2	Policy enforcement systems	229
5.10	Conclusion	232
6	Conclusion	234
6.1	Defense of the thesis statement	234
6.2	Future work	237
	References	239

List of Figures

2.1	CacheDP System Diagram	19
2.2	Strategy Decomposition Example.	22
2.3	Baseline approach for the ESTIMATEPRIVACYBUDGET interface in MMM.	31
2.4	Noise parameters searched during discrete and continuous binary searches in the ESTIMATEPRIVACYBUDGET interface.	33
2.5	Strawman approach for the Strategy Expander module.	44
2.6	Example of the SEARCHPROACTIVENODES function (??).	52
2.7	Comparison of the cumulative privacy budget spent by CacheDP and baselines across various tasks.	61
2.8	Ablation study over PQ, RP, SE modules of CacheDP.	64
2.9	Cumulative cache size and average runtime of the MMM, versus the workload index, for the IDEBench task.	67
2.10	Scatterplot to measure the dependence of the runtime of the MMM with WA^+	69
3.1	Quorum formation protocols.	80
3.2	DHTPIR API interfaces	85
3.3	Layers of protocols used within DHTPIR.	90
3.4	DHTPIR Join protocol	95
3.5	Total communication complexity of one DHTPIR query for one chunk.	104
3.6	Throughput and latency for DHTPIR.	105

4.1	Indistinguishability games for the security of a PIR scheme and the semantic security of a PKE scheme.	152
4.2	Black-box reduction from an adversary against the PIR security of PAILLIERPIR to an adversary against IND-CPA of the Paillier PKE cryptosystem.	154
4.3	Communication and runtime overheads for private peer routing.	166
4.4	Communication and computation overheads for private provider advertisements.	167
4.5	Communication and computation overheads for private peer routing.	168
5.1	Example of microservices managed by an orchestrator (Kubernetes) and a service mesh (Istio)	179
5.2	Iterative privacy policy workflow	185
5.3	Prose: Example of inter-service communication	190
5.4	Using inbound and outbound service proxies to detect different types of privacy policy violations	198
5.5	Example of Prose span tags	204
5.6	Response latency of Prose data plane	218
5.7	Prose: Receiver Operating Characteristic of our PII identification unit	224

List of Tables

1.1	Privacy design strategies realized in each contribution.	7
2.1	Notation for CacheDP.	16
2.2	Evaluation setup for CacheDP.	59
2.3	Frequency analysis study for CacheDP	63
2.4	Cache size and workload runtime comparison for CacheDP and baselines.	65
2.5	Indices of attributes included in each workload in the IDEBench task for the overhead evaluation (Figures ??)	68
3.1	Related work comparison for DHTPIR	81
3.2	Overview of the DHTPIR API	87
3.3	Notation for security constraints for DHTPIR	98
4.1	Client and server privacy properties offered by Peer2PIR	118
4.2	Examples of Kademia and Peer2PIR routing responses.	123
4.3	Notation for Peer2PIR.	134
4.4	Communication overheads for existing PIR schemes for use in Peer2PIR.	145
5.1	Queries used to analyze violations of the system-wide profile with a given privacy policy.	213
5.2	Prose data plane implementation components	216
5.3	Prose effectiveness experiment setup: Frequency of PII types in the ground truth and components of the classifier for each PII type.	222
5.4	Comparison of related work for Prose.	226

List of Algorithms

1	CacheDP Overview	24
2	Modified Matrix Mechanism (MMM): ANSWERWORKLOAD interface . . .	27
3	MMM: ESTIMATEPRIVACYBUDGET interface and helper function	34
4	Strategy Transformer (ST).	39
5	Full-rank Transformer (Section ??)	41
6	Strategy Expander (SE) (Section ??)	48
7	Proactive Querying (PQ) (Section ??)	50
8	RP: ESTIMATEPRIVACYBUDGET interface	54
9	RP: ANSWERWORKLOAD interface	55
10	DHTPIR file search function.	89
11	DHTPIR file insertion function.	91
12	Routing Table Normalization Algorithm	126
13	Private provider advertisements: QUERY, PREPROCESS and RESPONSE functions.	137
14	Private Provider Advertisements: EXTRACT function.	138
15	PRIVATEBLOCK using PIR	141
16	PAILLIER operations	146
17	PAILLIERPIR: Query function.	147
18	PAILLIERPIR: Response and Extract functions.	148
19	RLWEPIR: Query function.	149
20	RLWEPIR: Response function.	150
21	GENQRY function within ??	155
22	Indistinguishability game for symmetric PIR for Private Provider Adver- tisements.	158

Chapter 1

Introduction

In today’s big data paradigm, companies extensively collect their customers’ personal data through websites and apps [Bro20] and share it with many third parties. Collecting large amounts of personally identifiable information (PII) on users often inculcates a mindset of “collect now, use later” among developers, which in turn leads to this PII being processed or shared with third parties for purposes unrelated to the core functionality offered by the business. For instance, online mental health services such as BetterHelp and Crisis Text Line have shared their users’ personally identifiable information such as email addresses, their mental health history, and their usage of the service, with third parties [Fai23; Lev22]. Users may perceive the act of sharing their data with third parties, even if it is in a de-identified form, as a privacy violation [New22]. Similarly, Twitter obtained its users’ phone numbers for the purpose of improving security, such as two-factor authentication, and then used these phone numbers for targeted advertising. Twitter has been charged by the Federal Trade Commission (FTC) of continuing to deceptively use the users’ phone numbers for secondary purposes for five years [Fed22]. Indeed, from the perspective of users, the purposes for which websites and apps collect data items, or share them with third parties, often do not match with their mental models [LAH+12; RSS+16]. Following the Cambridge Analytica revelations [Mer18], users experienced a lack of agency as their data was appropriated for purposes that they did not expect, including surveillance, and was shared with or sold to third parties that they were unaware of [SM20; ARA+19].

For businesses, the benefits of collecting and processing large amounts of data may be outweighed by a multitude of risks. As businesses continue to collect more user data and retain it for longer periods of time, users’ data is exposed in larger, more severe and frequent data breaches [MEB+22]. Data breaches may result in financial

and legal ramifications for the impacted business. Furthermore, extensive data aggregation and poor processing practices risk alienating users who are concerned about these practices, or losing them altogether from the service [OFI21]. With large troves of data, businesses also risk violating data privacy legislation such as the Personal Information Protection and Electronic Documents Act (PIPEDA) [Off19] which governs Canadian businesses' data processing practices, the General Data Protection Regulations (GDPR) [Int18c] which governs the processing of European Union residents' data, and the California Consumer Privacy Act (CCPA) [Sta23] for California residents. Organizations are incentivized to follow regulations such as the GDPR, in order to avoid large fines [STUD22].

Data minimization can help businesses alleviate such risks, by systematically reducing the amount of data within their back-end systems and safeguarding the processing of such data [SN20; Dav23]. Data minimization refers to the use of design strategies that minimize the amount of data collected, shared, and linked by an information system, as well as strategies that minimize the amount of time for which data is retained [GTD15] [BBB+22, Pg. 30]. The privacy-by-design philosophy encourages software developers to incorporate privacy early and throughout the software development lifecycle [Cav11]. Hoepman [Hoe14; Hoe22] identifies eight software design patterns, namely privacy design strategies, which guide developers to realize privacy by design within their software development lifecycle. The principle of purpose limitation, one of the cornerstones of data minimization, states that the type and granularity of any personal data collected should be the minimum that is necessary for the purpose of collection [Section 3.1][Hoe14]. The aforementioned examples involve processing users' data for non-core purposes of use, and extensively sharing it with third parties, thereby ultimately violating the principle of purpose limitation.

Data protection authorities have recognized harms to users as a result of these privacy violations and consequently, have required organizations to follow data minimization strategies during their development process, in order to identify and potentially prevent such harms. The FTC has cited data minimization in its decisions [CE22]. It has also cited clauses pertaining to purpose limitation from the EU-U.S. Data Privacy Framework [Adm24] in its enforcement actions against businesses centered in the EU [Fed24b, Pg. 23]. As such, data minimization has been included in several proposed data privacy legislations, such as the American Data Privacy and Protection Act [But24] and American Privacy Rights Act [Wil24]. The Office of the Privacy Commissioner of Canada has recommended encoding the purpose limitation principle within the Digital Charter Implementation Act of 2022 [Par24; Duf23]. Data minimization at large, as well as purpose limitation in particular, have been codified in the GDPR [Com18], and have

been cited in enforcement decisions [Uni23; SJL+23].

Nevertheless, the FTC has found that businesses lack technical tools to operationalize data minimization in practice [Fed24a, Section IV-B, IV-D]. This finding is corroborated through user studies involving privacy engineers and software engineers, as we detail later [TLV22; HDG+24]. Existing privacy-enhancing technologies are insufficient at effectively and efficiently realizing data minimization across such modern back-end architectures. Therefore, we propose frameworks to realize privacy design strategies towards data minimization across modern back-end architectures—that is, to build and integrate privacy-enhancing technologies (PETs) that automatically apply these strategies into the software development process—in order to identify and potentially prevent the aforementioned harms.

We present the following thesis statement:

It is possible to build and integrate efficient and effective PETs that realize privacy design strategies within back-end web architectures.

In this thesis, we illustrate this statement through four main systems, namely CacheDP, DHTPIR, Peer2PIR and Prose.

1.1 Back-end web architectures

Websites are supported by different, constantly evolving web architectures. Web architectures typically consist of front-end and back-end components. Front-end components execute within the user’s web browser and determine how the user interacts with and navigates the website. Back-end components implement the core functionality offered by the website, by processing users’ data. In this work, we focus on back-end web architectures as opposed to front-end web architectures. Our contributions towards this thesis statement focus on different back-end architectures.

Websites have evolved from having monolithic back ends to microservices [Lou21; New20; Wea23]. Each microservice focuses on a certain business functionality or purpose of use. For example, an online e-commerce shop [Goo23] might have CHECKOUT, SHIPPING, AUTHENTICATION and ADVERTISING services. Cloud computing technologies, such as Kubernetes [Kub23] and Istio [Ist23b], facilitate deploying microservices on cloud computing platforms, such as GCP [Goo24b] and Amazon EC2 [Ama24a]. Over 25 open source and proprietary cloud computing projects have been graduated

by the Cloud Native Computing Foundation (CNCF), that is, they are well-maintained, production-ready codebases with widespread adoption, while over 150 projects are in the process of graduation [Clo24a; Clo24b]. Microservices are ubiquitous; big tech companies such as Google [McL16], Amazon [Ama19], Netflix [Var21], and Spotify [Gol15] drive the development of microservices-based architectures, and contribute to the aforementioned cloud computing technologies. Additionally, public and private sector enterprises around the world increasingly use microservices and adopt these technologies [IBM21; Clo23]. Our Prose system focuses on a microservices-based architecture.

Relational DBMSes such as Postgres [Pos24] and MySQL [Ora24] are commonly used to store users' data in databases. These DBMSes can themselves be deployed as microservices for a microservices-based back end [Goo24b]. Alternately, cloud-based data platforms, such as Snowflake [Sno24a], allow businesses to efficiently store and query large databases on top of cloud computing platforms. Our CacheDP system focuses on relational DBMSes.

Cloud computing platforms are centralized in that all machines that serve the website's back-end components are hosted by one entity, such as Google or Amazon. Peer-to-peer networks can be used to develop truly decentralized and distributed web architectures, since peers are located in different administrative regions. Popular networks such as the Interplanetary File System (IPFS) [HSE19] serve as the storage and computational back end for emerging distributed web applications. The IPFS ecosystem [Pro23a] includes many projects that facilitate building such applications. For example, Fleek [Fle22b] serves as a decentralized content distribution network backed by IPFS. Space Daemon [Fle22a] enables building distributed applications that process users' data privately on their own IPFS nodes. Our DHTPIR and Peer2PIR systems focus on peer-to-peer networks.

1.2 Data minimization

We begin with describing privacy design strategies, and literature on data minimization. We then discuss barriers for software engineers to apply these strategies.

We now outline Hoepman's privacy design strategies [Hoe14; Hoe22]. The MINIMIZE strategy refers to intentionally limiting the collection and processing of personal data; Hoepman discusses limiting this collection based on the purpose of use of the data. The HIDE strategy suggests protecting personal data (access control mechanisms, data confidentiality through encryption), or making it unlinkable (data or metadata unlinkability). The ABSTRACT strategy refers to reducing the granularity of the personal

data processed, through summarization, grouping, and perturbations. The SEPARATE strategy advises processing personal data items over different physical locations using distributed systems, and preferring client-side computations over collecting and disclosing all users' data to a single entity.

The INFORM strategy advocates supplying fine-grained data processing practices to users, using layered designs, and notifying users in real time of data breaches. The CONTROL strategy recommends obtaining users' consent before processing their data in tandem with the INFORM strategy, while providing them with basic functionalities in case they decline this consent. Users should also be enabled to withdraw this consent and to update and delete their data, to address modern subject access requests [Int18a]. The ENFORCE strategy refers to enforcing a commitment of processing personal data in a privacy-friendly way. In tandem with the ENFORCE strategy, the DEMONSTRATE strategy advises organizations to demonstrate compliance, namely that users' personal data is processed in a privacy-friendly way, to privacy regulators. Hoepman also enumerates several *tactics* to implement each of the aforementioned strategies. As tactics for the ENFORCE strategy, Hoepman directs organizations to CREATE a privacy policy, MAINTAIN or implement that policy through technical or organization controls and UPHOLD the privacy policy as the organization evolves. Hoepman's examples for this strategy include organization-level measures such as privacy impact assessments (PIAs). The DEMONSTRATE strategy consists of three tactics, namely logging system actions (RECORD tactic), auditing the logs (AUDIT tactic) and reporting the audit results to data protection authorities (REPORT tactic). Hoepman exemplifies this strategy through anomaly detection and sticky policies, which govern the processing of data items and are transmitted along with the data item.

Gürses et al. [GTD15] approached data minimization as minimizing the risk of data breaches and the need to trust external entities. They then deconstruct data minimization into strategies that minimize collection, disclosure, replication, retention, and linkability of data, through designs that avoid centralization; that is, single points of failure. They emphasize that while software engineers may first collect many data items and then select which data items to use for a specific purpose, privacy experts design systems that only collect data necessary towards an end purpose.

Data minimization in practice. While Hoepman's privacy design strategies encapsulate data minimization and purpose limitation [Hoe14, Pg. 457], they remain difficult to apply for non-experts who may want to integrate PETs into systems. Indeed, examples for the design strategies in Hoepman's handbook [Hoe22] range in complexity

from integrating standard symmetric encryption into the product, to using anonymity networks such as Tor. To assess whether Hoepman’s privacy design strategies can be used by developers, Tahaei et al. [TLV22] analyze advice for developers on the popular Stack Overflow forum, and classify it in terms of these strategies. The authors find that most advice fell into the INFORM and CONTROL strategies (such as privacy policies and permissions), whereas the HIDE and MINIMIZE strategies were less frequent and the remaining four strategies (ABSTRACT, SEPARATE, ENFORCE, and DEMONSTRATE) were rarely advised.

Tahaei et al. attribute the infrequency of the HIDE and MINIMIZE strategies to the fact that large platforms such as Google or Apple only indirectly suggest these strategies to developers. As an example of the MINIMIZE strategy, since collecting sensitive data items requires informing users through privacy policies and obtaining permissions from them through consent dialog boxes, developers were advised to not collect these items and thereby avoid dealing with privacy policies and consent dialog boxes. Similarly, developers may conduct data minimization while restructuring their databases. Tahaei et al. thus hypothesize that the MINIMIZE strategy may be applied *reactively*, to avoid legal issues or in the process of database restructuring. They expect that since the ENFORCE and DEMONSTRATE strategies require organizational support, they are rarely included in technical advice to developers.

More recently, through their user study, Horstmann et al. [HDG+24] found that software engineers and privacy engineers lack tooling to translate privacy concepts such as data minimization into source code. There are no common automated mechanisms for privacy engineers to glean data processing practices; privacy engineers rely on questionnaires and optionally, privacy impact assessments, to gather information about the data collected and maintained by, as well as the purpose of a service that is implemented by, a team and then respond back to the software engineers with privacy requirements. For a large system, relying on these manual processes may lead to outdated information about the service, as documentation inevitably drifts from source code. Importantly, software engineers, as non-experts, are left to actually implement complicated privacy requirements, such as automating the right to access data that is stored in separate sources, through scripting. In this process, they juggle with vaguely specified requirements, including data minimization. Software engineers also experience challenges in verifying that their implementation works through testing, and maintaining their systems as their source code evolves. The authors emphasize the need for the academic community to address these technical challenges, at scale. We observe that doing this would reduce replicated effort on the part of software engineers implementing privacy tooling, across different teams within an organization and across organizations.

TABLE 1.1: Privacy design strategies realized in each contribution.

System	Back-end Architecture	Web	Privacy Design Strategies
CacheDP (Chapter ??)	Relational DBMS		Reduce the granularity of data, or abstract it, through DP perturbations (ABSTRACT strategy).
DHTPIR (Chapter ??)	Distributed hash tables		Minimize the data collected by node operators, in order to answer search queries (MINIMIZE strategy); sensitive information is split across multiple nodes (SEPARATE strategy).
Peer2PIR (Chapter ??)	Distributed hash tables		MINIMIZE strategy as above. Also hides the content of the search query, through homomorphic encryption (HIDE strategy).
Prose (Chapter ??)	Distributed microservices		Minimize data collection and disclosure (MINIMIZE strategy). Demonstrate compliance to a target privacy policy (DEMONSTRATE strategy) and enforce the policy (ENFORCE strategy). Inform users of the data processing practices based on the source code (INFORM strategy).

Similarly, developing tooling catered to privacy engineers to automatically glean what a given service is collecting and processing, would also reduce the communication effort for privacy engineers to keep up with multiple teams, and eliminate outdated information about the service.

1.3 Contributions

Each contribution in this thesis operationalizes different privacy design strategies, and in doing so, it targets different aspects of data minimization that are relevant to the respective back-end architecture, as outlined in Table ?? . We also highlight key technical challenges for each contribution below. We associate privacy design strategies with the PET used in each contribution. We also highlight combinations of strategies that are commonly realized through the same PET across different contributions. Additionally, in our final contribution, we reconceive two strategies, which focus on organizational robustness, in terms of technical requirements for privacy engineers and software engineers. For each contribution, we also outline our notions of efficiency and effectiveness for the respective PET. We detail each system in its chapter; we group the second and

third contributions together in our discussion below, as they focus on the same back-end web architecture.

1.3.1 CacheDP

In the first contribution (Chapter ??), we consider a centralized relational database management system (DBMS). Organizations often maintain large databases with users' sensitive data. Frequently, data analysts from a different organization may conduct aggregate queries over these databases to derive insights about customers. However, a curious data analyst may use these query responses to infer sensitive data items of an individual user in a statistical inference attack. In this contribution, instead of minimizing the amount or types of data collected or shared, we seek to protect users' data in databases, while it is being queried by data analysts. We prevent these attacks by using differential privacy (DP) to release perturbed or noisy aggregate responses, thereby applying the data minimization strategy of reducing the granularity of released statistics. As such, DP instantiates the ABSTRACT privacy design strategy.

DP determines the amount of noise in, and thus the accuracy of, the query responses through a *privacy budget*. Spending more privacy budget on a query leads to more accurate responses, which is *worse* for the users' privacy, since these responses can be exploited by an analyst in a statistical inference attack. Alternately, an effective DP system would spend the least privacy budget to meet predetermined accuracy requirements. Thus, we measure the effectiveness of our system that uses DP, as a PET, through its consumption of the privacy budget at the given accuracy requirements, and its efficiency in terms of computation overheads.

Importantly, in this contribution, we demonstrate that features that already exist within the architecture, such as caching in DBMSes, can be extended to improve the effectiveness of the PET. We develop CacheDP, a DP system that provides differentially private responses to analysts' queries for relational DBMSes. CacheDP uses its cache to save privacy budget in comparison to state-of-the-art DP systems. In doing so, we address novel technical challenges such as structuring the cache to optimize reuse, and providing DP responses that satisfy analysts' accuracy requirements.

1.3.2 DHTPIR and Peer2PIR

In the second and third contributions (?? and ??), we consider a distributed architecture of peer-to-peer networks. Peer-to-peer networks such as the Interplanetary File System (IPFS) are widely used today. Unlike a traditional client-server architecture, wherein

clients conduct queries and servers respond to them, each peer in peer-to-peer network functions as both a client and a server in that it can both respond to and conduct queries. Both the content stored on the network and the routing information enabling peers to locate one another, are distributed across peers in a peer-to-peer network. In other words, neither does a single peer store all content, nor does it know the network addresses of all peers in the network. Thus, while functioning as a client, peers conduct a content retrieval query to find desired content or a routing query to find the network address of a given peer. Correspondingly, the queried peers, which function as servers, maintain tables, such as routing tables and content stores, to facilitate answering these queries. A client contacts multiple server peers in the process of answering each query, before it ultimately finds the desired content or peer. Importantly, the servers learn what content the client is attempting to fetch or who it is attempting to contact.

Thus, in these two contributions, we seek to minimize the data collected by peers who operate as servers, in the process of answering these queries. We want to ensure that the content of the query is only known to the peer acting as a client. Private Information Retrieval (PIR) is a PET designed to enable a client to retrieve a record from a table shared by one or more servers, while preventing the servers from knowing which record was retrieved. PIR can thus serve as a PET to hide the content of the query. PIR allows us to limit what information is collected by peers acting as servers, in order to answer a client's query, and thus enables us in realizing the MINIMIZE strategy within these contributions. PIR schemes incur communication and computation overheads on top of the non-private status quo, namely directly retrieving the records from the servers and consequently revealing which record was retrieved. We thus measure the efficiency of both of these contributions in terms of these two types of overheads.

These two contributions differ in that they rely on different types of PIR schemes. In particular, information-theoretic PIR schemes (IT-PIR) separate the information about a client's query across a quorum of multiple server peers, such that these server peers need to collude together to learn this information. DHTPIR uses IT-PIR schemes and in doing so, realizes the SEPARATE strategy for peer-to-peer networks. Instead of a single server peer responding to a client's query, as in the original peer-to-peer network, a quorum of server peers respond in DHTPIR. The effectiveness of DHTPIR is thus contingent on forming well-behaving groups with sufficient peers that behave honestly by not colluding. The key technical challenge that DHTPIR addresses is to form reasonably sized quorums, while reducing the aforementioned overheads for efficiency.

On the other hand, in Peer2PIR, a single server peer continues to respond to a client's query, but the underlying PIR schemes, namely computational PIR schemes, assume that the server cannot solve computationally hard problems. In particular, we rely on

homomorphic encryption to hide the content of the query from the server, and as such, Peer2PIR additionally realizes the HIDE strategy. While IT-PIR schemes have lower overheads than CPIR schemes, they require majority-honest quorums, which add to the overheads for DHTPIR and require peers to provide more storage overhead by a multiplicative factor; this leads us to our choice of CPIR schemes for Peer2PIR, as we discuss later. Nevertheless, composing computational PIR schemes within peer-to-peer networks is also challenging, and we address these challenges within Peer2PIR. CPIR schemes have been designed for a traditional client-server setting, and implicitly assume that the client repeatedly queries the same server peer. However, this assumption is broken within peer-to-peer networks, since a client contacts different peers in the process of answering both content retrieval and routing queries. Additionally, these schemes are optimized for certain table sizes, and one scheme cannot fit all of the aforementioned types of queries. In Peer2PIR, we highlight and address these design challenges, and adapt existing CPIR schemes for IPFS. Our proofs of security of our CPIR schemes demonstrate their effectiveness.

1.3.3 Prose

In the final contribution (??), we consider a distributed microservices architecture. Medium and large-sized organizations task privacy managers or engineers to ensure that their distributed systems handle users' data in a privacy-preserving manner or that these systems comply with privacy legislation. Privacy engineers could use existing PETs towards data minimization in this architecture. As outlined above, many PET designs use cryptographic or information-theoretic techniques to ultimately minimize the data exposed to certain adversaries in a specific threat model.

However, in a microservices-based architecture, simply eliminating the use of a data item at one service by implementing a PET, may not be sufficient to eliminate collecting that data item from the client altogether. Users' data items are processed by multiple services, each of which has different functionalities or purposes of use. Thus, for privacy managers to provide meaningful privacy guarantees over users' data, they require knowledge of the entire distributed back-end system; that is, how each microservice handles users' data. The privacy manager may then seek to minimize data collection and disclosure, by ensuring all services stick to a strict target privacy policy. To this end, we build a Privacy Observability and Enforcement system, namely Prose. We measure the effectiveness of Prose in terms of its false positives and negatives in detecting violations. We measure its efficiency in terms of its computation overheads on top of the distributed microservices architecture. We explicate requirements for privacy engineers and software engineers to easily use Prose. Key technical challenges include develop-

ing a scalable design that can operate over microservices written in multiple languages, without incurring significant effort on part of software engineers, such as requiring them to rewrite their code for our frameworks to work. As we describe later, Prose realizes the ENFORCE strategy, and can serve as an auditing framework, thereby realizing the DEMONSTRATE strategy. Importantly, Prose enables privacy engineers to continually realize the MINIMIZE strategy, even as the microservices' source code evolves.

Chapter 2

CacheDP

We focus on relational DBMSes as our back-end web architecture for this chapter. Organizations use relational DBMSes to maintain large datasets that contain users' sensitive data. They may permit data analysts, who may potentially be affiliated with another organization, to query these datasets for aggregate statistics. However, a curious data analyst may use these query responses to infer a user's record. Differential Privacy (DP) [DMNS06; DR14] allows organizations to provide a guarantee to their users that the presence or absence of their record in the dataset will only change the distribution of the query response by a small factor, depending on the *privacy budget*. This guarantee is typically achieved by perturbing the query response with noise that is inversely proportional to the privacy budget. DP is thus a PET that realizes Hoepman's ABSTRACT strategy. DP systems face an accuracy-privacy trade-off: they provide accurate query responses, at the cost of spending a large privacy budget. DP is offered as a built-in feature in DBMSes such as Snowflake [Sno24a]. DP has been deployed at the US Census Bureau [MKA+08], Google [WZL+20] and Microsoft [DKY17]. In this chapter, we delve into the technical challenges of composing a PET that realizes Hoepman's ABSTRACT strategy, namely DP, on top of a back-end web architecture, namely a relational DBMS.

2.1 Introduction

Existing DP deployments [MKA+08; BEM+17; DKY17; KTH+19] mainly consider a non-interactive setting, where the analyst provides all queries in advance. In contrast, in interactive DP systems [McS10; JNS18; WZL+20; GHV19], data analysts supply queries one at a time. These systems have been difficult to deploy as they often assume an

analyst has DP expertise. First, data analysts need to choose an appropriate privacy budget per query. Second, data analysts require each DP noisy query response to meet a specific accuracy criterion, whereas DP systems only seek to minimize the expected error over multiple queries. Ge et al.’s APEx [GHIM19] eliminates these two drawbacks, as data analysts need only specify accuracy bounds in the form of an error rate α and a probability of failure β . APEx chooses an appropriate DP mechanism and calibrates the privacy budget spent on each workload, to fulfill the accuracy requirements. However, interactive DP systems may run out of privacy budget for a large number of queries.

We observe that we can further save privacy budget on a given query, by exploiting *past*, related noisy responses, and thereby, we can answer a larger number of queries interactively under the same total privacy budget. The DP post-processing theorem allows arbitrary computations on noisy responses without affecting the DP guarantee. Hay et al. [HRMS10] have applied this theorem to enforce consistency constraints among noisy responses to related range queries, thereby improving their accuracy, through *constrained inference*. Peng et al. [PYZ+13] have proposed caching noisy responses and reusing them to answer future queries in Pioneer. However, their cache is unstructured and only operates with simple DP mechanisms such as the Laplace mechanism.

We design a usable interactive DP query engine, CacheDP, with a built-in differentially private cache, to support data analysts in answering data exploration workloads accurately. Our system is built on top of an existing non-private DBMS and interacts with it through standard SQL queries. Instead of requiring analysts to specify the privacy budget to spend on each query, they can bound the error in the DP response to that query in the form of an (α, β) accuracy requirement. CacheDP meets the analysts’ accuracy requirement on each workload, while minimizing the privacy budget spent per workload. We note that a similar reduction in privacy budget could be obtained if an expert analyst planned their queries, however our system removes the need for such planning.

Our contributions address four main challenges in the design of our engine. First, we structure our cache to maximize the possible reuse of noisy responses by DP mechanisms in Section ?? . Our cache design fully harnesses the post-processing theorem in the interactive setting, for cached noisy responses. Second, we integrate existing DP mechanisms with our cache, namely Li et al.’s Matrix Mechanism [LMH+15], and Koufogiannis et al.’s Relax Privacy mechanism [KHP16]. In doing so, we address technical challenges that arise due to the need to maintain accuracy requirements over cached responses while minimizing the privacy budget, and thus, we provide a novel privacy budget cost estimation algorithm.

Third, we extend our cache-aware DP mechanisms with two modules, which further reduce the privacy budget. Specifically, we apply DP sensitivity analysis to proactively fill our cache, and we apply constrained inference to increase cache reuse. We note that CacheDP internally chooses the DP module with the lowest privacy cost per workload; data admins or analysts do not need to tweak CacheDP to select the best DP module. We present our Modified Matrix Mechanism in Section ??, our optional modules in Section ??, and our Relax Privacy module in Section ?. Fourth, we develop the design of our cache to handle queries with multiple attributes efficiently in Section ?.

Finally, we conduct a thorough evaluation of our CacheDP against related work (APEX, Pioneer), in terms of privacy budget consumption and performance overheads in Section ?. We find that it consistently spends lower privacy budget as compared to related work, for a variety of workload sequences, while incurring modest performance overheads. Through an ablation study, we deduce that our standard configuration with all DP modules turned on, is optimal for the evaluated workload sequences. Thus, researchers implementing our system need not tinker with our module configurations.

2.2 Background

We consider a single-table relational schema $\mathcal{R}$ across d attributes: $\mathcal{R}(\mathcal{A}_1, \dots, \mathcal{A}_d)$. The domain of an attribute $\mathcal{A}_i$ is finite and is given by $\text{dom}(\mathcal{A}_i)$, where $|\text{dom}(\mathcal{A}_i)| = n_i$. Thus, the full domain of $\mathcal{R}$ is an ordered list given by $\text{dom}(\mathcal{R}) = \text{dom}(\mathcal{A}_1) \times \dots \times \text{dom}(\mathcal{A}_d)$ and is of size $n = \prod_i n_i$. A database instance D of relation $\mathcal{R}$ is a multiset whose elements are values in $\text{dom}(\mathcal{R})$. Consider $\text{dom}(\mathcal{R})$ to be an ordered list. We represent a database instance D by a column vector $\mathbf{x}$ of length n , where $\mathbf{x}[i]$ is the count of i th value from $\text{dom}(\mathcal{R})$ in D .

A predicate $\phi : \text{dom}(\mathcal{R}) \rightarrow \{0, 1\}$ is an indicator function that specifies the database rows that an analyst is interested in. A *linear or row counting query* (RCQ) takes a predicate ϕ and returns the number of tuples in D that satisfy ϕ , i.e., $\phi(D) = \sum_{t \in D} \phi(t)$. This corresponds to querying `SELECT COUNT(*) FROM D WHERE  $\phi$` in SQL. We focus on RCQs for this work as they are primitives that can be used to express histograms, multi-attribute range queries, marginals, and data cubes.

In this work, we express RCQs using matrices. We represent a single RCQ as a length- n row vector $\mathbf{w}$, with $\mathbf{w}[i] \in \{0, 1\}$ for $i = 1, \dots, n$. Specifically, $\mathbf{w}$ is an indicator vector, and thus, we represent a range $[p, q]$ as a row vector with a 1 at column j if $j \in [p, q]$. To obtain the ground truth response for a RCQ $\mathbf{w}$, we can simply compute $\mathbf{w} \cdot \mathbf{x}$. Hence, we can represent a workload of ℓ RCQs as a $\ell \times n$ matrix $\mathbb{W}$ and answer

this workload by matrix multiplication, as $\mathbb{W}\mathbf{x}$.

Most elements of the domain will be irrelevant for answering a given workload, and thus, we can transform the data vector $\mathbf{x}$ into another, ideally smaller data vector $\mathbf{x}$ that consists of $n' \leq n$ disjoint buckets. We can similarly map the workload matrix $\mathbb{W}$ over the full domain $\text{dom}(\mathcal{R})$ to a matrix $\mathbf{W}$ of size $\ell \times n'$, respectively. Since we consider a workload matrix $\mathbf{W}$ as a *set* of RCQs, applying a set operator over a workload matrix is equivalent to applying this operator over a set of RCQs. For example, $\mathbf{W}' \subseteq \mathbf{W}$ implies that the set of RCQs in $\mathbf{W}'$ is a subset of the RCQs in $\mathbf{W}$. We summarize key notation in Table ??.

Definition 2.2.1 (ϵ -Differential Privacy (DP) [DMNS06]). A randomized mechanism $M : \mathcal{D} \rightarrow \mathcal{O}$ satisfies ϵ -DP if for any output sets $O \subseteq \mathcal{O}$, and any *neighboring* database pairs (D, D') , i.e., $|D \setminus D' \cup D' \setminus D| = 1$,

$$\Pr[M(D) \in O] \leq e^\epsilon \Pr[M(D') \in O]. \quad (2.1)$$

The privacy parameter ϵ is also known as the privacy budget. A classic mechanism to achieve DP is the Laplace mechanism. We present the matrix form of Laplace mechanism below.

Theorem 2.2.1 (Laplace mechanism [DMNS06; LMH+15]). Given an $\ell \times n'$ workload matrix $\mathbf{W}$ and a data vector $\mathbf{x}$, the Laplace Mechanism $\mathcal{L}_b$ outputs $\mathcal{L}_b(\mathbf{W}, \mathbf{x}) = \mathbf{W}\mathbf{x} + \text{Lap}(b)^\ell$ where $\text{Lap}(b)^\ell$ is a vector of ℓ independent, identically distributed samples from a Laplace distribution with scale b . If $b \geq \frac{\|\mathbf{W}\|_1}{\epsilon}$, where $\|\mathbf{W}\|_1$ denotes the L_1 norm of $\mathbf{W}$, then $\mathcal{L}_b(\mathbf{W}, \mathbf{x})$ satisfies ϵ -DP.

Li et al. [LMH+15] present the matrix mechanism, which first applies a DP mechanism, M , on a new matrix $\mathbf{A}$, which is known as the strategy matrix, and then post-processes the noisy answers to the queries in $\mathbf{A}$ to estimate the queries in $\mathbf{W}$. This mechanism aims to achieve a smaller error than directly applying the mechanism M on $\mathbf{W}$. We use the Laplace mechanism $\mathcal{L}_b$ to illustrate matrix mechanism in Example ?. As we outline in this example and discuss later, $\mathbf{A}$ will have linearly independent columns and will thus be a full column rank matrix. We define the Moore–Penrose pseudoinverse of a matrix below.

Definition 2.2.2. The Moore-Penrose pseudoinverse of an $m \times n$ matrix A is given by the matrix A^+ that satisfies the following constraints [LMH+15, Definition 17]:

$$AA^+A = A, A^+AA^+ = A^+, (AA^+)^T = AA^+, (A^+A)^T = A^+A$$

TABLE 2.1: Notation for CacheDP.

Notation	Description
$\mathbf{x}, \mathbf{w}, \mathbb{W}, \mathbb{A}$	raw data vector, query vector, query workload matrix, strategy matrix over full domain $\text{dom}(\mathcal{R})$
$\mathbf{x}, \mathbf{w}, \mathbf{W}, \mathbf{A}$	mapped data vector, query vector, query workload matrix, strategy matrix over a subset of $\text{dom}(\mathcal{R})$
α, β	accuracy parameters for $\mathbb{W}$
$\mathcal{B}, B_c, \epsilon$	total budget, consumed budget, workload budget
$\mathbb{A}^*, \mathcal{C}_{\mathbb{A}^*}$	global strategy matrix, its cache over $\text{dom}(\mathcal{R})$
$b, \tilde{y}$	a scalar noise parameter, a scalar noisy response
$\mathbf{b}$	a vector of noise parameters
$\tilde{\mathbf{y}}, \tilde{\mathbf{z}}$	a vector of noisy responses to the strategy $\mathbf{A}$ or to the workload $\mathbf{W}$.
$(\mathfrak{a}, b, \tilde{y}, t)$	a cache entry for a strategy query $\mathfrak{a} \in \mathbb{A}^*$ stored at timestamp t . See Definition ??.
$\mathbf{F}, \mathbf{P}$	free strategy matrix, paid strategy matrix

In particular, the Moore–Penrose pseudoinverse of a full column rank matrix $\mathbf{A}$ is defined as follows:

$$\mathbf{A}^+ = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T$$

Since $\mathbf{A}^T \mathbf{A}$ will be invertible when $\mathbf{A}$ is full column rank, we have that $\mathbf{A}^+$ is the left inverse of $\mathbf{A}$, as follows:

$$\mathbf{A}^+ \mathbf{A} = (\mathbf{A}^T \mathbf{A})^{-1} (\mathbf{A}^T \mathbf{A}) = \mathbf{I}$$

Definition 2.2.3 (Matrix Mechanism (MM) [LMH+15]). Given an $\ell \times n$ workload matrix $\mathbf{W}$, a $p \times n$ strategy matrix $\mathbf{A}$, and the Laplace mechanism $\mathcal{L}_b(\mathbf{A}, \mathbf{x})$ that answers $\mathbf{A}$ on $\mathbf{x}$, the matrix mechanism $\mathcal{M}_{\mathbf{A}, \mathcal{L}_b}$ outputs the following answer: $\mathcal{M}_{\mathbf{A}, \mathcal{L}_b}(\mathbf{W}, \mathbf{x}) = \mathbf{W} \mathbf{A}^+ \mathcal{L}_b(\mathbf{A}, \mathbf{x})$, where $\mathbf{A}^+$ is the Moore–Penrose pseudoinverse of $\mathbf{A}$. We denote $\mathcal{L}_b(\mathbf{A}, \mathbf{x})$ by $\tilde{\mathbf{y}}$ and $\mathcal{M}_{\mathbf{A}, \mathcal{L}_b}$ by $\tilde{\mathbf{z}}$.

Example 2.2.1. In a database schema $\mathcal{R}$, consider an attribute $\mathcal{A}$ with an ordered, integer domain $\text{dom}(\mathcal{A}) = [0, 8)$. The analyst asks an RCQ of the form: `SELECT COUNT(*) FROM  $\mathcal{R}$  WHERE  $\mathcal{A} < 7$ .` We represent this workload query as $\mathbf{w} = [0, 7)$. We also form its corresponding matrix, $\mathbb{W}$, of size $|\mathbf{w}| \times |\text{dom}(\mathcal{A})| = 1 \times 8$; this matrix only consists of zeroes and ones, as shown below. Consider the query $\mathfrak{a} = \{[0, 4), [4, 6), [6, 7)\}$. We refer to this as a *strategy query*, since the responses to this query can be linearly combined to answer $\mathbf{w}$. Indeed, a ground truth response to $\mathbf{w}$ will be the same as the sum of the responses to $\mathfrak{a}$. (Other choices of $\mathfrak{a}$, such as $\mathfrak{a} = \{[0, 5), [5, 7)\}$, can also be used to answer $\mathbf{w}$. Our choice of strategy queries is instrumental to our cache design, as we

detail in Section ??, and illustrate in Example ??.) We represent each range $[p, q)$ in $\mathfrak{a}$ as an $\text{dom}(\mathcal{A})$ -sized indicator vector, similar to $\mathfrak{w}$. We thus represent $\mathfrak{a}$ as a matrix $\mathbb{A}$, of size $|\mathfrak{a}| \times |\text{dom}(\mathcal{A})|$, similar to $\mathbb{W}$. ($\mathbb{A}$ also consists of only zeroes and ones.)

$$\begin{aligned}\mathfrak{x}^\top &= [\mathfrak{x}[0, 1) \quad \mathfrak{x}[1, 2) \quad \dots \quad \mathfrak{x}[j-1, j)] \\ \mathbb{W} &= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \\ \mathbb{A} &= \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}\end{aligned}$$

To obtain the ground truth response for $\mathbb{W}$, we can simply compute $\mathbb{W} \cdot \mathfrak{x}$.

Instead of $\mathfrak{x}^\top$, consider the following data vector $\mathbf{x}^\top$, and its associated workload matrix ($\mathbf{W}$) and strategy matrix ($\mathbf{A}$). These are the full-rank forms of the aforementioned matrices, and we describe how they are generated later on.

$$\mathbf{x} = \begin{bmatrix} \mathfrak{x}[0, 4) \\ \mathfrak{x}[4, 6) \\ \mathfrak{x}[6, 7) \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Intuitively, each workload query in $\mathbf{W}$ can be represented as a linear combination of strategy queries in $\mathbf{A}$, i.e., $\mathbf{W}\mathbf{x} = \mathbf{W}\mathbf{A}^+(\mathbf{A}\mathbf{x})$, where $\mathbf{A}^+$ is the Moore-Penrose pseudoinverse of the full-rank matrix $\mathbf{A}$. Then, given the noisy responses to $\mathbf{A}$, through the Laplace mechanism, $\tilde{\mathbf{y}} = \mathbf{A}\mathbf{x} + \boldsymbol{\eta}$, we can compute the responses to $\mathbf{W}$ as $\tilde{\mathbf{z}} = \mathbf{W}\mathbf{A}^+\tilde{\mathbf{y}}$. As the matrix mechanism post-processes the output of a DP mechanism [DR14], it also satisfies ϵ -DP, when the noise parameter for the noisy responses in $\tilde{\mathbf{y}}$ satisfy the following bound.

Proposition 2.2.1 ([LMH+15]). If $b \geq \frac{\|\mathbf{A}\|_1}{\epsilon}$, then $\mathcal{M}_{\mathbf{A}, \mathcal{L}_b}$ satisfies ϵ -DP.

We would like to allow data analysts, who may not be able to choose an appropriate budget for a DP mechanism, to specify their accuracy requirements for their workload. We consider two popular error specifications for DP mechanisms.

Definition 2.2.4. Given an $\ell \times n$ workload matrix $\mathbf{W}$ and a DP mechanism M , (i) the α^2 -expected total squared error bound [LMH+15] is

$$\mathbb{E}[\|\mathbf{W}\mathbf{x} - \mathbf{W}\mathbf{A}^+M(\mathbf{W}, \mathbf{x})\|_2^2] \leq \alpha^2 \quad (2.2)$$

and (ii) the (α, β) -worst error bound [GHIM19] is defined as

$$\Pr[\|\mathbf{W}\mathbf{x} - \mathbf{W}\mathbf{A}^+M(\mathbf{W}, \mathbf{x})\|_\infty \geq \alpha] \leq \beta. \quad (2.3)$$

The error for the matrix mechanism is $\|\mathbf{W}\mathbf{A}^+Lap(b)^\ell\|$, which is independent of the ground truth data. This allows us to directly estimate the error bound without running the mechanism on the data. For example, Ge et al. [GHIM19] provide a loose bound for the noise parameter in the matrix mechanism to achieve an (α, β) -worst error bound.

Theorem 2.2.2 ([GHIM19]). The matrix mechanism $\mathcal{M}_{\mathbf{A}, \mathcal{L}_b}$ satisfies the (α, β) -worst error bound, if

$$b \leq b_L = \frac{\alpha\sqrt{\beta/2}}{\|\mathbf{W}\mathbf{A}^+\|_F} \quad (2.4)$$

where $\|\cdot\|_F$ is the Frobenius norm, which is the square root of the sum of squares of all elements in the matrix.

When we set b to this loose bound b_L , the privacy budget consumed by this mechanism is $\frac{\|\mathbf{A}\|_1}{b_L}$. To minimize the privacy cost, Ge et al. [GHIM19] conduct a binary search over noise parameters larger than b_L . The filtering condition for this search is whether the sampled error term $\|\mathbf{W}\mathbf{A}^+Lap(b)^\ell\|_\infty$, which is output by a Monte Carlo simulation, exceeds α with a probability less than or equal to β . This search outputs a noise parameter $b^* \geq b_L$, that satisfies the (α, β) -requirements, at the minimal privacy cost of $\frac{\|\mathbf{A}\|_1}{b^*}$.

2.3 System Design

We design an interactive inference engine with a built-in cache, CacheDP, that supports data analysts in answering data exploration queries under differential privacy with sufficient accuracy. The system architecture is given in Figure ?? . We follow a differential privacy model with a trusted data owner. The data owner instantiates an unmodified relational DBMS such as MySQL, with a database that includes sensitive data. To complete the setup stage, the data owner also provides a total privacy budget $\mathcal{B}$ to our system. At runtime, the data analyst inputs a workload $\mathbb{W}$, and an (α, β) accuracy requirement that the workload should satisfy, to CacheDP. Our system interacts with the DBMS, via an SQL interface, and a cache $\mathcal{C}$, to return a differentially private workload response $\tilde{\mathbf{z}}$, which satisfies this accuracy requirement, to the analyst. The i -th workload response consumes some privacy budget ϵ_i , out of $\mathcal{B}$, and the goal of CacheDP is to

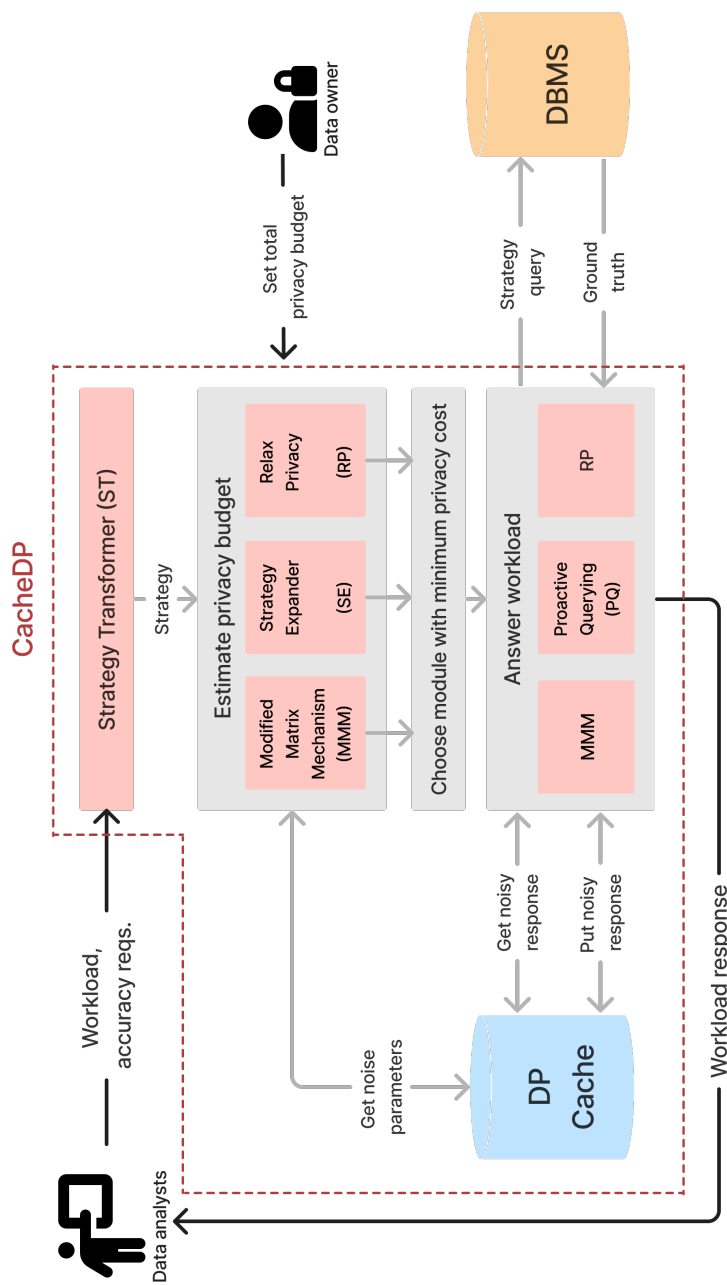


FIGURE 2.1: CacheDP System Diagram

reduce ϵ_i by using our cache, which stores historical noisy responses. The analyst continues this process with a series of workloads. We provide an overview of our system design in this section, while motivating our description through design challenges. Our system follows a *modular design*, in order to enable DP experts to develop new cache-aware, problem-specific modules in the future.

2.3.1 Cache Structure Overview

Our cache stores previously released noisy DP responses and related parameters; it does not store any sensitive ground truth data. Moreover, the cache does not interact directly with the DBMS at all. Therefore, the cache design evolves independently of the DBMS or other alternative data storage systems. We consider two design questions: (i) which queries and their noisy responses should be stored in the cache; and (ii) what other parameters are needed?

A naive cache design simply stores all historical workloads, their accuracy requirements and noisy responses $[(\mathbb{W}_1, \alpha_1, \beta_1, \tilde{\mathbf{z}}_1), \dots, (\mathbb{W}_t, \alpha_t, \beta_t, \tilde{\mathbf{z}}_t)]$. When a new workload $(\mathbb{W}_{t+1}, \alpha_{t+1}, \beta_{t+1})$ comes in, the system first uses the cache to infer a response $\tilde{\mathbf{z}}'_{t+1}$ and its error bound $(\alpha'_{t+1}, \beta'_{t+1})$. If its error bound is better than the accuracy requirement, that is, $\alpha'_{t+1} \leq \alpha_{t+1}$ and $\beta'_{t+1} \leq \beta_{t+1}$, then the workload response can be reused. Otherwise, the error bound is worse than the accuracy requirement, that is, $\alpha'_{t+1} > \alpha_{t+1}$ or $\beta'_{t+1} > \beta_{t+1}$, then additional privacy budget ϵ_{t+1} needs to be spent to improve $\tilde{\mathbf{z}}'_{t+1}$ to $\tilde{\mathbf{z}}_{t+1}$. This additional privacy cost ϵ_{t+1} should be smaller than a DP mechanism that does not use historical query answers.

This cache design is used in Pioneer [PYZ+13], but it has several drawbacks. First, this design results in a cache size that linearly increases with the number of workload queries. Second, we will not be able to *compose and reuse* cached past responses to overlapping workloads $(\mathbb{W}_{t-k} \cap \mathbb{W}_t \neq \emptyset)$. Simply put, this design works with only simple DP mechanisms, which answer the data analyst-supplied workloads directly with noisy responses. For instance, Pioneer [PYZ+13] considers only single-query workloads and the Laplace mechanism. We seek to design a reusable cache that can work with complex DP mechanisms, and in particular, the matrix mechanism. Thus, we need to *structure* our cache such that cached queries and their noisy responses can be reused efficiently, in terms of the additional privacy cost and runtime, while limiting the cache size.

Our key insight is that the strategy matrices in Matrix Mechanism in Def ?? can be chosen from a structured set. So, we store noisy responses to the matrix that the mechanism answers directly (the strategy matrix), instead of storing noisy responses that are post-processed and returned to the data analyst (the workload matrix). If all

strategy matrices share a similar structure, then we need to only track noisy responses to a limited set of queries in our cache. Relatedly, since the (α, β) accuracy requirements for different workload matrices can only be composed through a loose union bound, we instead track the noise parameters that are used to answer the associated strategy matrices. Thus, in our cache, we store the strategy queries, the noisy strategy query responses and the noise parameters.

This cache design motivates us to consider a global strategy matrix $\mathbb{A}^*$ for the cache that can support all possible workloads. Importantly, for a given workload matrix $\mathbb{W}$, we present a strategy transformer (ST) module to generate an instant strategy matrix, denoted by $\mathbb{A}$, such that each instant strategy matrix is contained in the global strategy matrix, i.e., $\mathbb{A} \subseteq \mathbb{A}^*$. In this design, the cache tracks each strategy entry $\mathfrak{a} \in \mathbb{A}^*$, with its noisy response, its noise parameter, and the timestamp.

Definition 2.3.1 (Cache Structure). Given a global strategy matrix $\mathbb{A}^*$ over the full domain $\text{dom}(\mathcal{R})$, a cache for differentially private counting queries is defined as

$$\mathcal{C}_{\mathbb{A}^*} = \{\dots, (\mathfrak{a}, b, \tilde{y}, t), \dots \mid \mathfrak{a} \in \mathbb{A}^*\}, \quad (2.5)$$

where b and $\tilde{y}$ are the latest noise parameter and noisy response for the strategy query $\mathfrak{a}$, and t is the time stamp for the latest update of $\mathfrak{a}$. At beginning, all entries are initialized as $(\mathfrak{a}, -, -, 0)$, where ‘-’ denotes invalid values. For simplicity, we use $\mathcal{C}$ instead of $\mathcal{C}_{\mathbb{A}^*}$ within the rest of this chapter.

In this work, $\mathbb{A}^*$ corresponds to a hierarchical structure, namely a k -ary tree over $\text{dom}(\mathcal{A}_i)$. This is a popular strategy matrix for MM [LMH+15, Section 6]. Figure ?? shows the global strategy matrix as a binary tree decomposition of a small integer domain $[0, 8)$, which is used in our examples.

2.3.2 Strategy Transformer (ST) Overview

We outline the Strategy Transformer (ST) module, which is commonly used by all of our cache-aware DP modules. The ST module consists of two components: a Strategy Generator (SG) and a Full-Rank Transformer (FRT). Prior work [LMH+15] uses the global strategy $\mathbb{A}^*$, which has a high $\|\mathbb{A}^*\|_1$. Given an input $\mathbb{W}$, the SG selects an instant strategy $\mathbb{A} \subseteq \mathbb{A}^*$, with a low $\|\mathbb{A}\|_1$, among other criteria. Though $\mathcal{C}$ is structured based on $\mathbb{A}^*$, it is not searched while generating $\mathbb{A}$. We generate instant strategies for two workloads, in the example below.

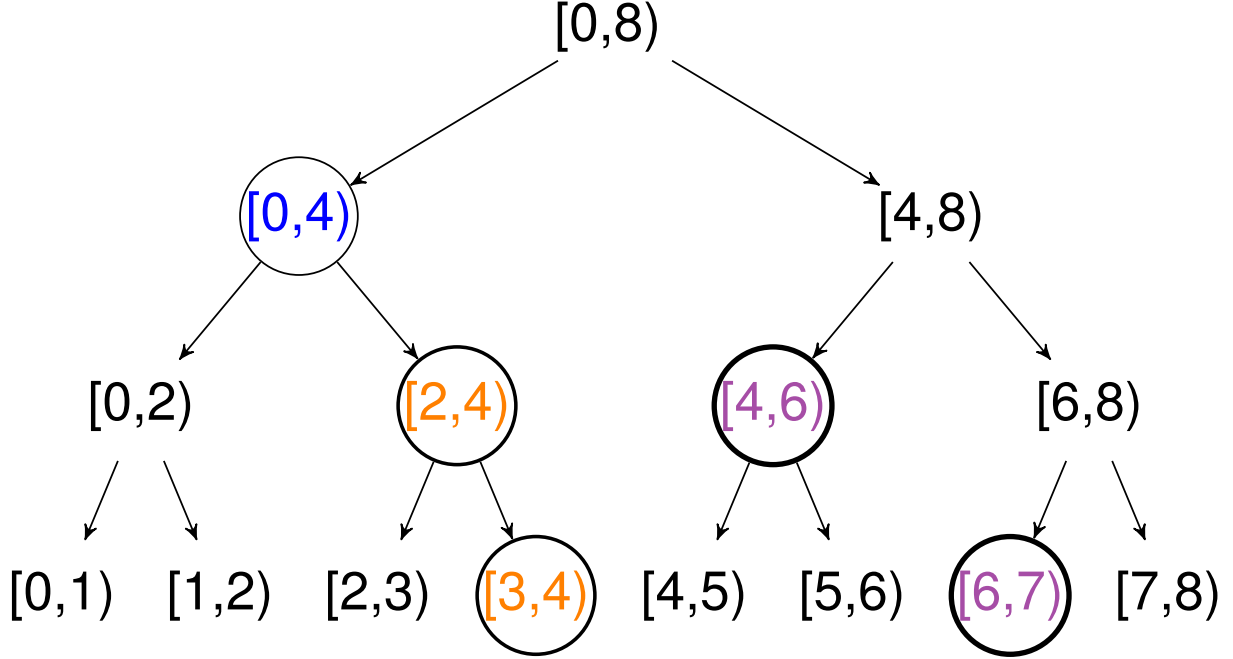


FIGURE 2.2: A global strategy $\mathbb{A}^*$ in a binary tree decomposition for an integer domain $[0, 8)$. Workload queries include $\mathbb{W}_1 = \{[0, 7)\}$, $\mathbb{W}_2 = \{[2, 6), [3, 7)\}$. Nodes present only in strategy $\mathbb{A}_1$ are shown in blue text, nodes only in $\mathbb{A}_2$ are in orange text, nodes in both $\mathbb{A}_1$ and $\mathbb{A}_2$ are shown in violet text.

Example 2.3.1. In Figure ??, for an integer domain $[0, 8)$, we show a binary tree decomposition for its global strategy $\mathbb{A}^*$. This strategy consists of $(2^3 + 2^2 + 2^1 + 1)$ row counting queries (RCQs), where each RCQ corresponds to the counting query with the predicate range indicated by a node in the tree. We use $\mathbb{A}_{[a,b)}^*$ to denote the RCQ with a range $[a, b)$ in the global strategy matrix.

The first workload $\mathbb{W}_1$ consists of a single query with a range predicate $[0, 7)$. It can be answered by summing over noisy responses to three RCQs in the global strategy matrix, $(\mathbb{A}_{[0,4)}^*, \mathbb{A}_{[4,6)}^*, \mathbb{A}_{[6,7)}^*)$. The second workload $\mathbb{W}_2$ has two queries with range predicates $([2, 6), [3, 7))$. It can be answered using $\mathbb{A}_2 = (\mathbb{A}_{[2,4)}^*, \mathbb{A}_{[4,6)}^*, \mathbb{A}_{[3,4)}^*, \mathbb{A}_{[6,7)}^*)$. We detail the strategy generation in Example ??.

We observe that the RCQs $\mathbb{A}_{[4,6)}^*$ and $\mathbb{A}_{[6,7)}^*$ are common to both $\mathbb{A}_1$ and $\mathbb{A}_2$. Thus, our cache-aware DP mechanisms can potentially reuse their noisy responses to answer $\mathbb{A}_2$.

□

The accuracy analysis of the matrix mechanism only holds over full-rank strategy matrices [LMH+15, Section 4]. However, in general, the instant strategy $\mathbb{A}$ may not be full rank. We address this challenge in the FRT module, by mapping the instant strategy $\mathbb{A}$, workload $\mathbb{W}$, data vector $\mathbb{x}$, to a compact, full-rank, efficient representation, resulting in $\mathbf{A}$, $\mathbf{W}$ and $\mathbf{x}$ respectively. Thus for an input $\mathbb{W}, \mathbb{x}$, the ST module outputs $(\mathbf{A}, \mathbf{W}, \mathbf{x})$. Since the cache entries should be uniquely addressable, the raw data vector $\mathbb{x}$ and strategy $\mathbb{A}$ are used to index the cache.

2.3.3 Cache-aware DP Modules

Our system supports two novel classes of cache-aware DP mechanisms: *Modified Matrix Mechanism* (MMM) and the *Relax Privacy Mechanism* (RP). These two cache-aware DP mechanisms commonly use the ST module to transform an input $\mathbb{W}, \mathbb{x}$ to $(\mathbf{W}, \mathbf{A}, \mathbf{x})$. Each cache-aware DP mechanism implements two interfaces (similar to APEx [GHIM19]) using the mapped representations $\mathbf{A}, \mathbf{W}, \mathbf{x}$, as well as the cache $\mathcal{C}$:

- The ANSWERWORKLOAD interface answers a workload $\mathbf{W}$. It first answers the instant strategy $\mathbf{A}$, by combining noisy responses from the cache $\mathcal{C}$, as well as deriving fresh noisy responses based on the ground truth from the DB, which requires privacy budget ϵ . Implementations of this interface differ in how they answer $\mathbf{A}$. However, they identically post-processes the strategy noisy response to form the noisy workload response, following the matrix mechanism, as in Definition ???. Each implementation also updates the cache.
- The ESTIMATEPRIVACYBUDGET interface estimates the minimum privacy budget ϵ required by the ANSWERWORKLOAD interface to achieve the (α, β) accuracy requirement.

For the first cache-aware DP mechanism, MMM, we have two optional modules, namely *Strategy Expander* (SE) and *Proactive Querying* (PQ), which modify the instant strategy $\mathbb{A}$ output by the basic ST module, for different purposes. While both modules reduce the privacy cost, they differ in terms of the workload whose privacy cost is reduced. The SE module reduces the cost for the given workload under MMM, whereas the PQ module reduces the cost of future workloads. The SE module expands $\mathbb{A}$ with related, accurate, cached strategy rows to exploit constrained inference, as discussed by Hay et al. [HRMS10]. On the other hand, the PQ module is designed to fill the cache proactively, for use by the other modules in answering future workloads. It expands $\mathbb{A}$

Algorithm 1 CacheDP Overview

Require: Dataset D , Total privacy budget $\mathcal{B}$.

- 1: Initialize privacy loss $B_c = 0$, cache $\mathcal{C} = \{(\mathbb{W}, -, -, 0) | \mathbb{W} \in \mathbb{A}^*\}$.
 - 2: **repeat**
 - 3: Receive (Q, α, β) from analyst
 - 4: $\mathbb{W}, \mathbb{A} \leftarrow \text{GENERATESTRATEGY}(Q) \triangleright (\text{Algorithm ??, Line ??})$
 - 5: $\mathbb{A}, \mathbb{T} \leftarrow \text{TRANSFORMSTRATEGY}(\mathbb{A}) \triangleright (\text{Algorithm ??, Line ??})$
 - 6: $\mathbb{W} \leftarrow \text{FRT}(\mathbb{T}, \mathbb{W}) \triangleright (\text{Algorithm ??, Line ??})$
 - 7: $(b, \epsilon_1) \leftarrow \text{MMM.ESTIMATEPRIVACYBUDGET}(\mathcal{C}, \mathbb{A}, \mathbb{W}, \alpha, \beta, \mathbb{A}, \mathbb{T}) \triangleright (\text{Algorithm ??})$
 - 8: $\epsilon_2 \leftarrow \text{RP.ESTIMATEPRIVACYBUDGET}(\mathcal{C}, \mathbb{A}, \mathbb{W}, \alpha, \beta) \triangleright (\text{Algorithm ??})$
 - 9: $\mathbb{A}_e, \mathbb{A}_e, \mathbb{T}_e \leftarrow \text{SE.GENERATEEXPANDEDSTRATEGY}(\mathbb{A}, \mathcal{C}, b) \triangleright (\text{Algorithm ??})$
 - 10: $\epsilon_3 \leftarrow \text{MMM.ESTIMATEPRIVACYBUDGET}(\mathcal{C}, \mathbb{A}_e, \mathbb{W}, \alpha, \beta, \mathbb{A}_e, \mathbb{T}_e) \triangleright (\text{Algorithm ??})$
 - 11: Pick $(\hat{M}, \hat{\mathbb{A}})$ from (MMM or RP, $\mathbb{A}$ or $\mathbb{A}_e$) that has smallest ϵ_i
 - 12: **if** $\epsilon_i + B_c \geq \mathcal{B}$ **then**
 - 13: Answering Q satisfying (α, β) will exceed $\mathcal{B}$. Reject Q .
 - 14: ∇ We use Algorithm ?? when $\hat{M}$ is MMM or SE and Algorithm ?? when $\hat{M}$ is RP.
 - 15: $z \leftarrow \hat{M}.\text{ANSWERWORKLOAD}(\mathcal{C}, \hat{\mathbb{A}}, \mathbb{W}, \epsilon_i, \mathbf{x})$
 - 16:
 - 17: **return** z to data analyst.
 - 18: $B_c \leftarrow B_c + \epsilon_i$
 - 19: **until** no more Q from the analysts or $B_c = \mathcal{B}$
-

with strategy queries that are not cached. Importantly, it does so without incurring any additional privacy budget to answer $\mathbb{A}$, over that required to answer $\mathbb{A}$ by MMM.

Putting it all together, we state the end-to-end algorithm in Algorithm ???. First, for an input workload (Q, α, β) , our system uses the ST module to formulate the workload query into a matrix $\mathbb{W}$ and to decompose the workload query to a strategy $\mathbb{A}$ (Line ??). We generate full-rank matrices for the strategy and workload, namely $\mathbb{A}$ and $\mathbb{W}$ (Line ??). We note that as such, we do not materialize the strategy and workload matrices denoted by $\mathbb{A}$ and $\mathbb{W}$; we only materialize their smaller, full-rank forms for matrix operations (Lines ??–??). We then execute the ESTIMATEPRIVACYBUDGET interface, with the input tuple $(\mathbb{W}, \mathbb{A}, \alpha, \beta)$, for the MMM, SE, and RP mechanisms (Lines ??–??). We choose the mechanism that returns the lowest privacy cost ϵ_i (Line ??). If the sum of this privacy cost and the consumed privacy budget is smaller than the total privacy budget, then the system proceeds to the next step; otherwise, it does not answer this query, as

expected for an interactive DP system. It then executes the ANSWERWORKLOAD interface for the optimal mechanism, with the input tuple $(\mathbf{W}, \hat{\mathbf{A}}, \epsilon_i)$ (Line ??). The consumed privacy budget will increase by ϵ_i (Line ??). We note the following caveats about our optional modules. The SE module uses the same implementations of both interfaces as the MMM module, with the expanded strategy $\hat{\mathbf{A}}$ as input, instead of the original strategy. Since the PQ module has the same privacy cost as that of MMM, we do not need to rerun the ESTIMATEPRIVACYBUDGET interface for MMM, with the expanded strategy. The PQ module is run if and only if either MMM or SE are the optimal mechanisms; it is not run when RP is optimal, as we describe later. (Thus, considering Figure ??, the MMM, SE and RP modules implement the ESTIMATEPRIVACYBUDGET and ANSWERWORKLOAD interfaces, and the PQ module is run, as part of the implementation of the ANSWERWORKLOAD interface, when either the MMM or SE modules are optimal.)

We present the MMM in Section ??, the common ST module and the optional modules (SE, PQ) in Section ??, and the RP mechanism in Section ??.

Theorem 2.3.1. The workload responses produced by CacheDP, as defined in Algorithm ??, satisfy $\mathcal{B}$ -DP.

Proof. We begin by addressing the cost estimation phase (Lines ??–??). The cost estimation phase is independent of the ground truth for each DP mechanism (MMM or RP), and does not consume any privacy budget. In addition, each DP mechanism with its corresponding chosen strategy ensures ϵ_i -DP (Proposition ?? for MMM; [KHP16, Theorem 1A] for RP). The DP sequential composition theorem [DR14] states that the total privacy budget spent when answering a given workload with privacy budget ϵ_i , given that B_c has been consumed in the past, is given by the sum of the two privacy budgets. We only provide a workload response when this sum is less than the total privacy budget $\mathcal{B}$ (Line ??). Thus, by the DP sequential composition theorem, the total privacy budget spent will be less than $\mathcal{B}$ at all times,¹ and this ensures that our Algorithm ?? satisfies $\mathcal{B}$ -DP. $\square$

2.4 Modified Matrix Mechanism (MMM)

We describe our core cache-aware DP mechanism, namely the Modified Matrix Mechanism. We wish to answer a workload $\mathbf{W}$ with an (α, β) -accuracy requirement using a

¹We note that queries that are rejected consume zero privacy budget; our cost estimation phase does not depend on the ground truth data for any of our modules. Thus, this statement also holds when considering rejected queries.

cache $\mathcal{C}$ and an instant strategy $\mathbf{A} \subseteq \mathbf{A}^*$, while minimizing the privacy budget ϵ . First, we intuit the mechanism design. Second, we describe the ANSWERWORKLOAD interface, which answers $\mathbf{W}$ through $\mathbf{A}$, under optimal parameters. Third, we describe the ESTIMATEPRIVACYBUDGET interface, which derives an optimal ϵ and other parameters for the former interface.

2.4.1 MMM Overview

The cacheless matrix mechanism (Definition ??) perturbs the ground truth response to the strategy, that is $\mathbf{A}\mathbf{x}$, with the noise vector freshly drawn from $Lap(b)^{|\mathbf{A}|}$ to obtain $\tilde{\mathbf{y}} = \mathbf{A}\mathbf{x} + Lap(b)^{|\mathbf{A}|}$. An input workload is then answered using $\mathbf{W}\mathbf{A}^+\tilde{\mathbf{y}}$. As we discussed in Section ??, in an accuracy-aware DP system such as APEX [GHIM19], the noise parameter b is calibrated, first through a loose bound b_L (Theorem ??) and then to a tighter noise parameter $b^* \geq b_L$, such that the aforementioned workload response meets the (α, β) -accuracy requirement. Ultimately, the cacheless matrix mechanism would expend privacy budget given by $\frac{\|\mathbf{A}\|_1}{b^*}$ (Proposition ??).

In MMM, we seek to reduce the privacy budget spent by using the cache $\mathcal{C}$. Given an instant strategy matrix $\mathbf{A} \subseteq \mathbf{A}^*$, we first look up the cache for all strategy range queries in $\mathbf{A}$. Note that not all rows in $\mathbf{A}$ may have their noisy responses in the cache. The cache may contain noisy responses for some rows of $\mathbf{A}$, given by $\mathcal{C} \cap \mathbf{A}$, whereas other rows in $\mathbf{A}$ may not have cached responses. A preliminary approach would be to simply reuse all cached strategy responses, and obtain noisy responses for non-cached strategy rows by expending some privacy budget through naive MM. However, some cached responses may be too noisy and thus including them will lead to a higher privacy cost than the cacheless MM.

Our key insight is that by reusing noisy responses for *accurately cached* strategy rows, MMM can ultimately use a smaller privacy budget for all other strategy rows as compared to the cacheless MM, while satisfying the accuracy requirements. Thus, out of all cached strategy rows $\mathcal{C} \cap \mathbf{A}$, MMM identifies a subset of accurately cached strategy rows $\mathbb{F} \subseteq \mathcal{C} \cap \mathbf{A}$ that can be directly answered using their cached noisy responses, without spending any privacy budget; we refer to $\mathbb{F}$ as the *free strategy matrix*. MMM only spends privacy budget on the remaining strategy rows, which forms the *paid strategy matrix* $\mathbb{P}$. The full-rank forms of the free and paid matrices are denoted by $\mathbf{F}$ and $\mathbf{P}$ respectively. The ESTIMATEPRIVACYBUDGET interface of MMM seeks the best pair of free and paid strategy matrices $(\mathbf{F}, \mathbf{P})$ that use the smallest privacy budget ϵ to achieve the (α, β) -accuracy requirement. The ANSWERWORKLOAD interface uses this parameter configuration $(\mathbf{F}, \mathbf{P}, \epsilon)$ to generate noisy responses to the workload.

Algorithm 2 Modified Matrix Mechanism (MMM): ANSWERWORKLOAD interface

```

1: function ANSWERWORKLOAD( $\mathcal{C}, \mathbf{A}, \mathbf{W}, \epsilon, \mathbf{x}$ )
2:   Parameters: Workload sequence number ( $i$ )
3:    $(\mathbf{F}, \mathbf{P}, b_{\mathbf{P}}, \epsilon) \leftarrow \text{ESTIMATEPRIVACYBUDGET}(\mathcal{C}, \mathbf{A}, \mathbf{W}, \alpha, \beta)$   $\triangleright$  Defined in Algorithm ??.
4:   if isPQModuleOn then  $\triangleright$  Expand  $\mathbb{P}$  using the PQ module (Section ??)
5:      $\Delta \mathbf{P} \leftarrow \text{GENERATEPROACTIVESTRATEGY}(\mathbb{P}, \mathcal{C})$   $\triangleright$  Defined in Algorithm ??.
6:      $\mathbf{P} \leftarrow \mathbf{P} \cup \Delta \mathbf{P}$ 
7:      $\tilde{\mathbf{y}}_{\mathbf{P}} \leftarrow \mathbf{P}\mathbf{x} + \text{Lap}(b_{\mathbf{P}})^{|\mathbf{P}|}$   $\triangleright$  Generate noisy responses to  $\mathbf{P}$ .
8:      $\mathcal{C} \leftarrow \mathcal{C} \cup (\mathbf{P}, b_{\mathbf{P}}, \tilde{\mathbf{y}}_{\mathbf{P}}, t = i)$   $\triangleright$  Update the cache with  $\tilde{\mathbf{y}}_{\mathbf{P}}$ .
9:      $\tilde{\mathbf{y}}_{\mathbf{F}} \leftarrow [c.\tilde{\mathbf{y}} \mid c \in \mathcal{C}, c.\mathbf{a} \in \mathbf{F}]$   $\triangleright$  Get noisy responses for  $\mathbf{F}$  from the cache.
10:     $\tilde{\mathbf{y}} \leftarrow \tilde{\mathbf{y}}_{\mathbf{F}} \cup \tilde{\mathbf{y}}_{\mathbf{P}}$   $\triangleright$  Combine free and paid noisy responses, according to their order in  $\mathbf{A}$ .
11:     $\tilde{\mathbf{z}} \leftarrow \mathbf{W}\mathbf{A}^+ \tilde{\mathbf{y}}$   $\triangleright$  Post-process noisy strategy response to answer  $\mathbf{W}$ .
12:  return  $\tilde{\mathbf{z}}$ 

```

2.4.2 Answer Workload Interface

We present an implementation of the ANSWERWORKLOAD interface for the MMM (Algorithm ??). We recall that CacheDP first calls the ESTIMATEPRIVACYBUDGET implementation for each module, and supplies its output to the ANSWERWORKLOAD implementation. For MMM, the latter implementation receives the following: the free and paid strategy matrices, the optimal noise parameter for the paid strategy matrix, and its associated privacy cost (Line ??). We observe that the ESTIMATEPRIVACYBUDGET implementation ensures that $b_{\mathbf{P}} = \frac{\|\mathbf{P}\|_1}{\epsilon}$, to satisfy ϵ -DP.

First, ANSWERWORKLOAD checks if the PQ module (Section ??) is turned on (Line ??). If it is, then the paid strategy matrix will be expanded for the remaining operations (Line ??–Line ??). Second, $\mathbf{P}$ is answered using the Laplace mechanism with the noise parameter $b_{\mathbf{P}}$ (Line ??). Then these fresh noisy responses will be inserted into the cache (Line ??). In particular, for each query $\mathbf{w} \in \mathbf{P}$, we update its corresponding noisy parameter, noisy response, and timestamp in $\mathcal{C}$ to $b_{\mathbf{P}}, \tilde{\mathbf{y}}$, and the sequence number of the current workload respectively. Third, this interface retrieves the responses $\tilde{\mathbf{y}}_{\mathbf{F}}$ for the free strategy matrix from the cache (Line ??). It then combines the free and paid noisy responses into $\tilde{\mathbf{y}}$, according to their order in the instant strategy $\mathbf{A}$ (Line ??). Finally, this interface derives the noisy workload response by post-processing the noisy response to the strategy, just as in the Matrix Mechanism (Line ??).

Proposition 2.4.1. The ANSWERWORKLOAD interface of MMM (Algorithm ??) satisfies ϵ -DP, where ϵ is the budget output by the ESTIMATEPRIVACYBUDGET interface.

Proof. When the cache is empty, the privacy of MMM follows from that of the matrix

mechanism, as defined in Proposition ?? . When there are entries in the cache, we split the strategy matrix into the free and paid matrices. The paid matrix satisfies ϵ -DP by the same reasoning as above. The privacy of the free matrix responses and the final concatenation of free and paid responses follow by the post-processing lemma of DP [DR14, Proposition 2.1]. $\square$

Error term for MMM. As the noisy response vector $\tilde{\mathbf{y}}$ to the strategy $\mathbf{A}$ is a combination of $\tilde{\mathbf{y}}_{\mathbf{F}}$ and $\tilde{\mathbf{y}}_{\mathbf{P}}$, its distribution is equivalent to a ground truth vector perturbed by a vector of Laplace noise with parameters: $\mathbf{b} = \mathbf{b}_{\mathbf{F}} \parallel \mathbf{b}_{\mathbf{P}}$, where $\mathbf{b}_{\mathbf{F}}$ is a vector of noise parameters for the cached entries in $\mathbf{F}$ with length $|\mathbf{F}|$ and $\mathbf{b}_{\mathbf{P}}$ is a vector of the constant $b_{\mathbf{P}}$, with length $|\mathbf{P}|$. This differs from the standard matrix mechanism with a single scalar noise parameter. We derive the error term for the noisy workload response $\tilde{\mathbf{z}}$ next.

Proposition 2.4.2. The noisy workload response $\tilde{\mathbf{z}}$ output by the ANSWERWORKLOAD interface of MMM is bounded by the following (α, β) -worst case error:

$$\Pr[\|\mathbf{W}\mathbf{x} - \mathbf{W}\mathbf{A}^+M(\mathbf{W}, \mathbf{x})\|_{\infty} \geq \alpha] \leq \frac{2\|\mathbf{W}\mathbf{A}^+diag(\mathbf{b})\|_F^2}{\alpha^2}$$

and the following total square expected error term:

$$\mathbb{E}[\|\mathbf{W}\mathbf{x} - \mathbf{W}\mathbf{A}^+M(\mathbf{W}, \mathbf{x})\|_2^2] = 2\|\mathbf{W}\mathbf{A}^+diag(\mathbf{b})\|_F^2$$

Proof. The error term for MMM is:

$$\boldsymbol{\eta} = \mathbf{W}\mathbf{A}^+(\mathbf{A}\mathbf{x} + Lap(\mathbf{b})) - \mathbf{W}\mathbf{x} = \mathbf{W}\mathbf{A}^+Lap(\mathbf{b})$$

The i -th element of this error vector is thus given by:

$$\eta[i] = \sum_{j=1}^k (\mathbf{W}\mathbf{A}^+)[i, j] Lap(\mathbf{b}[j])$$

Thus, the variance of the i -th element of this error vector is as follows:

$$Var[\boldsymbol{\eta}[i]] = Var\left[\sum_{j=1}^k (\mathbf{WA}^+)[i, j] Lap(\mathbf{b}[j])\right] \quad (2.6)$$

$$= \sum_{j=1}^k Var[(\mathbf{WA}^+)[i, j] Lap(\mathbf{b}[j])] \quad (Var[X + Y] = Var[X] + Var[Y]) \quad (2.7)$$

$$= \sum_{j=1}^k (\mathbf{WA}^+)[i, j]^2 \cdot Var[Lap(\mathbf{b}[j])] \quad (Var[kX] = k^2 Var[X]) \quad (2.8)$$

$$= 2 \sum_{j=1}^k (\mathbf{WA}^+)[i, j]^2 \cdot Lap(\mathbf{b}[j])^2 \quad (Var[Lap(\mathbf{b}[j])] = 2\mathbf{b}[j]^2) \quad (2.9)$$

We are interested in the L_∞ norm of the error vector for the (α, β) -bound (Definition ??). The probability that the i -th element is greater than α can be bounded using Chebyshev's inequality:

$$\begin{aligned} Pr[\boldsymbol{\eta}[i] \geq \alpha] &= Pr\left[\sum_{j=1}^k (\mathbf{WA}^+)[i, j] Lap(\mathbf{b}[j]) \geq \alpha\right] \leq \frac{Var[\boldsymbol{\eta}[i]]}{\alpha^2} \\ &= \frac{2 \sum_{j=1}^k (\mathbf{WA}^+)[i, j]^2 \cdot \mathbf{b}[j]^2}{\alpha^2} \end{aligned}$$

By the union bound, the probability that at least one element of this error vector exceeds α is bounded by the sum of the probability for any element to exceed α :

$$\begin{aligned} Pr[\|\mathbf{WA}^+ Lap(\mathbf{b})\|_\infty \geq \alpha] &\leq \sum_{i=1}^\ell \frac{2 \sum_{j=1}^k (\mathbf{WA}^+)[i, j]^2 \cdot \mathbf{b}[j]^2}{\alpha^2} \\ &= \frac{2 \sum_{i=1}^\ell \sum_{j=1}^k (\mathbf{WA}^+)[i, j]^2 \cdot \mathbf{b}[j]^2}{\alpha^2} \\ &= \frac{2 \|\mathbf{WA}^+ diag(\mathbf{b})\|_F^2}{\alpha^2} \end{aligned}$$

The expected total square error can be expanded as follows:

$$\mathbb{E}[\|\boldsymbol{\eta}\|_2^2] = \mathbb{E}[\|\mathbf{W}\mathbf{A}^+ \text{Lap}(\mathbf{b})\|_2^2] = \sum_{i=1}^l \mathbb{E}\left[\sum_{j=1}^k ((\mathbf{W}\mathbf{A}^+)[i, j] \text{Lap}(\mathbf{b}[j]))^2\right]$$

As the k noise variables are independent and have a zero mean, we have that $\mathbb{E}[X^2] = \text{Var}[X]$ where $X = \boldsymbol{\eta}[i]$. Computing $\text{Var}[\boldsymbol{\eta}[i]]$ as through Equations (??)–(??), we can obtain the following value for the expected total square error:

$$\mathbb{E}[\|\boldsymbol{\eta}\|_2^2] = 2 \sum_{i=1}^l \sum_{j=1}^k (\mathbf{W}\mathbf{A}^+)[i, j]^2 \mathbf{b}[j]^2,$$

which is equivalent to $2\|\mathbf{W}\mathbf{A}^+ \text{diag}(\mathbf{b})\|_F^2$. (Thus, when computing the expected total square error within examples throughout this chapter, we minimize this Frobenius norm and ignore the constant factor.) $\square$

2.4.3 Estimate Privacy Budget Interface

The ESTIMATEPRIVACYBUDGET interface chooses the free and paid strategy matrices and determines the privacy budget required for the ANSWERWORKLOAD interface. An implementation of the former interface answers the following questions:

- (1) Which cached strategy rows out of $\mathcal{C} \cap \mathbb{A}$ should be included in $\mathbb{F}$? The choice of $\mathbb{F}$ directly determines the paid strategy matrix $\mathbb{P}$ as $\mathbb{A} - \mathbb{F}$, and $\mathbb{F}$ also determines $\tilde{\mathbf{y}}_{\mathbb{F}}$, which is a vector of noise parameters for the cached strategy entries. We derive the full-rank versions of $\mathbb{F}$ and $\mathbb{P}$, namely $\mathbf{F}$ and $\mathbf{P}$, through the FRT function (Algorithm ??), as we detail in our implementation of this interface.
- (2) Given $\mathbf{P}$ and $b_{\mathbf{P}}$, the privacy budget paid by MMM is given by $\epsilon = \|\mathbf{P}\|_1 / b_{\mathbf{P}}$. To minimize this privacy budget, what is the maximum noise parameter value $b_{\mathbf{P}}$ that can be used to answer $\mathbf{P}$ while meeting the accuracy requirement?

A baseline approach to the first question is to simply set $\mathbb{F} = \mathcal{C} \cap \mathbb{A}$, that is, we reuse all cached strategy responses. This approach may reuse inaccurate cached responses with large noise parameters, which results in a larger ϵ (or a smaller $b_{\mathbf{P}}$) to achieve the given accuracy requirement than answering the entire $\mathbb{A}$ by resampling new noisy responses without using the cache.

$$\mathbf{W}_1 = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}, \mathbf{A}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{b}_1 = \begin{bmatrix} b_c \\ b \\ b \end{bmatrix}, \mathbf{x}_1 = \begin{bmatrix} \mathbb{x}[0, 4) \\ \mathbb{x}[4, 6) \\ \mathbb{x}[6, 7) \end{bmatrix}$$

FIGURE 2.3: Consider $\mathbb{W}_1 = \{[0, 7)\}$ with its corresponding mapped workload matrix, instant strategy, noise vector, and data vector. Reusing a cached response for the first row with noise parameter b_c requires a smaller noise parameter b (and hence a bigger privacy budget) for the other rows than the cacheless MM to achieve the same accuracy level.

Example 2.4.1. Continuing with Example ??, we consider the first workload $\mathbb{W}_1 = \{[0, 7)\}$ and its strategy $\mathbb{A}_1 = \{\mathbb{A}_{[0,4)}^*, \mathbb{A}_{[4,6)}^*, \mathbb{A}_{[6,7)}^*\}$. We present the full-rank workload and strategy matrices, namely $\mathbf{W}_1$ and $\mathbf{A}_1$, in Figure ??; these matrices operate with a data vector for the following ranges: $\{[0, 4), [4, 6), [6, 7)\}$. For simplicity, we use the expected square error to illustrate the drawback of the baseline approach, but the same reasoning applies to (α, β) -worst error bound. Without using the cache, when we set $\mathbf{b} = [10, 10, 10]$, we achieve an expected error $\|\mathbf{W}_1 \mathbf{A}_1^+ \text{diag}(\mathbf{b})\|_F^2 = 300$ for the workload $\mathbf{W}_1$. Suppose the cache has an entry for the first RCQ $[0, 4)$ of the strategy at a noise parameter $b_c = 15$. Using this cached entry, the noise vector becomes $\mathbf{b} = [15, b_P, b_P]$, and the expected square error is $\|\mathbf{W}_1 \mathbf{A}_1^+ \text{diag}(\mathbf{b})\|_F^2 = 15^2 + 2b_P^2$. To achieve the same or a smaller error than the cacheless MM, we need to set $b_P \leq \sqrt{(300 - 15^2)/2} \approx 6.12$ for the remaining entries in the strategy. This tighter noise parameter b_P corresponds to a larger privacy budget. $\square$

Privacy Cost Optimizer. We formalize the two aforementioned questions as an optimization problem, subject to the accuracy requirement, as follows.

Cost estimation (CE) problem: Given a cache $\mathcal{C}$, an instant strategy matrix $\mathbb{A}$, as well as the full-rank matrices $\mathbf{A}$ and $\mathbf{W}$, determine $\mathbb{F} \subseteq (\mathbb{A} \cap \mathcal{C})$ and $b_P \in [b_L, b_T]$ that:

- i minimize the paid privacy budget $\epsilon = \frac{\|\mathbf{P}\|_1}{b_P} = \frac{\|\text{FRT}(\mathbb{A}-\mathbb{F})\|_1}{b_P}$, and
- ii satisfy one of the following accuracy requirements: $\|\mathbf{W} \mathbf{A}^+ \text{diag}(\mathbf{b}_F \| \mathbf{b}_P)\|_F^2 \leq \alpha^2$ or $\Pr[\|\mathbf{W} \mathbf{A}^+ \text{Lap}(\mathbf{b}_F \| \mathbf{b}_P)\|_\infty \geq \alpha] \leq \beta$.

Here, b_L is as defined in Definition ?? and b_T can be made as large as desired. The choice of $\mathbb{F}$ directly determines $\mathbf{b}_F$, which is a vector of noise parameters for these cached entries.

In this optimization problem, the lower bound for b_P , namely b_L , is the loose bound for the cacheless MM (Equation (??)), and the upper bound b_T can be made as large as desired. In a brute-force solution to this problem, we can search over all possible pairs of $\mathbb{F} \subseteq (\mathbb{A} \cap \mathcal{C})$ and $b_P \in [b_L, b_T]$, and check whether each pair of $(\mathbb{F}, b_P)$ can lead to an accurate response. In this solution, the search space for $\mathbb{F}$ will be $O(2^{|\mathbb{A} \cap \mathcal{C}|})$. For each value of $\mathbb{F}$, we can conduct a binary search for b_P within $[b_L, b_T]$, with a resolution parameter of b_δ . This leads to a search space of $\log_2((b_T - b_L)/b_\delta)$ for b_P . Thus the total search space will be $O(2^{|\mathbb{A} \cap \mathcal{C}|} \cdot \log_2((b_T - b_L)/b_\delta))$. Hence, we need another way to efficiently determine optimal values for $(\mathbb{F}, b_P)$.

Simplified Privacy Cost Optimizer. We simplify the CE problem to arrive at a much smaller search space for $(\mathbb{F}, b_P)$, while ensuring that b_P improves over the noise parameter of the cacheless MM. We observe that if we only reuse cached entries that are at least as accurate as the paid entries, that is, the entries in $\mathbb{F}$ have noise parameters smaller than b_P , then we will have a smaller error than a cacheless MM with a noise parameter $b = b_P$. This motivates us to consider the following search space for $\mathbb{F}$. When given b_P , we choose a free strategy matrix fully determined by this noise parameter:

$$\mathbb{F}_{b_P} = [c.a \mid c \in \mathcal{C} \cap \mathbb{A}, c.b \leq b_P], \quad (2.10)$$

and $\mathbf{F}$ is simply the full-rank form of this matrix. We then formalize a simplified optimization problem.

Simplified CE problem: Given a cache $\mathcal{C}$, an instant strategy matrix $\mathbb{A}$, as well as the full-rank matrices $\mathbf{A}$ and $\mathbf{W}$, determine $b_P \in [b_L, b_T]$ that:

- i minimizes the paid privacy budget $\epsilon = \frac{\|\mathbf{P}\|_1}{b_P} = \frac{\|\text{FRT}(\mathbb{A}-\mathbb{F})\|_1}{b_P}$, where $\mathbb{F} = \mathbb{F}_{b_P}$, as defined in Equation ??, and
- ii satisfies one of the following accuracy requirements: $\|\mathbf{W}\mathbf{A}^+ \text{diag}(\mathbf{b}_F \| \mathbf{b}_P)\|_F^2 \leq \alpha^2$ or $\Pr[\|\mathbf{W}\mathbf{A}^+ \text{Lap}(\mathbf{b}_F \| \mathbf{b}_P)\|_\infty \geq \alpha] \leq \beta$.

Here, b_L is as defined in Definition ?? and b_T can be made as large as desired. The choice of $\mathbb{F}$ directly determines $\mathbf{b}_F$, which is a vector of noise parameters for these cached entries.

To solve the simplified CE problem, we only need to conduct the binary search for b_P once, rather than for each value of $\mathbb{F}$ as in the CE problem. Thus, this leads to a much smaller search space of $O(\log_2((b_T - b_L)/b_\delta))$. Hence, this approach significantly improves over the brute-force search solution for the CE problem.

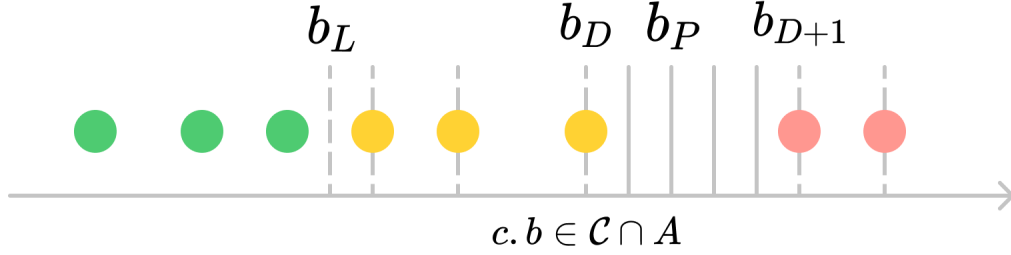


FIGURE 2.4: An example of cached noise parameters $c.b \in \mathcal{A} \cap \mathcal{C}$ (dots) and the discrete (dashed lines) and continuous (full lines) binary searches through these parameters. Parameters in green ($c.b < b_L$) always meet the accuracy requirements and are always reused. The discrete search scans over $c.b \geq b_L$ and outputs b_D ; the $\mathbf{F}$ includes all cached entries in green and yellow. Cache entries in red are thus too inaccurate to be reused; they are placed into $\mathbf{P}$ and re-sampled. The continuous search scans over the interval $[b_D, b_{D+1}]$ and identifies an optimal b_P .

Theorem 2.4.1. The optimal solution to the simplified CE problem has the same or smaller privacy cost ϵ than the privacy cost $\epsilon_{\mathbf{F}=\emptyset}$ of the matrix mechanism without cache, i.e., MMM with $\mathbf{F} = \emptyset$.

Proof. Let b^* be any noise parameter that achieves the desired accuracy requirement for $\mathbf{A}$ in the cacheless MM. We observe that b^* is a valid solution to the simplified CE problem, since setting $\mathbf{b}_{\mathbf{F}} = [c.b \in \mathcal{C} \mid c.a \in \mathcal{C} \cap \mathcal{A}, c.b \leq b^*]$ and $\mathbf{b}_{\mathbf{P}} = [b^* \mid a \in \mathcal{P}]$ satisfies the accuracy requirement. Since $\|\mathbf{P}\|_1 \leq \|\mathbf{A}\|_1$, the privacy cost for $\mathbf{b}_{\mathbf{P}} = b^*$ when using the cache, namely $\epsilon = \frac{\|\mathbf{P}\|_1}{b^*}$, is smaller than the privacy cost for the cacheless MM, namely $\epsilon_{\mathbf{F}=\emptyset} = \frac{\|\mathbf{A}\|_1}{b^*}$. Thus, the optimal solution to the simplified CE problem has the same or smaller privacy cost than the cacheless matrix mechanism at $b_{\mathbf{P}} = b^*$. $\square$

We solve the simplified CE problem, through our implementation of the ESTIMATEPRIVACYBUDGET interface for MMM, which is presented in Algorithm ???. At a high level, we conduct a discrete and a continuous binary search through the cached noise parameters, as visualized in Figure ???. Our discrete search has two goals, as follows. First, it splits the strategy matrix into optimal free and paid strategy matrices. Second, it arrives at a sub-optimal paid noise parameter, b_D , from among the noise parameters of accurate cached entries. Whereas, our continuous search looks for a slightly more optimal b_P between two consecutive cached noise parameters ($[b_D, b_{D+1}]$).

Step 1: Discrete search for b_P . In this step, we search for b_P from the noise parameters of the cached strategy rows $\mathcal{A} \cap \mathcal{C}$. We denote the list of all cached noise parameters,

Algorithm 3 MMM: ESTIMATEPRIVACYBUDGET interface and helper function

```

1: function ESTIMATEPRIVACYBUDGET( $\mathcal{C}, \mathbf{A}, \mathbf{W}, \alpha, \beta, \mathbb{A}, \mathbb{T}$ )
2:    $b_L \leftarrow \frac{\alpha\sqrt{\beta/2}}{\|\mathbf{W}\mathbf{A}^+\|_F}$   $\triangleright$  Set lower bound for discrete search based on Theorem ??
3:    $\mathbf{b} \leftarrow \text{sort}([c.b \mid c \in \mathbb{A} \cap \mathcal{C}, c.b > b_L] \cup [b_L])$   $\triangleright$  Set search space for discrete search.
4:    $b_D \leftarrow \text{BINARYSEARCH}(\text{CHECKACCURACY}(\cdot, \mathcal{C}, \mathbf{A}, \mathbf{W}, \alpha, \beta), \mathbf{b})$   $\triangleright$  Discrete search.
5:    $\mathbb{F} \leftarrow [c.a \mid c \in \mathbb{A} \cap \mathcal{C}, c.b \leq b_D]$   $\triangleright$  Set  $\mathbb{F}$  given  $b_D$ .
6:    $\mathbb{P} \leftarrow \mathbb{A} - \mathbb{F}$   $\triangleright$  Set  $\mathbb{P}$  given  $\mathbb{F}$ .
7:    $\mathbf{F} \leftarrow \text{FRT}(\mathbb{F}, \mathbb{T})$   $\triangleright$  Transform  $\mathbb{F}$  into full-rank  $\mathbf{F}$ .
8:    $\mathbf{P} \leftarrow \text{FRT}(\mathbb{P}, \mathbb{T})$   $\triangleright$  Transform  $\mathbb{P}$  into full-rank  $\mathbf{P}$ .
9:    $\triangleright$  Set an upper bound for the continuous search to the smaller of  $b_{\mathbb{T}}$  and the next largest
   cached noise parameter,  $b_{D+1}$  if it exists.
10:   $b_{\mathbb{T}} \leftarrow \min(b_{D+1}, b_{\mathbb{T}})$ 
11:   $\triangleright$  Continuous search using a noise parameter resolution of  $b_{\delta}$ .
12:   $b_{\mathbf{P}} \leftarrow \text{BINARYSEARCH}(\text{CHECKACCURACY}(\cdot, \mathcal{C}, \mathbf{A}, \mathbf{W}, \alpha, \beta, \mathbb{A}), [b_D, b_{\mathbb{T}}], b_{\delta})$ 
13:   $\epsilon \leftarrow \frac{\|\mathbf{P}\|_1}{b_{\mathbf{P}}}$   $\triangleright$  Derive the budget to spend on answering  $\mathbf{P}$ .
14:  return ( $\mathbf{F}, \mathbf{P}, b_{\mathbf{P}}, \epsilon$ )

15: function CHECKACCURACY( $b', \mathcal{C}, \mathbf{A}, \mathbf{W}, \alpha, \beta, \mathbb{A}$ )  $\triangleright b'$  is a candidate for  $b_{\mathbf{P}}$ .
16:   Parameters: Sample size ( $N = 10000$ ), failure counter ( $n_f = 0$ ).
17:    $\mathbf{b}_{\mathbf{F}} \leftarrow [c.b \mid c \in \mathbb{A} \cap \mathcal{C}, c.b \leq b']$   $\triangleright$  Set  $\mathbf{b}_{\mathbf{F}}$  given  $b'$ .
18:    $x \leftarrow |\mathbf{A}| - |\mathbf{b}_{\mathbf{F}}|$   $\triangleright x$  represents the length of the paid noise parameter vector  $\mathbf{b}_{\mathbf{P}}$ .
19:    $\mathbf{b}_{\mathbf{P}} \leftarrow [b']^x$   $\triangleright$  Set  $\mathbf{b}_{\mathbf{P}}$  to a vector of length  $x$ , containing the constant  $b'$ .
20:   for  $i = 1, \dots, N$  do
21:      $\tilde{\mathbf{y}} \leftarrow \text{Lap}(\mathbf{b}_{\mathbf{F}} \|\mathbf{b}_{\mathbf{P}})$   $\triangleright$  Sample a noise vector from the Laplace distribution, given  $\mathbf{b}_{\mathbf{F}}$ 
    and  $\mathbf{b}_{\mathbf{P}}$ .
22:     if  $\|\mathbf{W}\mathbf{A}^+ \tilde{\mathbf{y}}\|_{\infty} > \alpha$  then  $\triangleright$  Check if the noise vector fails the accuracy requirement.
23:        $n_f \leftarrow n_f + 1$   $\triangleright$  Increment the failure counter.
24:    $\beta_e = n_f / N$   $\triangleright$  Empirical failure rate for  $N$  runs.
25:    $p = \beta / 100$   $\triangleright$  Confidence interval of  $1 - p$ .
26:    $\beta' = z_{1-p/2} \sqrt{\beta_e(1 - \beta_e) / N}$ 
27:   if  $(\beta_e + \beta' + p/2) < \beta$  then
28:     return True
29:   else
30:     return False

```

sorted in increasing order, by $\mathbf{b}$ (Line ??). All cached strategy rows whose noise parameters are less than or equal to the loose bound for cacheless MM always meet the accuracy requirement (Theorem ??). Consequently, the lower bound for the search is set to b_L (Line ??). We conduct a binary search to find the largest possible $b_D \in \mathbf{b}$ that meets the accuracy requirement (Line ??). As described in the simplified CE problem, the free strategy matrix $\mathbb{F}$ is fully determined using b_D (Line ??). The paid strategy matrix $\mathbb{P}$ is then computed using $\mathbb{P}$ (Line ??). The full-rank forms of both matrices are computed through the FRT function (Lines ??–??).

Step 2: Refining b_P in a continuous space. We observe that we may further increase b_P , by examining the interval between b_D , which is output from the discrete search, and the next largest cached noise parameter, denoted by b_{D+1} . If b_{D+1} does not exist, then we set the upper bound to a predefined, large b_T (Line ??). We conduct a binary search in the continuous domain $[b_D, b_T]$ (Line ??), with b_δ controlling the resolution of this search. This continuous search does not impact the split of the strategy matrix $\mathbf{A}$ into $\mathbf{F}$ and $\mathbf{P}$, which was done during the discrete search, since the chosen noise parameter will be strictly smaller than b_{D+1} . The continuous search is depicted through full lines in Figure ?. This search outputs a noise parameter b_P . Finally, this function computes the privacy budget for $\mathbf{P}$, as $\epsilon = \frac{\|\mathbf{P}\|_1}{b_P}$ (Line ??). The function outputs the free and paid strategy matrices derived from the discrete search, as well as the optimal noise parameter and privacy budget for the paid strategy.

CHECKACCURACY function. During both discrete and continuous searches, to check if a candidate b' achieves the accuracy requirement, we run the CHECKACCURACY function, which is defined in Algorithm ?. We initialize the sample size N and the failure counter n_f for the Monte Carlo simulation (Line ??). This function first fetches the cached noise parameter vector $\tilde{\mathbf{y}}_F$ given the candidate paid noise parameter b' (Line ??). It constructs the paid noise parameter vector of the appropriate length, consisting of the constant b' (Line ??). Then it runs N iterations of the Monte Carlo simulation (Line ??). In each iteration, it samples a noise vector from the Laplace distribution, given $\mathbf{b}_F$ and $\mathbf{b}_P$ (Line ??). We observe that our MC simulation differs from Ge et al.'s [GHIM19] in this step. Their simulation uses only a single noise parameter for all entries of the strategy, and samples noise from the same distribution for each entry. Our function then evaluates whether the noise vector *fails* the accuracy requirement, as described in Proposition ?? (Line ??). If the requirement is violated, then the function increments the failure counter (Line ??). At the end of N iterations, the function uses N and n_f to compute the empirical failure probability and evaluate whether it exceeds β (Lines ??–??).

Convergence of searches. We argue that both discrete and continuous searches will converge, as when the candidate b' for b_P increases, the resulting privacy budget ϵ , as defined in Line ??, decreases monotonically. When b' increases, one or more cached strategy rows may be added to $\mathbb{F}$, as evident from Line ?. In other words, one or more rows that would have previously been within $\mathbb{P}$, and thus paid for, may become free. Thus, the sensitivity of the paid strategy matrix $\mathbb{P}$, or equivalently, that of $\mathbb{P}$, may decrease. Since the privacy cost is directly proportional to this sensitivity, the privacy cost would decrease monotonically as the sensitivity decreases and b' increases.

Computational complexity. We observe that there are two main computationally intensive operations within our ESTIMATEPRIVACYBUDGET interface. First, the Monte Carlo simulation within the CHECKACCURACY function is a computationally intensive operation. However, each run of the simulation can be easily parallelized. Second, computing $\mathbf{WA}^+$ becomes the bottleneck as the attribute domain size and the number of the workload RCQs increases, since they respectively correspond to the number of columns and rows in this product. We note that we can reuse this product across the discrete and continuous searches.

2.5 Strategy Modules

In this section, we first present the Strategy Transformer (ST), which is used by all of our cache-aware DP mechanisms. We then present two optional modules: the Strategy Expander (SE) and Proactive Querying (PQ) modules.

2.5.1 Strategy Transformer

The ST module selects an instant strategy from the given global strategy $\mathbb{A} \subseteq \mathbb{A}^*$ based on the workload $\mathbb{W}$. Since our cache-aware MMM and RP modules build on the matrix mechanism, we require a few properties for this instant strategy $\mathbb{A}$ to run the former mechanisms with good utility. First, the strategy $\mathbb{A}$ should be a *support* to the workload $\mathbb{W}$ [LMH+15], that is, it must be possible to represent each query in $\mathbb{W}$ as a linear combination of strategy queries in $\mathbb{A}$. In other words, a solution matrix $\mathbb{X}$ should exist for the linear system $\mathbb{W} = \mathbb{X}\mathbb{A}$. Second, $\mathbb{A}$ should have a *low L_1 norm*, such that the privacy cost $\epsilon = \frac{\|\mathbb{A}\|_1}{b}$ for running MM is small, for a given a noise parameter b (Proposition ?). Third, using noisy responses to $\mathbb{A}$ to answer $\mathbb{W}$ should incur minimal *noise compounding* [HRMS10]. We minimize noise compounding by preferring strategy queries that

span a larger part of the domain; in particular, suppose a workload can be answered using two strategy queries: $\mathbb{A}_1 = \mathbb{A}_{[a,d]}^*$ and a set of strategies $\mathbb{A}_2 = \{\mathbb{A}_{[a,b]}^*, \mathbb{A}_{[b,d]}^*\}$, then we prefer $\mathbb{A}_1$ over $\mathbb{A}_2$. We thus present the strategy generator (SG) component, to address all of these requirements. The strategy generator only uses the global strategy $\mathbb{A}^*$, and does not use the cached responses, to generate an instant strategy $\mathbb{A}$ for the workload $\mathbb{W}$.

Last, we require that $\mathbb{A}$ must be mapped to a *full-rank* matrix $\mathbf{A}$, such that $\mathbf{A}^+ \tilde{\mathbf{y}}$ is the estimate of the mapped data vector $\mathbf{x}$ that minimizes the total squared error given the noisy observations $\tilde{\mathbf{y}}$ of the strategy queries $\mathbf{A}$ [LMH+15, Section 4]. We present a full-rank transform (FRT) component to address this last requirement. The ST module runs the SG and FRT components sequentially.

Strategy Generator. Our global strategy $\mathbb{A}^*$ is a k -ary tree over the full domain ($\text{dom}(\mathcal{R})$), hence, it supports all possible counting queries on the full domain; we use $k = 2$ in our implementation. A baseline instant strategy $\mathbb{A}$ just uses the full global strategy matrix ($\mathbb{A} = \mathbb{A}^*$), thus satisfying the first requirement. To answer the first workload, we obtain the noisy strategy responses for all nodes on the tree, thereby fully populating the cache and reusing the cached noisy responses for future workloads. However, this instant strategy has a very high norm $\|\mathbb{A}^*\|$, equal to the tree height $\log_k(n) + 1$, where n is the full domain size. Thus, answering the first workload would require spending a high upfront privacy budget. Moreover, this high upfront cost may not be amortized across future workload queries, for example, if the future queries do not require many nodes on this tree. Future workload queries may also have higher accuracy requirements, and we would thus need to re-sample noisy responses to the entire tree again, with a lower noise parameter.

To obtain a low norm strategy matrix, we only choose those strategy queries from $\mathbb{A}^*$ that support the workload $\mathbb{W}$. Intuitively, we wish to fill the cache with noisy responses to as many strategy queries as possible, thus we should bias our strategy generation algorithm towards the leaf nodes of the strategy tree. However, the DP noisy responses for the strategy nodes would be added up to answer the workload, and summing up responses to a large number of strategy leaf nodes compounds the DP noise in the workload response [HRMS10]. Thus, for each query in the workload $\mathbb{W}$, we apply a top-down tree traversal to fetch the *minimum* number of nodes in the strategy tree (and the corresponding queries in $\mathbb{A}^*$) required to answer this workload query. Then we include all these queries into the instant strategy $\mathbb{A}$ for this workload $\mathbb{W}$. The L_1 norm of the output strategy matrix is then simply the maximum number of nodes in any path

of the strategy tree, and it is upper-bounded by the tree height. We present an example strategy generation below.

Example 2.5.1. We continue with Example ?? shown in Figure ??, for an integer domain $[0, 8)$. For a workload with a single query, such as $\mathbb{W}_1 = \{\mathbb{w}\} = \{[0, 7)\}$, the first iteration of our SG workload decomposition algorithm computes the overlap of $\mathbb{w}$ with the left child of the root $c_1 = \mathbb{A}_{[0,4)}^*$ as $\mathbb{w}_{c1} = [0, 4)$ and the overlap with the right child $c_2 = \mathbb{A}_{[4,8)}^*$ as $\mathbb{w}_{c2} = [4, 7)$. The function only iterates once for the left child c_1 , directly outputting that node's range $\mathbb{A}_{[0,4)}^*$, as the base condition is satisfied (Lines ??–??, Algorithm ??). In the next iteration for the right child c_2 , the overlaps with both of its children are non-null ($[4, 6)$ with $\mathbb{A}_{[4,6)}^*$ and $[6, 7)$ with $\mathbb{A}_{[6,8)}^*$), and the corresponding strategy nodes are returned in subsequent iterations. Since $\mathbb{A}_1$ has no overlapping intervals, $\|\mathbb{A}_1\|_1 = 1 < \|\mathbb{A}^*\|_1$.

The second workload $\mathbb{W}_2$ has two queries with range predicates ($\mathbb{W}_2 = \{\mathbb{w}_1, \mathbb{w}_2\} = \{[2, 6), [3, 7)\}$). The first workload query predicate ($\mathbb{w}_1 = [2, 6)$) requires the strategy nodes $\mathbb{A}_{[2,4)}^*$ and $\mathbb{A}_{[4,6)}^*$, whereas the second query ($\mathbb{w}_2 = [3, 7)$) requires the following three nodes: $\mathbb{A}_{[3,4)}^*$, $\mathbb{A}_{[4,6)}^*$ and $\mathbb{A}_{[6,7)}^*$. Hence, the second instant strategy $\mathbb{A}_2$ is a set of all of these strategy nodes.

The global strategy has an L_1 norm $\|\mathbb{A}^*\|_1 = 4$. The matrix forms of $\mathbb{A}_1$ and $\mathbb{A}_2$ can be generated as shown in Example ?. Both strategy matrices improve over the global strategy in terms of their L_1 norms: $\mathbb{A}_1 = 1 < \|\mathbb{A}^*\|_1$ and $\mathbb{A}_2 = 2 < \|\mathbb{A}^*\|_1$. We observe that though $\mathbb{A}^*$ is full rank, due to the removal of strategy queries that do not support the workloads, both $\mathbb{A}_1$ and $\mathbb{A}_2$ are not full rank. $\square$

We formalize our strategy generation algorithm in the recursive function `DECOMPOSEWORKLOAD` given in Algorithm ?. This function takes as input a single workload RCQ $\mathbb{w}$ and a node v on the tree $\mathcal{T}$. It first checks if the input predicate matches the range interval for the node v . If it does, it returns that range interval (Line ??). Otherwise, for each child c of node v , it computes the *overlap* $\mathbb{w}_c$ of the range interval of that child with the interval $\mathbb{w}$ (Line ??). For instance, the overlap of the range interval $[2, 6)$ with $[0, 4)$ is given by $[\max\{(0, 2)\}, \min\{(4, 6)\}) = [2, 4)$. For each child with a non-null range interval overlap, the function is called recursively with that overlap $\mathbb{w}_c$ (Line ??). Our `GENERATESTRATEGY` function (Line ??) calls the `DECOMPOSEWORKLOAD` function with each RCQ $q \in Q$ as the first argument, and the root of the tree $\mathcal{T}$, as the second argument, and the latter function returns with the decomposition of q over all child nodes in the tree ($\mathbb{A}$). This function outputs the strategy $\mathbb{A}$, which is the union of the decomposition of each workload RCQ.

Algorithm 4 Strategy Transformer (ST).

```

1: function DECOMPOSEWORKLOAD( $\mathbb{w}$ , node  $v$ )
2:   if  $v$ .query ==  $\mathbb{w}$  then
3:     return  $v$ 
4:    $\mathbb{A} \leftarrow \emptyset$ 
5:   if  $v$  has children then
6:     for child  $c$  of node  $v$  do
7:        $\mathbb{w}_c \leftarrow \text{OVERLAPPINGRCQ}(\mathbb{w}, \text{node } c.\text{query})$ 
8:        $\nabla$  Given  $\mathbb{w} = [ws, we)$ , node  $c.\text{query} = [cs, ce)$ , this function returns a tuple
          $[\max(ws, cs), \min(we, ce))$  or  $\emptyset$  if it does not exist.
9:       if  $\mathbb{w}_c \neq \emptyset$  then
10:         $\mathbb{A} \leftarrow \mathbb{A} \cup \text{DECOMPOSEWORKLOAD}(\mathbb{w}_c, \text{node } c)$ 
11:   return  $\mathbb{A}$ 

12: function GENERATESTRATEGY( $Q$ )
13:   Initialize  $\mathbb{A}$  and  $\mathbb{W}$  to be empty matrices.
14:   for  $q \in Q$  do  $\triangleright$  Iterate over each RCQ  $q$  in the workload  $Q$ .
15:      $[s_1, e_1) \leftarrow q$ 
16:      $\nabla$  GETMATRIXROW generates a row of the length of the domain vector  $\mathbf{x}$  that has a
       one only for all indices in the input range and a zero otherwise.
17:      $\mathbb{W} \leftarrow \mathbb{W} \parallel \text{GETMATRIXROW}([s_1, e_1))$   $\triangleright$  Append a row to  $\mathbb{W}$  for each workload RCQ.
18:      $\nabla \mathcal{T}$  is the strategy tree corresponding to the global strategy  $\mathbb{A}^*$ .
19:      $\mathbb{A} \leftarrow \text{DECOMPOSEWORKLOAD}(q, \mathcal{T}.\text{root})$ 
20:     for  $\mathbb{a} \in \mathbb{A}$  do  $\triangleright$  Iterate through each strategy RCQ.
21:        $[s, e) \leftarrow \mathbb{a}$ 
22:        $\mathbb{A} \leftarrow \mathbb{A} \parallel \text{GETMATRIXROW}([s, e))$   $\triangleright$  Append a row to  $\mathbb{A}$  for each strategy RCQ.
23:   return  $\mathbb{A}, \mathbb{W}$ 

```

Full Rank Transformer (FRT). We transform an instant strategy matrix $\mathbb{A}$ to another full-rank matrix $\mathbf{A}$ by mapping the full domain $\text{dom}(\mathcal{R})$ of size n to a subset of the full domain of $n' \leq n$ non-overlapping counting queries or buckets. The resulting subset should still support all the queries in the instant raw strategy $\mathbb{A}$ output by our SG. For efficiency, the subset should have the smallest possible number of buckets such that the transformed strategy $\mathbf{A}$ will be full rank. First, we define a domain transformation matrix $\mathbb{T}$ of size $n' \times n$ that transforms the data vector $\mathbf{x}$ over the full domain to the smaller data vector $\mathbf{x}$, such that $\mathbf{x} = \mathbb{T}\mathbf{x}$. Using $\mathbb{T}$, we can then transform a raw $\mathbb{A}$ to $\mathbf{A}$.

Definition 2.5.1 (Transformation Matrix). Given a subset of n' non-overlapping buckets over the full domain $\text{dom}(\mathcal{R})$, if the i th value in $\text{dom}(\mathcal{R})$ is in the j th bucket, $\mathbb{T}[j, i] = 1$;

else, $\mathbb{T}[j, i] = 0$.

We elaborate on how we create a transformation matrix $\mathbb{T}$ to support the strategy $\mathbb{A}$, as presented in GETTRANSFORMATIONMATRIX in Algorithm ???. To create $\mathbb{T}$, GETTRANSFORMATIONMATRIX starts with the first row of $\mathbb{A}$ (Line ??), then iterates through the remaining rows (Line ??), updating the transformation matrix $\mathbb{T}$ as needed. If row i is disjoint from all buckets, we simply add this row as a new bucket (Line ??). Otherwise, we construct a new bucket matrix $\mathbb{T}'$. To do this, we first copy all rows of $\mathbb{T}$ that do not intersect with the current row of $\mathbb{A}$ (Line ??). Then, for the buckets that do intersect, we remove the intersection from that bucket and add a new bucket containing the intersection (Line ??). Finally, if the row is a super-set of some buckets, we add the part of the row that is not covered by the buckets as a new bucket (Line ??).

The TRANSFORMSTRATEGY function in Algorithm ?? transforms $\mathbb{A}$ to $\mathbf{A}$, using the transformation matrix $\mathbb{T}$ to determine which buckets are used for each query. We first initialize $\mathbf{A}$ to be the zero matrix (Line ??). For row i of $\mathbb{A}$ and row j of $\mathbb{T}$, we compute whether bucket j is needed to answer row i by checking if $\mathbb{T}[j] \subseteq \mathbb{A}[i]$ (Line ??). If bucket j is needed, we set the corresponding entry of $\mathbf{A}$ to 1 (Line ??).

Example 2.5.2. We consider $\mathbb{A}_2$ from example ??. The domain vector $\mathbf{x}$ consists of the leaves of the tree depicted in Figure ??. We get the following raw matrix form for $\mathbb{A}_2$.

$$\mathbb{A}_2 = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \mathbf{A}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We generate $\mathbf{A}_2$ above using $\mathbb{T}$:

$$\mathbb{T} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

We note that while $\mathbb{A}_2$ and $\mathbf{A}_2$ both happen to be full-rank matrices, in general, $\mathbb{A}_2$ may not be a full rank matrix. We also note that $\mathbf{A}_2$ has fewer columns; this is relevant for efficiency and in particular, faster matrix operations.

The ST module finally outputs $\mathbb{A}$, $\mathbf{A}$, as well as the transformation matrix, as it can be used to transform $\mathbb{W}$. We use the full-rank versions $\mathbf{W}$, $\mathbf{A}$ for all invocations of the matrix mechanism (i.e., computing $\mathbf{WA}^+$).

Algorithm 5 Full-rank Transformer (Section ??)

```

1: function GETTRANSFORMATIONMATRIX( $\mathbb{A}$ )
2:    $\mathbb{T} = \{\mathbb{A}[0]\}$ 
3:   for  $i \in \{1, 2, \dots, |\mathbb{A}|\}$  and  $\mathbb{A}[i] \notin \mathbb{T}$  do
4:     if  $\mathbb{t} \cdot \mathbb{A}[i] = 0 \forall \mathbb{t} \in \mathbb{T}$  then  $\triangleright \mathbb{A}[i]$  is disjoint with all  $\mathbb{t} \in \mathbb{T}$ 
5:        $\mathbb{T} \leftarrow \mathbb{T} \cup \{\mathbb{A}[i]\}$   $\triangleright$  Add  $\mathbb{A}[i]$  to  $\mathbb{T}$ .
6:     else
7:        $\mathbb{T}' \leftarrow \{\mathbb{t} \in \mathbb{T} \mid \mathbb{t} \cdot \mathbb{A}[i] = 0\}$   $\triangleright$  Copy buckets that do not overlap with  $\mathbb{A}[i]$  to  $\mathbb{T}'$ .
8:        $\mathbb{T}_1 \leftarrow \mathbb{T}'$   $\triangleright$  Copy  $\mathbb{T}'$  to  $\mathbb{T}_1$ .
9:       for  $\mathbb{t}$  in  $\mathbb{T}$  and  $\mathbb{t} \cdot \mathbb{A}[i] \neq 0$  do  $\triangleright$  Overlapping buckets.
10:         $\triangleright$  Remove overlap from  $\mathbb{t}$ , add it as a new bucket.
11:         $\mathbb{T}' \leftarrow \mathbb{T}' \cup (\mathbb{t} \cap \mathbb{A}[i]) \cup (\mathbb{t} - \mathbb{A}[i])$ 
12:         $\mathbb{S} \leftarrow \{\mathbb{t} \mid \mathbb{t} \in \mathbb{T}' \text{ and } \mathbb{t} \cdot \mathbb{A}[i] \neq 0\}$   $\triangleright$  Set of buckets in  $\mathbb{T}'$  that still overlap with  $\mathbb{A}[i]$ .
13:         $\mathbb{T}' \leftarrow \mathbb{T}' \cup (\mathbb{A}[i] - \mathbb{S})$   $\triangleright$  When  $\mathbb{A}[i]$  is a superset of  $\mathbb{S}$ , add remaining rows into  $\mathbb{T}'$ .
14:         $\mathbb{T} \leftarrow (\mathbb{T} - \mathbb{T}_1) \cup \mathbb{T}'$   $\triangleright$  Drop overlapping buckets  $\mathbb{T}_1$  from  $\mathbb{T}$ , add disjoint buckets  $\mathbb{T}'$ .
15:   return  $\mathbb{T}$ 

16: function TRANSFORMSTRATEGY( $\mathbb{A}$ )
17:    $\mathbb{T} \leftarrow \text{GETTRANSFORMATIONMATRIX}(\mathbb{A})$ 
18:    $\mathbf{A} \leftarrow \text{FRT}(\mathbb{A}, \mathbb{T})$ 
19:   return  $\mathbf{A}, \mathbb{T}$ 

20: function FRT( $\mathbb{B}, \mathbb{T}$ )
21:   Initialize  $\mathbf{B}$  as a  $|\mathbb{B}| \times |\mathbb{T}|$  zero-valued matrix
22:   for  $i \in \{1, 2, \dots, |\mathbb{B}|\}$  and  $j \in \{1, 2, \dots, |\mathbb{T}|\}$  do
23:     if  $\mathbb{T}[j] \subseteq \mathbb{B}[i]$  then  $\triangleright$  Determine whether bucket  $j$  is needed to answer  $\mathbb{B}[i]$ .
24:        $\mathbf{B}[i, j] \leftarrow 1$ 
25:   return  $\mathbf{B}$ 

```

Lemma 2.5.1. Given a global strategy $\mathbb{A}^*$ in a k -ary tree structure, and an instant strategy $\mathbb{A} \subseteq \mathbb{A}^*$, running $\text{GETTRANSFORMATIONMATRIX}(\mathbb{A})$ in Algorithm ?? increases the number of non-empty buckets in $\mathbb{T}$ by at most one, for each row $\mathbb{A}[i] \in \mathbb{A}$.

Proof. If the condition in line ?? is met the result follows trivially. We consider the remaining case where $\mathbb{A}[i]$ intersects with at least one bucket, and we assume, for the inductive case, that all buckets in $\mathbb{T}$ are disjoint. For each new row $\mathbb{A}[i]$, if this row intersects with the buckets in $\mathbb{T}$, then $\mathbb{A}[i]$ is either (i) a descendant node of one bucket in $\mathbb{T}$, or (ii) an ancestor node of one or more buckets in $\mathbb{T}$. For the first case, assume $\mathbb{A}[i]$ is a descendant node of $\mathbb{t} \in \mathbb{T}$. It cannot be the descendant of other buckets, as the other buckets are disjoint with $\mathbb{t}$. Then $\mathbb{t}$ will be replaced by two disjoint buckets, namely

$\mathbb{T} \cap \mathbb{A}[i]$ and $\mathbb{T} - \mathbb{A}[i]$ (line ??), and hence the size of $\mathbb{T}$ increases by 1. For the second case, assume $\mathbb{A}[i]$ is the ancestor node of multiple buckets in $\mathbb{T}$. We denote these buckets as $\{\mathbb{T}[j_1], \dots, \mathbb{T}[j_k]\}$, then all these buckets will remain the same (line ?? just adds $\mathbb{T}$), and at most one additional bucket $\mathbb{A}[i] - \sum_{\mathbb{T} \in \mathbb{T}' \wedge \mathbb{A}[i] \neq 0} \mathbb{T}$ is added in line ??.

□

We now prove that TRANSFORMSTRATEGY ensures full rank matrices.

Theorem 2.5.1. Given a global strategy $\mathbb{A}^*$ in a k -ary tree structure, and an instant strategy $\mathbb{A} \subseteq \mathbb{A}^*$, TRANSFORMSTRATEGY outputs a strategy $\mathbf{A}$ that is full rank and supports $\mathbb{A}$.

Proof. We begin by discussing how $\mathbf{A}$ represents the queries in $\mathbb{A}$. First, we note that $\mathbf{A}$ and $\mathbb{A}$ have the same number of rows. The only difference is that $\mathbf{A}$ represents the queries on the data vector $\mathbf{x}$, whereas $\mathbb{A}$ uses $\mathbf{x}$. By construction, we have $\mathbb{A}\mathbf{x} = \mathbf{A}\mathbb{T}\mathbf{x} = \mathbf{A}\mathbf{x}$.

Since $\mathbf{A}$ is a representation of $\mathbb{A}$ using the only the buckets generated in GETTRANSFORMATIONMATRIX, the number of columns in $\mathbf{A}$ is the same as the number of buckets, $|\mathbb{T}|$. To show that $\mathbf{A}$ is always full rank, we will first show that $|\text{row}(\mathbf{A})| \geq |\text{col}(\mathbf{A})|$, where $|\text{row}(\mathbf{A})|$ and $|\text{col}(\mathbf{A})|$ represent the number of rows and the number of columns in $\mathbf{A}$. Or equivalently, $|\text{row}(\mathbf{A})| \geq |\mathbb{T}|$. This follows by inductively applying Lemma ??.

Next we show that $\text{rank}(\mathbf{A}) = |\text{col}(\mathbf{A})|$. We once again show this by induction on the number of rows in $\mathbb{A}$. In the base case ($|\text{row}(\mathbb{A})| = 1$) applying TRANSFORMSTRATEGY, we get $\mathbf{A} = [1]$ and thus $\text{rank}(\mathbf{A}) = 1$. Now we assume that $\text{rank}(\mathbf{A}) = |\text{col}(\mathbf{A})|$ for some $\mathbf{A}$ obtained from $\mathbb{A}$. We consider adding a new row r_{new} to $\mathbb{A}$ and define $\bar{\mathbb{A}} = \mathbb{A} \cup r_{\text{new}}$. Let $\bar{\mathbf{A}}$ represent the result of applying TRANSFORMSTRATEGY to $\bar{\mathbb{A}}$.

When adding this row there are two possible cases: First, consider the case where the newly added row, r_{new} , can be represented as a linear combination of $\mathbb{T}$. That is, the buckets created in GETTRANSFORMATIONMATRIX are the same for $\mathbb{A}$ and $\bar{\mathbb{A}}$. Thus $|\text{col}(\bar{\mathbf{A}})| = \text{rank}(\bar{\mathbf{A}})$ since the number of linearly independent columns (or buckets) did not increase or decrease by adding a row. On the other hand, if the newly added row, r_{new} , cannot be represented as a linear combination of $\mathbb{T}$, then r_{new} must be linearly independent of $\mathbb{A}$. Thus, $\bar{\mathbf{A}}$ has one additional linearly independent row and thus $\text{rank}(\bar{\mathbf{A}}) = \text{rank}(\mathbf{A}) + 1$. Furthermore by Lemma ??, we know that adding a row can add at most one new bucket. Since we assume r_{new} cannot be represented as a linear combination of $\mathbb{T}$, this means $|\text{col}(\bar{\mathbf{A}})| = |\text{col}(\mathbf{A})| + 1$. Thus we have that $|\text{col}(\bar{\mathbf{A}})| = \text{rank}(\bar{\mathbf{A}})$. Combining the fact that $|\text{row}(\mathbf{A})| \geq |\text{col}(\mathbf{A})|$ and $\text{rank}(\mathbf{A}) = |\text{col}(\mathbf{A})|$, it follows that $\mathbf{A}$ is full rank.

□

$$\begin{aligned}
\mathbf{W}_1 &= \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{A}_1 = I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{W}_1 \mathbf{A}_1^+ = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{b}_1 = \begin{bmatrix} b \\ b \\ 5b \end{bmatrix} \\
\mathbf{A}_{1e} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \mathbf{W}_1 \mathbf{A}_{1e}^+ = \begin{bmatrix} 0.75 & -0.25 & -0.25 & 0.25 \\ 0.5 & 0.5 & -0.5 & 0.5 \\ -0.25 & -0.25 & 0.75 & 0.25 \end{bmatrix}, \mathbf{b}_{1e} = \begin{bmatrix} b \\ b \\ \frac{5b}{4b} \end{bmatrix} \\
\mathbf{W}_1 \mathbf{A}_1^+ \text{diag}(\mathbf{b}_1) &= \begin{bmatrix} 1.00b & 0 & 0 \\ 1.00b & 1.00b & 0 \\ 0 & 0 & 5.00b \end{bmatrix} \\
\mathbf{W}_1 \mathbf{A}_{1e}^+ \text{diag}(\mathbf{b}_{1e}) &= \begin{bmatrix} 0.750b & -0.250b & -1.25b & 1.00b \\ 0.500b & 0.500b & -2.50b & 2.00b \\ -0.250b & -0.250b & 3.75b & 1.00b \end{bmatrix} \\
\|\mathbf{W}_1 \mathbf{A}_1^+\| &= 4 > \|\mathbf{W}_1 \mathbf{A}_{1e}^+\| = 2.5, \\
\|\mathbf{W}_1 \mathbf{A}_1^+ \text{diag}(\mathbf{b}_1)\| &= 28b^2 < \|\mathbf{W}_1 \mathbf{A}_{1e}^+ \text{diag}(\mathbf{b}_{1e})\| = 29.1b^2
\end{aligned}$$

FIGURE 2.5: An input instant strategy $\mathbf{A}_1$ is expanded to a strategy $\mathbf{A}_{1e}$. Using their noise parameter vectors $(\mathbf{b}_1, \mathbf{b}_{1e})$, we can see that $\mathbf{A}_{1e}$ has an error term of larger magnitude.

2.5.2 Strategy Expander

We recall that our goal with CacheDP is to use cached strategy responses, in order to save privacy budget on new strategy queries. Section ?? shows that MMM achieves this goal by directly reusing accurate strategy responses from the cache for the basic instant matrix, i.e., by selecting $\mathbb{F} \subseteq \mathcal{C} \cap \mathbb{A}$. In this strategy expander (SE) module, we provide efficient heuristics to add cached strategy entries from $\mathcal{C} - \mathbb{A}$, to $\mathbb{A}$ in order to save more privacy budget. Since these additional entries are cached, they are added to $\mathbb{F}$ within $\mathbb{A}$. Thus, within the expanded strategy, we have $\mathbb{F} \subseteq \mathcal{C}$ and as such, the SE module addresses a generalization of the CE problem (??) wherein $\mathbb{F} \subseteq \mathcal{C} \cap \mathbb{A}$. We also note that these additional cached strategy entries do not need to support the workload.

Example 2.5.3. Consider the cache structure $\mathcal{C}$ in Figure ?? and a new workload $\mathbb{W}_1 = \{[0, 1), [0, 2), [2, 4)\}$ and so $\mathbb{A}_1 = \{\mathbb{A}_{[0,1)}^*, \mathbb{A}_{[1,2)}^*, \mathbb{A}_{[2,4)}^*\}$. The cache includes entries for

$\mathbb{A}_{[0,1)}^*, \mathbb{A}_{[1,2)}^*$ at noise parameter b , as well as $\mathbb{A}_{[0,4)}^*$ at $4b$. The MMM module decides to reuse the first two cache entries, and pay for $\mathbb{A}_{[2,4)}^*$ at $5b$, resulting in the noise parameter vector $\mathbf{b}_1$, as depicted in Figure ?? . The SE problem is to decide which cached responses (such as $\mathbb{A}_{[0,4)}^*$) can be added to the strategy to reduce its cost.

Consider a strawman solution to choosing cache entries: we simply add all strategy queries from $\mathcal{C} - \mathbb{A}$ to $\mathbb{A}$, in order to obtain an expanded strategy $\mathbb{A}_e$. We remind the reader that the error term for MMM includes the product of $\mathbf{W}\mathbf{A}^+$ and the noise vector $\text{Lap}(\mathbf{b}_F \| \mathbf{b}_F)$ (Proposition ??). Li et al. show that when expanding the strategy, the Frobenius norm of the first term of this product reduces; that is, $\|\mathbf{W}\mathbf{A}_e^+\|_F \leq \|\mathbf{W}\mathbf{A}^+\|_F$ [LMH+15, Theorem 6]. Combining this result with the assumption that all strategy queries are answered using the *same* Laplace distribution [LMH+15, Proposition 9], they conclude that adding more queries to $\mathbf{A}$ always reduces the error of the matrix mechanism. However, our cached noisy responses can be drawn at different noise parameters, and in general the noise vector $\text{Lap}(\mathbf{b}_F \| \mathbf{b}_F)$ consists of noise drawn from *different* Laplace distributions. Thus, the aforementioned result does not hold. In Figure ??, we present a counterexample for Li et al.'s result, using a noise vector with different noise parameters.

Example 2.5.4. Continuing Example ??, we expand $\mathbf{A}_1$ to $\mathbf{A}_{1e}$ by adding a row $\mathbb{A}_{[0,4)}^*$. In Figure ??, we compute the α^2 -expected error using both $\mathbf{A}_1$ and $\mathbf{A}_{1e}$ and find that $\mathbf{A}_{1e}$ has a larger error term.

We can see that the strawman solution can lead to a strategy with an increased error term. Importantly, this figure shows that adding a strategy query results in changed coefficients in $\mathbf{W}\mathbf{A}_e^+$, that is, this added query changes the weight with which noisy responses to the original strategy queries are used to form the workload response. The added strategy query response must also be accurate, since adding a large, cached noise parameter to $\mathbf{b}_e$ will also likely increase the magnitude of the error term (recall the example in Figure ??).

In an optimal solution to this problem, one would have to consider adding each combination of cache entries from $\mathcal{C} - \mathbb{A}$. This induces an exponentially large search space of $O(2^{|\mathcal{C}|})$ possible solutions for $\mathbb{A}_e$. Then for each candidate $\mathbb{A}_e$ we need to evaluate this error term and compare it to the error for the original strategy. Instead of navigating this large search space for $\mathbb{A}_e$, we propose a series of efficient heuristics to obtain a greedy solution to this problem, as presented in Algorithm ?? . In designing our algorithm, we have three goals:

1. *Search space*: Reduce the search space from $O(2^{|\mathcal{C}|})$ to $O(|\mathcal{C}|)$.
2. *Efficiency*: Ensure that the additional strategy rows do not significantly increase the run-time of CacheDP.
3. *Greediness*: Select strategy rows that are most likely to reduce the privacy budget from that for MMM (ϵ_P).

We achieve the first goal above by conducting a single lookup over cache entries in $\mathcal{C} - \mathbb{A}$ (Line ??), which would only incur a worst-case complexity of $O(|\mathcal{C}|)$. We limit the number of selected cached strategy rows to λ (Line ??), thereby achieving our second goal of efficiency. Our greediness heuristics to select a strategy query are based on the two aforementioned factors that impact the workload error term, namely, the *accuracy* of its cached noisy response, and how the noisy response is *related* to noisy responses to the original strategy.

First, we must ensure that the strategy queries selected from $\mathcal{C} - \mathbb{A}$ are accurate enough. Before conducting our cache lookup, we sort our cache entries in increasing order of the noise parameter. Therefore our algorithm greedily prefers more accurate cache entries. Recall that the `MMM.ESTIMATEPRIVACYBUDGET` interface outputs the noise parameter b_P . Just as we used b_P to compute $\mathbf{F}$, we can also use it to select cache entries for $\mathbb{A}_e$ that are at least as accurate as other entries in $\mathbf{F}$. These accurate cached responses will likely improve the accuracy of the workload response. Thus, out of cache entries in $\mathcal{C} - \mathbb{A}$, we only consider cache entries whose noise parameter is lower than b_P (Line ??).

Second, our heirarchical global strategy $\mathbb{A}^*$ structures cache entries, and induces relations between the cached noisy responses. Relatedly, Hay et al. [HRMS10] describe the constrained inference problem; it focuses on minimizing the error term for multiple noisy responses, while following consistency constraints among them. For example, if we add the strategy queries corresponding to the siblings and parent nodes of an existing query in $\mathbb{A}$, we obtain an additional consistency constraint which tends to reduce error. However, if we only added the sibling node, we would not have seen as significant (if any) improvement. Thus, our second greedy heuristic, in Line ??, ensures that each query $\mathfrak{a}$ added to $\mathbb{A}_e$ is an ancestor or a descendant of an existing query $\mathfrak{a}' \in \mathbb{A}$.

We analyze the conditions under which our heuristics result in SE module being selected. The SE module is only selected if it has a lower cost than the original strategy in MMM. The privacy cost for our mechanisms is inversely proportional to the error term, for a given (α, β) or α^2 -expected total square error. Thus, our heuristics are only

successful if they lead to the error term for the SE module to be smaller than the error term for MMM.

We recall that Figure ?? exemplified that expanding the strategy can lead to an increased error term. We analyze why such situations can arise and describe conditions for when our noise parameter-based heuristic can reduce the error. Our heuristic to choose $b_{\ell+1} < b_P$ strictly improves the error under the following condition:

Theorem 2.5.1. Given a workload $\mathbf{W}$, a strategy $\mathbf{A}$ of ℓ rows, and a noise vector $\mathbf{b}$, adding a new row to $\mathbf{A}$ to form $\mathbf{A}_e$ reduces the error, i.e.,

$$\|\mathbf{W}\mathbf{A}^+ \text{diag}(\mathbf{b})\|_F^2 \geq \|\mathbf{W}\mathbf{A}_e^+ \text{diag}(\mathbf{b}||b_{\ell+1})\|_F^2 \quad (2.11)$$

if all entries in $\mathbf{b}$ equal b^* and $b_{\ell+1} < b_P \leq b^*$, for any b^* .

Proof. We recall the following theorem proved by Li et al. [LMH+15] in their MM paper.

$$\|\mathbf{W}\mathbf{A}^+\|_F^2 \geq \|\mathbf{W}\mathbf{A}_e^+\|_F^2 \quad (2.12)$$

For $\mathbf{b} = [b^* \cdots b^*]$, we transform Equation ?? to:

$$\|\mathbf{W}\mathbf{A}^+ \text{diag}(\mathbf{b})\|_F^2 = \|\mathbf{W}\mathbf{A}^+(b^*I)\|_F^2 = (b^*)^2 \|\mathbf{W}\mathbf{A}^+\|_F^2 \quad (2.13)$$

$$\geq (b^*)^2 \|\mathbf{W}\mathbf{A}_e^+\|_F^2 \quad (2.14)$$

$$= \|\mathbf{W}\mathbf{A}_e^+ \text{diag}(\mathbf{b}||b^*)\|_F^2 \quad (2.15)$$

$$\geq \|\mathbf{W}\mathbf{A}_e^+ \text{diag}(\mathbf{b}||b_{\ell+1})\|_F^2 \quad (2.16)$$

where the first inequality comes from applying Li et al.'s result (Equation ??) and the final inequality is obtained by applying the condition: $b_{\ell+1} \leq b^*$. $\square$

When the noise parameters are not all equal, the sufficiency conditions become more complicated. For example, if we modify the above counterexample to have a slightly more accurate expanded row ($b_{\ell+1} = 3b < 4b$), we get a lower error than MMM:

$$\|\mathbf{W}_1 \mathbf{A}_1^+ \text{diag}(\mathbf{b}_1)\| = 28b^2, \quad \|\mathbf{W}_1 \mathbf{A}_{1e}^+ \text{diag}(\mathbf{b}_{1e})\| = 26.5b^2$$

Thus our greedy heuristic, which does not simply add entries less than b_p , but prioritizes the most accurate rows first, would be effective in this scenario. We also observe that distribution of the existing noise parameters in $\mathbf{b}$ is important. If the noise parameters are all close to each other, strategy expansion is likely to be more beneficial. For example

changing the second entry of $\mathbf{b}_1$ from b to $2b$ (i.e. $\mathbf{b}_{1e} = [b, 2b, 5b, 4b]$), results in a smaller error than MMM:

$$\|\mathbf{W}_1 \mathbf{A}_1^+ \text{diag}(\mathbf{b}_1)\| = 31.0b^2, \quad \|\mathbf{W}_1 \mathbf{A}_{1e}^+ \text{diag}(\mathbf{b}_{1e})\| = 30.2b^2$$

It is evident that both the distribution of the noise parameters in $\mathbf{b}$ and the new noise parameter $b_{\ell+1}$, influence the success of our accuracy heuristic. Additionally, we observe that a newly added row could satisfy our final heuristic by being an ancestor or descendant of *any* of the existing rows in $\mathbf{A}_1$. We find that the structure of the expanded strategy $\mathbf{A}_{1e}$ also significantly influences the error term. For example, changing the final row in $\mathbf{A}_{1e}$ from $(1, 1, 1)$ to $(1, 0, 1)$ or $(0, 1, 1)$ also reduces the error for $\mathbf{A}_{1e}$ to $26.5b^2$ and $20b^2$ respectively, under the exact same noise parameters as our counterexample. However, changing the final row to $(1, 1, 0)$ increases the error to $34.7b^2$.

We conclude that whether an additional row selected by our noise parameter heuristics will succeed in decreasing the error for SE over MMM, depends on how the new row changes elements in $\mathbf{W}\mathbf{A}_e^+$. However, we can guarantee that when the noise parameters in $\mathbf{b}$ are sufficiently similar, our strategy expander will reduce the error regardless of the structure. Our structure-based heuristic only allows strategy queries related through a parent-child relationship to an existing strategy query in $\mathbf{A}$. Future research may model the success of this heuristic, by analyzing the relation between $\mathbf{W}\mathbf{A}^+$ and $\mathbf{W}\mathbf{A}_e^+$ for $\mathbf{A}$ and $\mathbf{A}_e$ that differ by a parent or child row.

Integration. To summarize, the GENERATEEXPANDEDSTRATEGY function takes as input the original strategy, the cache, and the paid noise parameter b_P . Specifically, this paid noise parameter is the one output by ESTIMATEPRIVACYBUDGET implementation of the MMM module (Algorithm ??, Line ??). As such, this imposes a chronological order between the MMM and SE modules; MMM’s ESTIMATEPRIVACYBUDGET implementation needs to be run before the GENERATEEXPANDEDSTRATEGY function is run. This function generates an expanded strategy $\mathbf{A}_e$ and outputs its full-rank form $\mathbf{A}_e$ (Line ??). The privacy budget for $\mathbf{A}_e$ is then estimated using the ESTIMATEPRIVACYBUDGET implementation of the MMM module.

We encapsulate SE as a module rather than integrate it with MMM, since our heuristics might fail and the expanded strategy might cost a higher privacy budget than the original strategy used by MMM, as illustrated in Example ??. Since Algorithm ?? chooses to run the ANSWERWORKLOAD interface for the module and strategy with the lowest privacy cost, in the above case, $\mathbf{A}_e$ is simply not used.

Algorithm 6 Strategy Expander (SE) (Section ??)

```
1: function GENERATEEXPANDEDSTRATEGY( $\mathbb{A}, \mathcal{C}, b_{\mathbf{P}}$ )
2:    $\mathbb{A}_e \leftarrow \mathbb{A}$ 
3:   for  $(\mathbf{a}, b, \tilde{y}, t) \in (\mathcal{C} - \mathbb{A})$  in an ascending order of  $b$  do
4:      $\nabla$  Efficiency and greediness (accuracy) heuristics.
5:     if  $|\mathbb{A}_e| \geq |\mathbb{A}| + \lambda$  or  $b < b_{\mathbf{P}} \triangleright \lambda$  rows have already been added, or, the cached
       entry is too inaccurate. then
6:       break  $\triangleright$  Remaining cached entries would also be inaccurate due to sort-
         ing by  $b$ .
7:       if  $\mathbf{a}' \cdot \mathbf{a} \neq 0$  for some  $\mathbf{a}' \in \mathbb{A}$  then  $\triangleright$  Ancestor-descendant greediness heuristic.
8:          $\mathbb{A}_e \leftarrow \mathbb{A}_e \cup \{\mathbf{a}\}$ 
9:    $(\mathbf{A}_e, \mathbb{T}_e) \leftarrow \text{ST.TRANSFORMSTRATEGY}(\mathbb{A}_e) \triangleright \text{Algorithm ??, Line ??}$ 
10:  return  $\mathbb{A}_e, \mathbf{A}_e, \mathbb{T}_e$ 
```

2.5.3 Proactive Querying

The proactive querying (PQ) module is an optional module for MMM. The MMM obtains fresh noisy responses only for the paid strategy matrix $\mathbf{P}$, and inserts them into the cache. The goal of the PQ module is to proactively populate the cache with noisy responses to a subset $\Delta\mathbb{P}$ out of the remaining, non-cached strategy queries of the global strategy $(\mathbb{A}^* - \mathcal{C} - \mathbb{P})$, where $\mathbb{P}$ corresponds to the raw, non-full rank form of $\mathbf{P}$. Thus, we run the PQ module in the function `MMM.ANSWERWORKLOAD( $\cdot$ )` after obtaining the paid strategy matrix $\mathbf{P}$. Our cache-aware modules, including MMM, RP and SE, can use the cached noisy responses to $\Delta\mathbb{P}$ to answer future instant strategy queries. We wish to satisfy this goal without consuming any additional privacy budget over the MMM.

We first motivate key constraints for the PQ algorithm. First, we do not assume any knowledge of future workload query sequences. However, all future workload queries will be transformed into instant strategy matrices, and our cache-aware mechanisms will look up the cache for cached strategy rows. Second, we also do not know the accuracy requirements for future workload queries. Future workloads may be asked at different accuracy requirements than the current workload. Thus, we choose to obtain responses to $\Delta\mathbb{P}$ at the highest possible accuracy requirements without spending any additional privacy budget over that required for $\mathbf{P}$ by MMM, which is $\epsilon = \frac{\|\mathbf{P}\|_1}{b_{\mathbf{P}}}$. Our key insight is to generate $\Delta\mathbb{P} \subseteq (\mathbb{A}^* - \mathcal{C} - \mathbb{P})$ such that $\|\mathbb{P} \cup \Delta\mathbb{P}\|_1 = \|\mathbb{P}\|_1$. Therefore, answering both instant strategies ($\mathbb{P}$ and $\Delta\mathbb{P}$) with the Laplace mechanism using $b_{\mathbf{P}}$ costs no more privacy budget than simply answering $\mathbb{P}$ at $b_{\mathbf{P}}$.

We begin with the k -ary tree representation of the global strategy $\mathbb{A}^*$, denoted by $\mathcal{T}$.

Algorithm 7 Proactive Querying (PQ) (Section ??)

```

1: function SEARCHPROACTIVENODES( $v, r, \mathcal{C}, \Delta^{\mathbb{P}}$ )
2:    $\nabla$  First check if the node should be added to  $\Delta^{\mathbb{P}}$  and set  $r$  for recursive calls.
3:   if  $v \in \mathbb{P}$  then  $\triangleright$  Check if the node's RCQ is in  $\mathbb{P}$ 
4:      $r \leftarrow r - 1$ 
5:   else if  $v \notin \mathcal{C}$  and  $\mathcal{S}_{\mathbb{P}}(v) < r$  then  $\triangleright v$  is not cached and its subtree norm is less than  $r$ .
6:      $\Delta^{\mathbb{P}} \leftarrow \Delta^{\mathbb{P}} \cup \{v\}$   $\triangleright$  Add the node's RCQ to the proactively fetched strategy matrix.
7:      $r \leftarrow r - 1$ 
8:    $\nabla$  Only call the function recursively if any nodes can be fetched from its subtree.
9:   if  $v$  has children and  $r > 0$  then
10:    for child  $c$  of node  $v$  do
11:      SEARCHPROACTIVENODES( $c, r, \mathcal{C}, \Delta^{\mathbb{P}}$ )

12: function GENERATEPROACTIVESTRATEGY( $\mathbb{P}, \mathcal{C}$ )
13:    $\Delta^{\mathbb{P}} \leftarrow \emptyset$ 
14:   SEARCHPROACTIVENODES( $\mathcal{T}.\text{root}, \|\mathbb{P}\|_1, \mathcal{C}, \Delta^{\mathbb{P}}$ )  $\triangleright$  Initial call.
15:    $(\Delta^{\mathbb{P}}, \mathbb{T}) \leftarrow \text{ST.TRANSFORMSTRATEGY}(\Delta^{\mathbb{P}})$   $\triangleright$  Algorithm ??, Line ??
16:   return  $\Delta^{\mathbb{P}}$ 

```

This is a directed tree with edges directed from the root to leaves and all paths refer to paths from a node to its leaves. For a node $v \in \mathcal{T}$, we define an indicator function $\mathcal{M}_{\mathbb{P}}(v)$ to mark a node if its corresponding query is in $\mathbb{P}$.

Definition 2.5.2. We define the subtree norm of a node v as the maximum number of marked nodes across all paths p from node v to all of its leaves in its subtree, i.e.,

$$\mathcal{S}_{\mathbb{P}}(v) = \max_{p \in \text{subtree}(v)} \sum_{v \in p} \mathcal{M}_{\mathbb{P}}(v) \quad (2.17)$$

The subtree norm of a node can be computed recursively as the sum of the mark function for that node and the maximum subtree norm of all of its children nodes, if any. The proactive module first recursively computes the subtree norm of each node before generating $\Delta^{\mathbb{P}}$. This step requires a single top-down traversal of the strategy decomposition tree. Given that the children of each node have non-overlapping ranges, the subtree norm of the root of the tree is equal to the L_1 norm of $\mathbb{P}$:

Lemma 2.5.2. The L_1 norm of the $\mathbb{P}$ matrix is equal to the subtree norm of the root of the tree with marked nodes corresponding to $\mathbb{P}$:

$$\mathcal{S}_{\mathbb{P}}(\mathcal{T}.\text{root}) = \|\mathbb{P}\|_1 \quad (2.18)$$

Proof. We note that $\mathbb{P}$ is a subset of $\mathbb{A}$, which is formed through the DECOMPOSEWORKLOAD function (Algorithm ??), as shown in Examples ?? and ?. Thus, $\|\mathbb{P}\|_1$ simply represents the maximum number of overlapping RCQs in $\mathbb{P}$. Overlapping RCQs on the tree $\mathcal{T}$ must occur on the same path, that is, they form an ancestor-descendant relationship, since the children of each node n have non-overlapping ranges. Thus, the maximum number of overlapping RCQs across all tree paths, is equal to $\|\mathbb{P}\|_1$. $\square$

Our proactive strategy generation function, GENERATEPROACTIVESTRATEGY, and its helper function SEARCHPROACTIVENODES, are presented in Algorithm ?. The latter function conducts a recursive top-down traversal of the strategy decomposition tree (Line ??), and outputs a list of nodes to be fetched proactively into $\Delta\mathbb{P}$, such that the sum of the number of marked and proactively fetched nodes for each path in the tree is at most $\|\mathbb{P}\|_1$.

The first argument to SEARCHPROACTIVENODES represents the current node v , and the second argument r represents the number of *remaining* nodes that can be fetched proactively for the subtree originating at node v . Thus in the first call to this function, we pass the root node for the first argument and the second argument is initially set to $\|\mathbb{P}\|_1$ (Line ??). We decrement r whenever we encounter a marked node (Line ??) or when we add a node to the proactive output list (Line ??). We only add a node to the proactive output list if it is not cached and that the subtree norm of the node, $\mathcal{S}_{\mathbb{P}}(v)$, is less than r (Line ??). This condition ensures that the number of marked and proactively fetched nodes for the current path of the tree is less than r , and thus, we satisfy Equation (??) for each path of the tree, as we discuss below. We only continue iterating over the children of v if at least one node can still be fetched proactively for the current path of the tree (Line ??).

Lemma 2.5.3. The proactive strategy $\Delta\mathbb{P}$ generated by GENERATEPROACTIVESTRATEGY for an input $\mathbb{P}$ satisfies the condition:

$$\forall \text{ paths } p \in \mathcal{T}, \sum_{v \in p} \mathcal{M}_{\mathbb{P} \cup \Delta\mathbb{P}}(v) \leq \mathcal{S}_{\mathbb{P}}(\mathcal{T}.\text{root}) = \|\mathbb{P}\|_1 \quad (2.19)$$

Proof. This inequality holds for $\Delta\mathbb{P} = \emptyset$ by applying Lemma ?? and Definition ?. GENERATEPROACTIVESTRATEGY only adds a node to $\Delta\mathbb{P}$ if it satisfies the condition on line ??, namely that the remaining path has a length less than r . At the root, this condition is set to be less than $\|\mathbb{P}\|_1$. $\square$

Theorem 2.5.2. Given a paid strategy matrix $\mathbb{P}$, Algorithm ?? outputs $\Delta\mathbb{P}$ such that $\|\mathbb{P} \cup \Delta\mathbb{P}\|_1 = \|\mathbb{P}\|_1$.

Proof. Applying Lemma ?? and Definition ?? for the matrix $\mathbb{P} \cup \Delta\mathbb{P}$:

$$\|\mathbb{P} \cup \Delta\mathbb{P}\|_1 = \mathcal{S}_{\mathbb{P} \cup \Delta\mathbb{P}}(\mathcal{T}.\text{root}) = \max_{p \in \text{subtree}(v)} \sum_{v \in p} \mathcal{M}_{\mathbb{P} \cup \Delta\mathbb{P}}(v) \quad (2.20)$$

Rephrasing Lemma ??:

$$\max_{p \in \text{subtree}(v)} \sum_{v \in p} \mathcal{M}_{\mathbb{P} \cup \Delta\mathbb{P}}(v) = \|\mathbb{P}\|_1 \quad (2.21)$$

Using equations ?? and ??, we get:

$$\|\mathbb{P} \cup \Delta\mathbb{P}\|_1 = \|\mathbb{P}\|_1 \quad (2.22)$$

□

Example 2.5.5. In Figure ??, we apply our proactive strategy generation function to $\mathbb{P}_2 = \{\mathbb{A}_{[2,4]}^*, \mathbb{A}_{[3,4]}^*\}$ for $\mathbb{W}_2$ in our example sequence. We annotate each node with the values of r and its subtree norm $\mathcal{S}_{\mathbb{P}}(v)$, which we denote by s . The tree nodes $\mathbb{A}_{[4,8]}^*$, $\mathbb{A}_{[0,2]}^*$, $\mathbb{A}_{[0,1]}^*$, $\mathbb{A}_{[1,2]}^*$, $\mathbb{A}_{[2,3]}^*$ and $\mathbb{A}_{[7,8]}^*$ satisfy the condition $r > \mathcal{S}_{\mathbb{P}}(v)$. $\Delta\mathbb{P}_2$ consists of all of these nodes other than $\mathbb{A}_{[7,8]}^*$, which is excluded since it had already been proactively cached due to $\mathbb{A}_1$ for $\mathbb{W}_1$. Note that $\Delta\mathbb{P}_2$ has overlapping query predicates. For example, $\mathbb{A}_{[0,2]}^*$ and $\mathbb{A}_{[0,1]}^*$ overlap. Nevertheless, we ensure that $\mathcal{S}_{\mathbb{P}_2}(\mathcal{T}) = \mathcal{S}_{\mathbb{P}_2 \cup \Delta\mathbb{P}_2}(\mathcal{T}) = 2$.

Integration. The GENERATEPROACTIVESTRATEGY function calls the SEARCHPROACTIVENODES on the root of the tree, as described earlier (Line ??). Once this function outputs $\Delta\mathbb{P}$, it is transformed to the corresponding full-rank form $\Delta\mathbf{P}$ through the FRT module (Line ??).

Importantly, since we do not use the noisy responses to the proactively fetched queries to answer the analyst's workload, we do not incorporate $\Delta\mathbb{P}$ in estimating $b_{\mathbb{P}}$ in our ESTIMATEPRIVACYBUDGET implementation of MMM (Algorithm ??). Since we ensure that $\|\mathbb{P} \cup \Delta\mathbb{P}\|_1 = \|\mathbb{P}\|_1$, we do not need to pay any more for $\mathbb{P}$ than computed in that function (Algorithm ??, Line ??). Consequently, the MMM ANSWERWORKLOAD function perturbs $\Delta\mathbb{P}$ with the same noise parameter as for $\mathbb{P}$, namely $b_{\mathbb{P}}$, to obtain the noisy responses $\tilde{\mathbf{y}}_{\Delta\mathbb{P}}$ (Algorithm ??, Line ??). It then updates the cache with these noisy responses, along with those for $\mathbb{P}$ (Algorithm ??, Line ??). The PQ module can also be used while the SE module is turned on. Algorithm ?? can also be applied to multi-attribute strategies, as we discuss in Section ??.

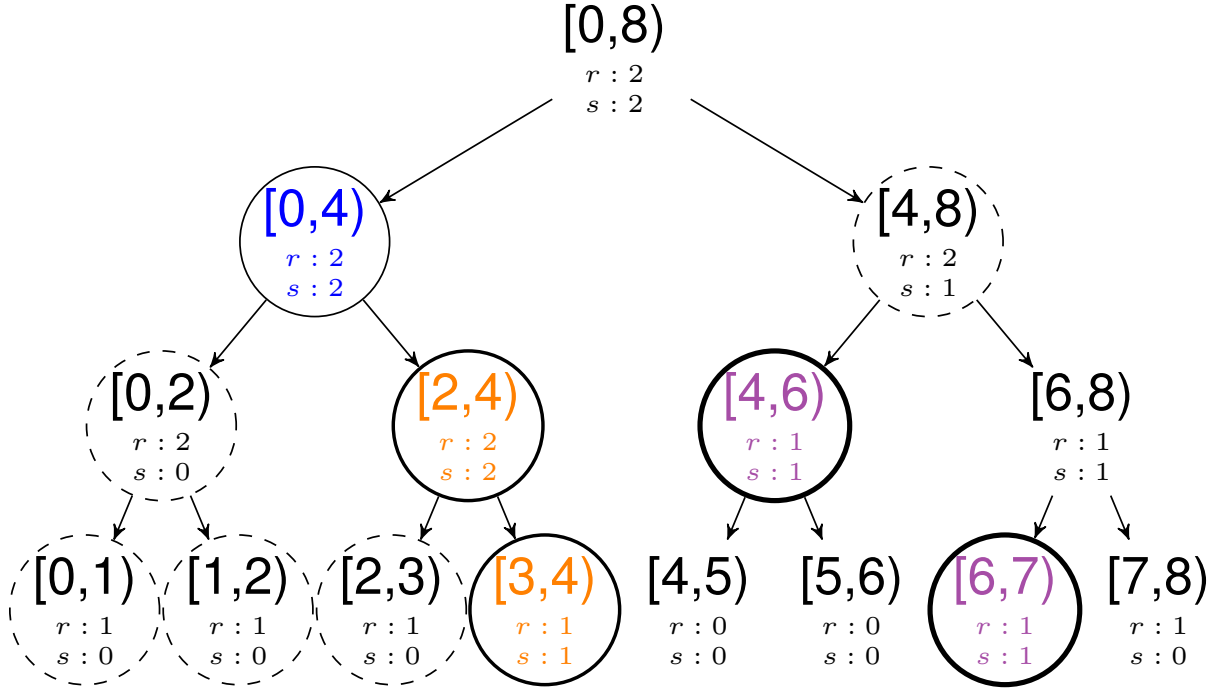


FIGURE 2.6: Example of the SEARCHPROACTIVENODES function (??) for an integer domain $[0, 8)$. Workload queries include $\mathbb{W}_1 = \{[0, 7)\}$, $\mathbb{W}_2 = \{[2, 6), [3, 7)\}$. Nodes present only in strategy $\mathbb{A}_1$ are shown in blue text, nodes only in $\mathbb{A}_2$ are in orange text, nodes in both $\mathbb{A}_1$ and $\mathbb{A}_2$ are shown in violet text. The dashed nodes are output by the Proactive Querying (PQ) module (Section ??), for $\mathbb{A}_2$. The annotated value of r refers to Algorithm ??, while the annotated value of s represents $\mathcal{S}_{\mathbb{P}}(v)$ (Definition ??) in that algorithm.

2.6 Relax Privacy Mechanism

When exploring a database, a data analyst may first ask a series of workloads at a low accuracy (spending ϵ_1), and then re-query the most interesting workloads at a higher accuracy (spending $\epsilon_2 > \epsilon_1$). (The repeated workload can also be asked by a different analyst.) The cumulative privacy budget spent by the MMM will be $\epsilon_1 + \epsilon_2$ due to sequential composition. The goal of the Relax Privacy module is to spend less privacy budget than MMM on such repeated workloads with higher accuracy requirements (or equivalently, relaxed privacy requirements).

Koufogiannis et al. [KHP16] refine a noisy response at a smaller ϵ_1 , to a more accurate response at a larger ϵ_2 , using only a privacy cost of $\epsilon_2 - \epsilon_1$. However, their framework

Algorithm 8 RP: ESTIMATEPRIVACYBUDGET interface

```
1: function ESTIMATEPRIVACYBUDGET( $\mathcal{C}, \mathbf{A}, \mathbf{W}, \alpha, \beta$ )
2:    $\nabla$  Group strategy queries in  $\mathcal{C}$  (originally defined in ??) by timestamp.
3:    $\mathcal{C} \leftarrow \{(\mathbb{A}_1, \tilde{\mathbf{y}}_1, b_1), (\mathbb{A}_2, \tilde{\mathbf{y}}_2, b_2), \dots (\mathbb{A}_t, \tilde{\mathbf{y}}_t, b_t)\}$ 
4:    $S_{RP} \leftarrow \{\mathbb{A}_j \in \mathcal{C} | \mathbb{A}_j \supseteq \mathbb{A}\}$   $\triangleright$  Set of all strategy matrices  $\mathbb{A}_j$  that contain  $\mathbb{A}$ 
5:   if  $S_{RP} = \emptyset$  then
6:     return RP cannot run for this input  $\mathbb{A}$ .
7:    $b \leftarrow \text{MMM.ESTIMATEPRIVACYBUDGET}(\mathcal{C} = \emptyset, \mathbf{A}, \mathbf{W}, \alpha, \beta)$ 
8:    $o = \arg \min_j \epsilon_{RP,j} = \frac{\|\mathbb{A}_j\|_1}{b} - \frac{\|\mathbb{A}_j\|_1}{b_j}$   $\triangleright \mathbb{A}_o$  has the lowest RP cost
9:   return  $\mathbb{A}_o, b_o, \epsilon_{RP,o}, b$ 
```

only operates with the simple Laplace mechanism. Thus, we achieve the aforementioned goal by closely integrating their framework with the matrix mechanism and our DP cache. This design of the RP module meets a secondary goal, namely, our RP module handles not only repeated analyst-supplied *workloads*, but also different workloads that result in identical strategy matrices. We exploit the fact that other modules, such as the PQ module, also operate over the matrix mechanism and DP cache. Therefore, our RP module can be seen as generalizing these frameworks to operate over more workload sequences.

2.6.1 Estimate Privacy Budget Interface

We first describe the ESTIMATEPRIVACYBUDGET interface for RP and present it in Algorithm ?. The privacy cost of the RP mechanism is defined as the difference between the new or *target* privacy budget for the output strategy noisy responses $\tilde{\mathbf{y}}$ to meet the accuracy guarantees, and the old or *cached* privacy budget (ϵ_c) that cached responses to $\mathbb{A}$ were obtained at. Then the main challenge in implementing this interface for the RP module is in choosing which past strategy entries should be updated by the RP mechanism, based on their RP cost as defined above.

Each strategy query $\mathbf{a} \in \mathbb{A}$ may be cached at a different timestamp. For simplicity, we design the RP module to update the entirety of a past strategy matrix at a single timestamp, rather than picking and choosing strategy entries across different timestamps. We remind the reader that a cache entry consists of a strategy query, its noisy response, its noise parameter and the timestamp at which this noisy response was fetched (Definition ?). Our RP cache lookup condition groups cache entries by their timestamps to form cached strategy matrices (Line ??), and identifies a set of candidate matrices that include the entire input strategy (Line ??). The inclusion condition (instead of an equal-

ity) allows proactively fetched strategy entries to be updated, at no additional cost to updating $\mathbb{A}_j$. If answering $\mathbb{A}$ using the cache requires either composing cache entries spanning multiple timestamps, or composing cache entries at one timestamp and paid (freshly noised) strategy queries at the current timestamp, then the RP cost estimation interface simply returns nothing (Line ??); CacheDP will instead use another module.

Example 2.6.1. Suppose that the workloads shown in Figure ?? have been asked in the past at α_1 , and have been answered through MMM, as discussed in Example ?. Now $\mathbb{W}_3 = \{[3, 8]\}$ is asked at $\alpha_3 < \alpha_1$. We have $\mathbb{A}_3 = \{[3, 4), [4, 8)\}$. Thus $\mathbb{A}_3 \subset \mathbb{P}_2 \cup \Delta\mathbb{P}_2$, and the RP module updates all of $\mathbb{A}_{3,RP} = \mathbb{P}_2 \cup \Delta\mathbb{P}_2$ from α_1 to α_3 . On the other hand, as an example of a strategy that is not answered by our RP module, consider $\mathbb{W}_3 = \{[3, 7]\}$ and its strategy $\mathbb{A}_3 = \{[3, 4), [4, 6), [6, 7)\}$. While $[3, 6) \in \mathbb{P}_2$ and it was cached at $t = 2$ at $b = b_2$, we find that $[4, 6), [6, 7) \in \mathbb{P}_1$ and were cached at $t = 1$ at $b = b_1$. So, $\mathbb{A}_3$ is not answered by our RP module.

For each candidate cached strategy matrix $\mathbb{A}_j$, we now need to compute the RP cost, namely $\epsilon_{RP,j}$, to update its cached noisy response vector $\tilde{\mathbf{y}}_j$ from one that is sampled at b_j to one that is sampled at the new target b . The target noise parameter is that required by the cacheless MM to achieve an (α, β) -accuracy guarantee for $\mathbf{W}, \mathbf{A}$. It can be obtained by running the ESTIMATEPRIVACYBUDGET implementation of MMM with an empty cache (Line ??). Importantly, we only seek to obtain accuracy guarantees over $\mathbf{WA}^+$, and not over $\mathbf{WA}_o^+$, and thus we compute the target noise parameter b based on the input strategy matrix $\mathbb{A}$, rather than considering the cached strategy matrices. Lastly, the RP module chooses to update the candidate cached strategy $\mathbb{A}_j$ with the minimum RP cost (Line ??). We return the optimal cached strategy matrix $\mathbb{A}_o$, its cached noise parameter b_o , its RP cost $\epsilon_{RP,o}$ and the target noise parameter b .

2.6.2 Answer Workload Interface

Our implementation of the ANSWERWORKLOAD interface for the RP module is shown in Algorithm ?? and is a straightforward application of Koufogiannis et al.’s noise down module. We recall the aforementioned parameters returned by the ESTIMATEPRIVACYBUDGET implementation: $\mathbb{A}_o, b_o, \epsilon_{RP,o}, b$ (Line ??). We first fetch the noisy responses to $\mathbb{A}_o$ from the cache (Line ??). We then compute the Laplace noise vector used in the past $\boldsymbol{\eta}_o$, by subtracting the ground truth for the cached strategy from the cached noisy response (Line ??). We can now supply Koufogiannis et al.’s noise down algorithm with the old noise vector $\boldsymbol{\eta}_o$, the cached noise parameter b_o , and the target noise parameter b . This algorithm draws noise from a correlated noise distribution [KHP16, Algorithm 1], and

Algorithm 9 RP: ANSWERWORKLOAD interface

```

1: function ANSWERWORKLOAD( $\mathcal{C}, \mathbf{A}, \mathbf{W}, \mathbb{x}$ )
2:   Parameters: Workload sequence number ( $i$ )
3:    $\mathbb{A}_o, b_o, \epsilon_{RP,o}, b \leftarrow \text{RP.ESTIMATEPRIVACYBUDGET}(\mathcal{C}, \mathbf{A}, \mathbf{W}, \alpha, \beta)$ 
4:    $\tilde{\mathbf{y}}_o \leftarrow [c.\tilde{y} \mid c \in \mathcal{C}, c.a \in \mathbb{A}_o]$   $\triangleright$  Get noisy responses for  $\mathbb{A}_o$  from the cache.
5:    $\boldsymbol{\eta}_o \leftarrow \tilde{\mathbf{y}}_o - \mathbb{A}_o \mathbb{x}$   $\triangleright$  Old noise vector for  $\mathbb{A}_o$ .
6:    $\boldsymbol{\eta} \leftarrow \text{LAPNOISEDOWN}(\boldsymbol{\eta}_o, b_o, b)$   $\triangleright$  Koufogiannis et al. [KHP16]
7:    $\tilde{\mathbf{y}} \leftarrow \mathbb{A}_o \mathbb{x} + \boldsymbol{\eta}$   $\triangleright$  Compute new noisy responses to  $\mathbb{A}_o$ .
8:    $\mathcal{C} \leftarrow \mathcal{C} \cup (\mathbb{A}_o, b, \tilde{\mathbf{y}}, t = i)$   $\triangleright$  Update the cache with new noisy responses to  $\mathbb{A}_o$ .
9:    $\tilde{\mathbf{y}}' \leftarrow \tilde{\mathbf{y}}$  for  $\mathbb{A} \subseteq \mathbb{A}_o$   $\triangleright$  Retain only new noisy responses to  $\mathbb{A}$  to answer  $\mathbb{W}$ , dropping noisy
      responses to  $a \in \mathbb{A}_o \cap a \notin \mathbb{A}$ .
10:   $\tilde{\mathbf{z}} \leftarrow \mathbf{W}\mathbf{A}^+ \tilde{\mathbf{y}}'$   $\triangleright$  Compute workload response.
11:  return  $\tilde{\mathbf{z}}$ 

```

outputs a new, more accurate noise vector $\tilde{\mathbf{y}}$ (Line ??). We can simply compute the new noisy response vector to $\tilde{\mathbf{y}}_o$ using the ground truth and the new noise vector (Line ??). We then update the cache with the new, more accurate noisy responses, which can be used to answer future strategy queries (Line ??). Finally, we do not need the noisy strategy responses to $\mathbb{A}_o - \mathbb{A}$ to answer the analyst's workload, and so we filter them out to simply obtain new noisy responses $\tilde{\mathbf{y}}'$ to $\mathbb{A}$ (Line ??). We use $\tilde{\mathbf{y}}'$ to compute the workload response (Line ??), just as we did for MMM, and return it to the analyst.

2.7 Multiple Attribute Workloads

We describe how CacheDP can operate over queries with multiple attributes. We define a single data vector $\mathbb{x}$ over $\text{dom}(\mathcal{R})$ as the cross product of d single-attribute domain vectors. It represents the frequency of records for each value of a marginal over all attributes. However, $|\mathbb{x}|$ and thus $|\mathcal{C}|$ could be very large due to the cross product.

We observe that not all attributes may be referenced by analysts in their workloads. Suppose that each workload includes marginals over a set of attributes $S_{\mathcal{A}} \in \mathcal{R}$. That is, each marginal $\mathbb{w} \in \mathbb{W}$ includes $|S_{\mathcal{A}}| = k \leq d$ RCQs, with one RCQ over each attribute ($\mathbb{w} = \prod_j^k \mathbb{w}_j$). These workloads would share a common, smaller domain and hence a smaller data vector $\mathbb{x}_{S_{\mathcal{A}}} = \bigotimes_{i=1}^{k \leq d} \mathbb{x}_{A_i}$. Similarly, instead of creating a large cache, we create a set of smaller caches, with one cache $\mathcal{C}_{S_{\mathcal{A}}}$ for each *unique* combination of attributes $S_{\mathcal{A}}$ encountered in a workload *sequence*. Cache entries can thus be reused across workloads that span the same set of attributes. The entries of each smaller cache $\mathcal{C}_{S_{\mathcal{A}}}$ are indexed by its associated domain vector $\mathbb{x}_{S_{\mathcal{A}}}$.

Example 2.7.1. Consider three attributes with $\text{dom}(\mathcal{A}_1) = [0, 4)$, $\text{dom}(\mathcal{A}_2) = [0, 8)$, and $\text{dom}(\mathcal{A}_3) = [0, 2)$. The analyst supplies a workload sequence of RCQ over different sets of attributes: $\mathbb{W}_1$ over $S_1 = \{\mathcal{A}_1, \mathcal{A}_2\}$, $\mathbb{W}_2$ over $S_2 = \{\mathcal{A}_2, \mathcal{A}_3\}$ and then $\mathbb{W}_3$ over $S_3 = S_1 = \{\mathcal{A}_1, \mathcal{A}_2\}$. Then $\mathbb{W}_1$ and $\mathbb{W}_3$ are answered using the same domain vector $\mathbb{x}_{S_1} = \mathbb{x}_{\mathcal{A}_1} \otimes \mathbb{x}_{\mathcal{A}_2}$ and a cache indexed over it, $\mathcal{C}_{S_1}$. However, $\mathbb{W}_2$ is answered using a different domain vector $\mathbb{x}_{S_2} = \mathbb{x}_{\mathcal{A}_2} \otimes \mathbb{x}_{\mathcal{A}_3}$ and the corresponding cache $\mathcal{C}_{S_2}$.

Our cache-aware MMM, SE and RP modules can be extended trivially to the multi-attribute case, since these modules would simply operate on the larger domain vector. However, in order to generate $\mathbb{A}$, the ST module relies on a k -ary strategy tree, corresponding to $\mathbb{A}^*$ for the single-attribute case. Thus, we extend the ST and PQ modules by defining this global strategy tree using marginals over multiple attributes. Our extended ST and PQ modules serve as a proof-of-concept that other modules can be extended for other problem domains.

For a given workload $\mathbb{W}$ that spans a set of attributes $S_{\mathcal{A}}$, we can use the single-attribute strategy tree $\mathcal{T}_i$ for each attribute $\mathcal{A}_i \in S_{\mathcal{A}}$ to construct a multi-attribute strategy tree $\mathcal{T}$ for $\mathbb{W}$, as follows. Intuitively, $\mathbb{x}$ consists of marginals that are constructed by taking each element in $\text{dom}(\mathcal{A}_i)$, and forming a cross product with each element in the $\text{dom}(\mathcal{A}_{i+1})$, and so on. Elements in $\text{dom}(\mathcal{A}_i)$ are represented by leaf nodes on its strategy tree $\mathcal{T}_i$, as illustrated in Figure ?? . So, for example, we form a two-attribute strategy tree $\mathcal{T}$ over $\mathcal{A}_1$ and $\mathcal{A}_2$, by attaching the tree $\mathcal{T}_2$ to each *leaf* node for the tree $\mathcal{T}_1$. Nodes on $\mathcal{T}$ either represent marginals over these attributes or sums of marginals. In particular, the leaf nodes on $\mathcal{T}$ represent elements of the marginals over $\mathcal{A}_1$ and $\mathcal{A}_2$. We can then simply extend this definition to $k = \|S_{\mathcal{A}}\|$ attributes in $\mathbb{W}$.

We can thus decompose each workload marginal $\mathbb{w}$ in terms of nodes on this tree $\mathcal{T}$. In the first step, we decompose each single-attribute marginal $\mathbb{w}_i$ following our DECOMPOSEWORKLOAD function (Algorithm ??) to get a set of single-attribute strategy nodes $\mathbb{A}_i$. The second step is run for all but the last (k th) attribute: in this step, we *further* decompose $\mathbb{A}_i$ in terms of the leaf nodes on tree $\mathcal{T}_i$, to obtain $\mathbb{A}_{\text{leaf},i}$. In the final step, we compute the cross-product of the single-attribute strategy nodes $\mathbb{A}_{\text{leaf},i}$, over all i , and a final cross product with the decomposition of the k th attribute, $\mathbb{A}_k$, to form the strategy marginal $\mathbb{a}$ for the workload marginal $\mathbb{w}$, as follows: $\mathbb{a} = \bigotimes_{i=1}^{k-1} \mathbb{A}_{\text{leaf},i} \otimes \mathbb{A}_k$.

Example 2.7.2. Consider the first workload in the sequence in Example ?? . It consists of one marginal tuple over attributes $\mathcal{A}_1$ and $\mathcal{A}_2$: $(\mathbb{W} = [0, 3)_{\mathcal{A}_1}, [1, 6)_{\mathcal{A}_2})$. We denote the global strategy matrix for attribute i as $\mathbb{A}i^*$. We decompose the RCQ for each attribute in the marginal, using our single attribute strategy generator. Thus: $\mathbb{A}_{\mathcal{A}_1} = \{\mathbb{A}1_{[0,2)}^*, \mathbb{A}1_{[2,3)}^*\}$

and $\mathbb{A}_{\mathcal{A}2} = \{\mathbb{A}2_{[1,2]}^*, \mathbb{A}2_{[2,4]}^*, \mathbb{A}2_{[4,6]}^*\}$. We then construct the two-attribute strategy marginals as the cross product of these RCQs: $\mathbb{A} = \mathbb{A}_{\mathcal{A}1} \otimes \mathbb{A}_{\mathcal{A}2} = \{(\mathbb{A}1_{[0,2]}^*, \mathbb{A}2_{[1,2]}^*), (\mathbb{A}1_{[0,2]}^*, \mathbb{A}2_{[2,4]}^*), (\mathbb{A}1_{[0,2]}^*, \mathbb{A}2_{[4,6]}^*), (\mathbb{A}1_{[2,3]}^*, \mathbb{A}2_{[1,2]}^*), (\mathbb{A}1_{[2,3]}^*, \mathbb{A}2_{[2,4]}^*), (\mathbb{A}1_{[0,2]}^*, \mathbb{A}2_{[4,6]}^*)\}$

We design the above strategy transformer such that our PQ module can easily be applied to the structure. The only difference is that the GENERATEPROACTIVESTRATEGY algorithm is run over the strategy tree for the combination of attributes in the workload ($S_{\mathcal{A}}$) rather than the global tree structure. The PQ module can thus add queries for any subset of attributes in $S_{\mathcal{A}}$, prioritizing nodes higher up in the tree, which correspond to marginals over fewer number of attributes.

2.8 Evaluation

We conduct a thorough experimental evaluation of CacheDP. We describe our experimental setup in ???. To demonstrate the effectiveness of our PET, we evaluate the cumulative privacy budget of interactive workload sequences in ???. We assess how often each module is used, and quantify its impact on the privacy budget, through our ablation study in ???. We then demonstrate the efficiency of our PET by evaluating its storage and computation overheads across the aforementioned workload sequences in ??.

2.8.1 Experimental Setup

Baseline Solutions. We consider a number of baseline, accuracy-aware solutions from the literature to compare with CacheDP. We implement an initial prototype of CacheDP in Python. We make our full evaluation publicly available at <https://git.uwaterloo.ca/m2mazmud/cachedp-public.git>.

APEx [GHIM19]: APEx is a state-of-the-art accuracy-aware interactive DP query engine. We use the source code provided by Ge et al. to evaluate APEx.² APEx consumes accuracy requirements in the form of an (α, β) bound (see Definition ??). To run APEx on workloads with a variance-based accuracy requirement, we use a tail bound on the Laplace distribution to derive values for α and β . APEx treats all workload queries separately and has no cache of previous responses.

APEx with cache: We simulate APEx with a naive cache of all past workloads and their responses. If a client repeats a workload asked in the past by any client, with the same

²<https://github.com/cgebest/APEx>

TABLE 2.2: Datasets, their sizes, and associated tasks, with the attributes or number of attributes used in each task.

Dataset	ADULT [Koh96]	TAXI [Com22]	PLANES [Tra22a; EZBK20]
Size	48842×14	1028527×19	$500,000 \times 12$
Tasks (Attributes)	BFS (Age) DFS (Country)	BFS (Lat, Long) DFS (Lat, Long)	IDEBench (8 out of 12)

or a lower accuracy requirement, we do not count its privacy budget towards the cumulative budget spent by APEx with cache.

Pioneer [PYZ+13]: Pioneer is a DP query engine that incorporates a cache of previous noisy responses to save the privacy budget on future queries. Since the authors of Pioneer did not publish their code, we implement it from scratch in Python, based on the paper. We implement their simple composite plan using all hyperparameters as described in the paper. As we demonstrate in this section, our implementation reproduces similar performance to the results given in the original paper [PYZ+13, Section 7.2.1]. Pioneer expects the accuracy over its responses in the form of a target variance. To run Pioneer on workloads with an (α, β) requirement, we use an MC simulation to search for the variance that satisfies the accuracy requirement. Since Pioneer can only answer single range queries, we decompose all workloads into single queries for our evaluation, and let Pioneer answer them sequentially.

Datasets and Tasks. We use three datasets: the 1994 US Census data in the ADULT dataset [Koh96] ($48842 \text{ rows} \times 14 \text{ attributes}$), a log of NYC yellow taxi trip records from 2015 in the TAXI dataset [Com22] (1028527×19), and US domestic flight records in the PLANES dataset [Tra22a; EZBK20] ($500,000 \times 12$). The TAXI dataset is used to model a single strategy tree over a pair of correlated (*Lat, Long*) attributes. Each node on the tree (or a range query) represents a rectangle of area on a map, and each level on the tree splits each node’s rectangle into its quarters. We outline the datasets used for each task in ??.

BFS and DFS tasks: A common data exploration task involves traversing a decomposition tree over the domain [ZXX16], by progressively asking more fine-grained queries over a subset of the domain. In each iteration, the analyst decomposes each query in the past workload whose noisy response satisfies a certain criteria, into k new children queries on the attribute decomposition tree. In the BFS task, the analyst explores only past queries with a sufficiently *high* noisy count. The BFS task thus returns the smallest subsets of a domain that are sufficiently populated.

In contrast, in a DFS task, the analyst focuses on past queries with a sufficiently *low non-zero* count (i.e., underrepresented subgroups). A DFS task terminates if a query’s noisy count is non-zero and falls within the low DFS threshold range. In the DFS task, when the analyst reaches a leaf node without finding a node with a sufficiently low count, they backtrack a random number of steps up the tree, and resume the search with the second smallest node at that level. We group the range queries that satisfy the BFS or DFS criteria, into a single workload per level of the tree.

Randomized range queries (RRQ) over synthetic data: We replicate the evaluation of Pioneer [PYZ+13] through 50,000 randomized range queries over a synthetic dataset. We fix a domain size of $m = 1,000$ such that each range query is contained in $(0, m)$. Each query is in the form of $(s, s + \ell)$ where s and ℓ are both selected from a normal distribution with the following mean (μ) and standard deviation (σ): $\mu_s = 500, \sigma_s = 10, \mu_\ell = 320$, and $\sigma_\ell = 10$. The accuracy requirement is supplied as an expected square error, which is also selected from a normal distribution with $\mu = 250000, \sigma = 25000$.

IDEBench for Multi-Attribute queries: Eichman et al. develop a benchmarking tool to evaluate interactive data exploration systems [EZBK20]. We use this tool to construct a sequence of exploration workloads for a multi-attribute case study over the PLANES dataset. Specifically, we extract the SQL workloads for their $1 : N$ case where each query triggers an additional N dependent queries [EZBK20]. We simplify these queries for easy integration with our prototype.

Client Modelling. We model multiple data analysts querying the aforementioned systems, as clients. We schedule the clients’ interactions with each system by randomly sampling clients, with replacement, from the set of clients until no queries remain. A client chooses the accuracy requirements and task parameters for each run of the experiment independently and at random.

Experiment setup. We run the BFS and DFS tasks with $c = 25$ clients and the IDEBench task with $c = 10$ clients. (We only run the RRQ task with a single client, to precisely replicate the Pioneer paper. Since Pioneer can only work with a single attribute, we do not run it for tasks over the TAXI or PLANES datasets.) We outline the task parameters in ?? . The BFS and DFS tasks are conducted over the *Age* and *Country* attributes of the ADULT dataset respectively. Both tasks are also run over the *Latitude (Lat)* and *Longitude (Long)* attributes of the TAXI dataset. Each client randomly draws the minimum threshold for their BFS and the maximum threshold for their DFS. The BFS, DFS and IDEBench tasks consume (α, β) accuracy requirements, with fixed $\beta = 0.05$ across all

clients. For a database D , each client randomly selects $\alpha = \alpha_s |D|$ for $\alpha_s \in [0.01, 0.16]$, with step size 0.05.

2.8.2 Privacy Budget Evaluation

We begin with evaluating the total privacy budget consumed by our system and baseline solutions across the aforementioned tasks. We repeat each interactive exploration task $N = 100$ times, and we compute the average cumulative privacy budget for our solution CacheDP and baselines (APEX, APEX with cache, Pioneer) over all N experiment runs. We plot the mean and 95% confidence intervals in Figure ?? . We have two hypotheses:

H1 The baselines arranged in order of increasing cumulative privacy budget should be: APEX, APEX with cache, Pioneer.

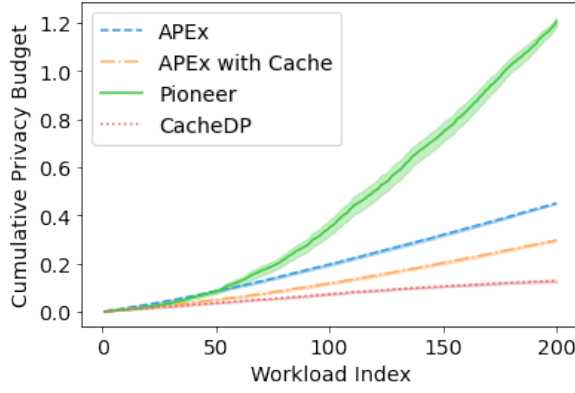
H1.1 Pioneer should outperform APEX with cache, since Pioneer saves privacy budget over any related workloads, whereas APEX with cache only saves privacy budget over repeated workloads.

H1.2 Baselines with a cache (APEX with cache, Pioneer) should outperform the baseline without a cache (APEX).

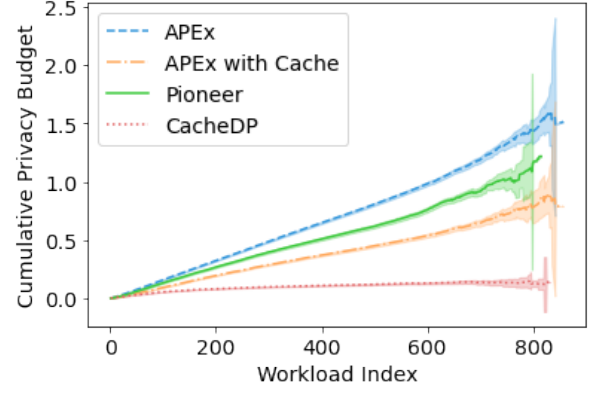
H2 CacheDP should outperform all baselines.

First, we observe that hypothesis ?? holds for all tasks, other than the single-attribute BFS and DFS tasks (Figures ??, ??). Since we decompose each workload into multiple single range queries for Pioneer, this sequential composition causes it to perform worse than APEX without a cache in the BFS task, and thus hypothesis ?? is violated. For the same reason, APEX with cache outperforms Pioneer for the single-attribute DFS task, and so, hypothesis ?? is violated. Though, we note that hypothesis ?? holds for the RRQ task (Figure ??). Our Pioneer implementation replicates a similar privacy budget trendline to the original paper [PYZ+13, Figure 15].

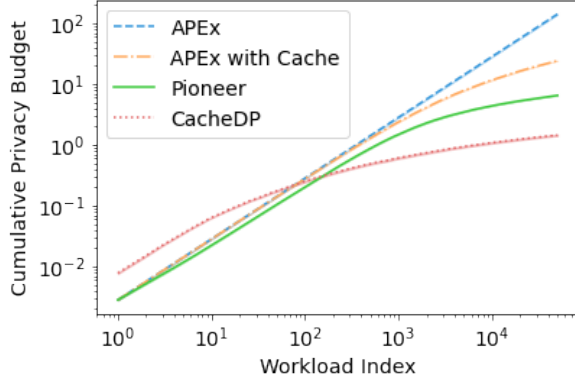
Hypothesis ?? holds for all tasks, and the cumulative privacy budget spent by CacheDP scales slower per query, over all graphs. We note that in the RRQ task (Figure ??), CacheDP spends more privacy budget *upfront* than the other systems, since these systems use the simpler Laplace Mechanism, which is optimal for single range queries over our underlying Matrix Mechanism. However, any upfront privacy budget spent by CacheDP is used to fill the cache, which yields budget savings over a large number of workloads, as CacheDP requires an order of magnitude less cumulative



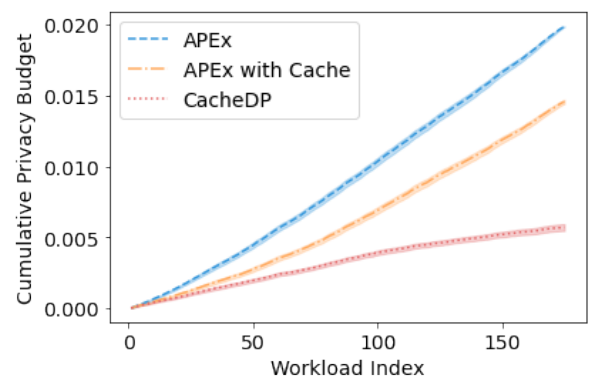
(A) BFS - Adult (Age attribute)



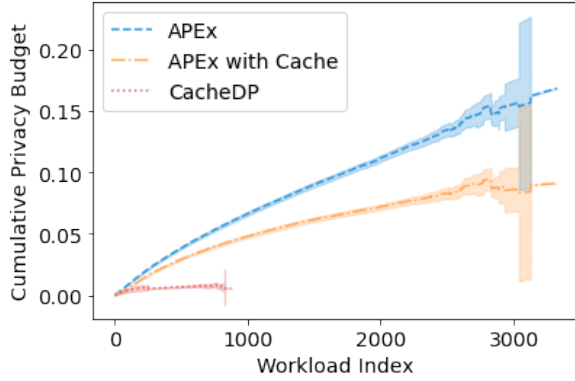
(B) DFS - Adult (Age attribute)



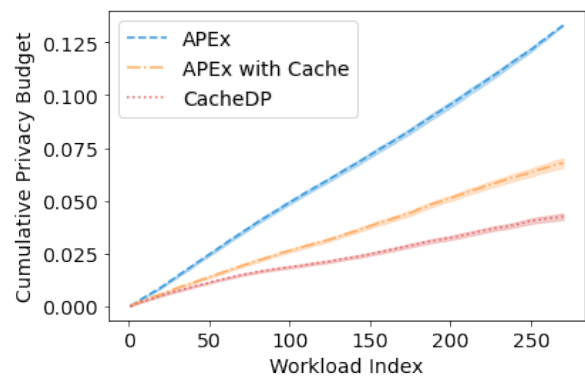
(C) RRQ - Synthetic (1D)



(D) BFS - Taxi (Lat, Long attributes)



(E) DFS - Taxi (Lat, Long attributes)



(F) IDEBench - Planes (multiple attributes)

FIGURE 2.7: Average cumulative privacy budget comparison between CacheDP and baselines (APEX, APEX with cache, Pioneer).

privacy budget than the best baseline (Pioneer). We observe that even in the computationally intensive tasks due to larger data vectors for two attributes (Figures ??, ??) and multiple attributes (Figure ??), CacheDP outperforms the best baseline (APEX with cache), by at least a factor of 1.5 for Figure ??.

For both DFS tasks (Figures ??, ??), since each experiment can terminate after a different number of workloads have been run, we observe large confidence intervals for higher workload indices for each system. CacheDP simply returns cached responses to a workload if they meet the accuracy requirements, whereas our simulation for APEX with cache resamples noisy workload responses and may traverse the tree again in a possibly different path. One client may be satisfied with looser accuracy requirements than another client; loosened accuracy requirements can lead to frequent re-use of our cache (Section ??), and thus, we find that in Figure ??, CacheDP ends the DFS exploration faster than APEX with cache.

2.8.3 Ablation Study

Each of our modules contribute differently to the success of our system across different workloads. We conduct an ablation study in two parts analyzing our modules. First, we analyze the frequencies at which each module is selected to answer a workload, and second, we run a study to quantify the impact of each module on the cumulative privacy budget. We begin with our frequency analysis, noting that a module is chosen to answer a given workload if it is estimated to cost the lowest privacy budget. We only include the MMM, RP, and SE modules in this analysis, since the PQ module is not involved in the cost estimation stage. We present the number of times each module is chosen to answer a workload in each of the BFS, DFS, RRQ and IDEBench tasks, averaged over $N = 100$ runs, in Table ?. If MMM reports $\epsilon = 0$ for a workload, CacheDP simply uses MMM to answer the workload using cached responses, and it does not run RP or SE modules. We thus separately record the number of free workloads per task in the first row of Table ?. First, we observe that most workloads for each task are free, indicating that using solely the cached strategy responses, CacheDP can successfully answer most workloads for these tasks.

Second, considering all non-free workloads, each of the modules are used the most frequently for at least one task. SE is chosen most frequently for the BFS and DFS tasks, answering 52% and 64% of non-free workloads respectively. Furthermore, for many workloads in these tasks, we observed that MMM had $\epsilon > 0$ cost, but under SE, these workloads became free ($\epsilon = 0$). Consequently, the SE module was chosen over MMM and RP. RP is chosen most frequently for the IDEBench task (57%), whereas MMM is

TABLE 2.3: Average number of times each module in CacheDP was chosen for each task; most frequently chosen modules are in bold.

	BFS	DFS	RRQ	IDEBench
Free	164.8 ± 0.9	591 ± 6	49801 ± 3	216 ± 1
MMM	2.1 ± 0.1	2.25 ± 0.09	137 ± 3	15.1 ± 0.2
RP	14.7 ± 0.5	21.1 ± 0.5	1.4 ± 0.1	31 ± 1
SE	18.5 ± 0.6	42 ± 1	60 ± 4	7.8 ± 0.2

used most frequently for the RRQ task (69%). Thus, we can see that each module plays a role in CacheDP’s performance in one or more tasks.

We also quantify savings in the cumulative privacy budget due to each module. We rerun our single-attribute tasks (BFS, DFS, RRQ) while disabling each of our modules (MMM, SE, RP, PQ) one at a time, and present the cumulative privacy budget consumed by each such configuration in Figure ?? . The standard configuration consists of all modules turned on. Turning an effective module off should lead to an increase in the cumulative privacy budget, in comparison to the standard configuration. First, we observe that the standard configuration performs the best in all three tasks, while considering CI overlaps. Therefore, data analysts need not pick which modules should be turned on in order to answer a workload sequence with the lowest privacy budget. Second, the PQ module significantly lowers the cost for the BFS and RRQ task, proactively fetching all (≈ 12) queries in a BFS workload at the cost of one strategy node. Third, the RP module also lowers the cost for BFS, when the same workload is repeated by other clients.

Fourth, turning the SE module off only contributes to minor differences in the cumulative privacy budget (B_c). However, the reader may expect that turning the SE module off would lead to a higher B_c , since based on the frequency analysis, the SE module is most frequently chosen to answer non-zero workloads for the BFS and DFS tasks.

We reconcile this discrepancy as follows. When the SE module is chosen, constrained inference reduces the privacy cost for paid strategy queries. As a result, the cache entries for these queries remain at a lower accuracy than if MMM or RP had been used to answer them, and they may not be reusable for later workloads. Thus CacheDP may need to obtain noisy responses for these queries, later on, at a higher accuracy. The SE module thus provides savings on earlier workloads at the cost of a less accurate cache to answer future workloads. In summary, different tasks exploit different modules and the standard configuration incurs the least privacy budget, and thus data analysts need not turn off any modules.

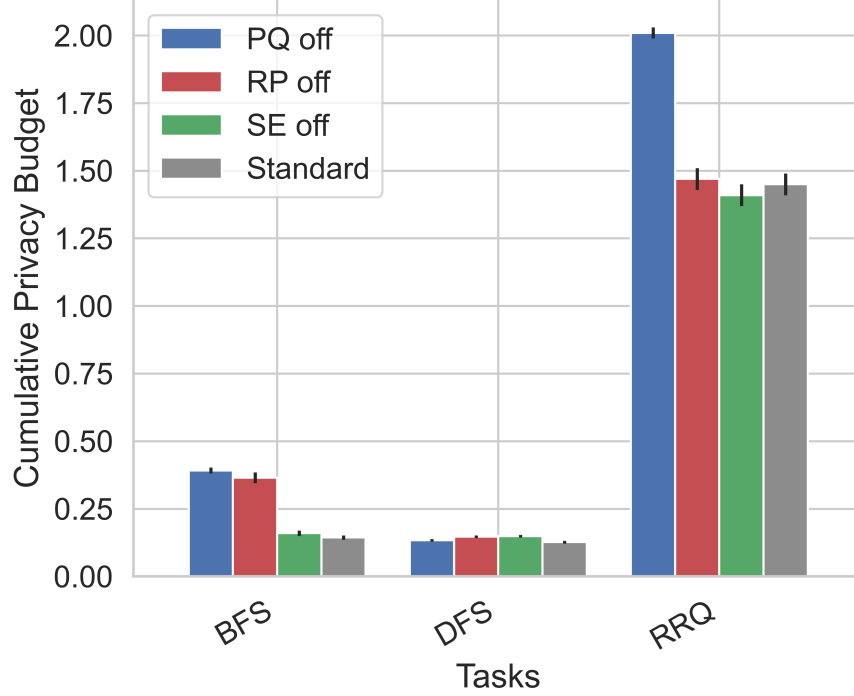


FIGURE 2.8: Ablation study over PQ, RP, SE modules of CacheDP. Note that each bar, except for the grey bar, corresponds to one module, out of the PQ, RP and SE modules, being turned *off*, while the others remain on. The cumulative privacy budget spent across all single-attribute tasks (BFS, DFS, RRQ) is averaged over $N = 100$ runs; standard deviation bars are also shown in black.

2.8.4 Overhead Evaluation

We compute the following storage and computation overheads for all systems, averaged over all $N = 100$ experiment runs: (1) *cache size* in terms of total number of cache entries at the end of a run, and (2) *workload runtime*, averaged over all workloads in a run. In Table ??, we present these overheads for representative tasks. (Since our simulation for APEx with cache only differs from APEx by a recalculation of the privacy budget (Section ??), the latter has the same runtime as the former. We thus only depict the overheads for APEx with cache, Pioneer and CacheDP.) We also note that the RRQ task only consists of 10 workloads, rather than 50,000 as noted earlier in ??.

Regarding workload runtimes, first, we observe that APEx with cache is around two orders of magnitude faster for the RRQ task than for the BFS and DFS tasks. Empiri-

TABLE 2.4: Comparison of CacheDP and baselines, in terms of cache size, which is denoted by *size* and measured in number of cache entries, and workload runtime, which is denoted by *time*. We note that *BFS (1D)* and *DFS (1D)* refer to the single-attribute BFS and DFS tasks over the Adult dataset. We refer to the APEx with Cache baseline by *APEx (C)*. We report the mean and the standard deviation for both the cache size and the workload runtime.

System	RRQ		BFS (1D)		DFS (1D)	
	Size	Time (ms)	Size	Time (ms)	Size	Time (ms)
APEx (C)	9.9±0.9	5±1	70±8	390±40	14±3	270±30
Pioneer	9.9±0.9	0.57±0.06	42±8	1.2±0.2	14±3	0.60±0.06
CacheDP	1100±400	4000±600	147±3	3200±200	60±10	3700±700

cally, we found that for all single range queries, APEx uses the Laplace Mechanism, and applies a theoretical bound [GHIM19, Appendix A.1] to compute the noise parameter. In contrast, for the BFS and DFS tasks, APEx uses the Matrix Mechanism and executes a Monte Carlo (MC) simulation to tightly bound the noise parameter for these tasks. Thus, we observe that APEx with cache is faster for the RRQ task since it does not use an MC simulation for this task. Second, we also find that the MC simulation is a bottleneck for CacheDP. We observe that for all tasks, once the cache starts filling up after a few initial queries, CacheDP runs the SE module as well, and thus runs the MC simulation at least twice for later queries. When the preconditions for using the RP module are also met (Line ??, Algorithm ??), which is relatively infrequent, CacheDP also runs the MC simulation (Line ??, Algorithm ??) and this contributes to the increased workload runtime.

Third, we acknowledge that Pioneer is roughly three orders of magnitude faster than CacheDP for the single-attribute BFS and DFS tasks. We remind the reader that Pioneer spends an order of magnitude worse privacy budget on these tasks (Figures ??, ??). We argue that the larger computation overhead is a reasonable tradeoff for CacheDP’s privacy budget savings. We note that in absolute terms, CacheDP only takes a few seconds per workload for all single-attribute tasks; we discuss ways to reduce the runtime of CacheDP below. We also analyze how the cache size and runtime of CacheDP scales with more workloads for the multi-attribute IDEBench task below.

Regarding cache sizes, we find that our PQ module inserts strategy query nodes into the cache, and significantly increases our cache size over the baselines. We conclude that at most CacheDP incurs up to a three orders of magnitude storage overhead in terms of its cache size over baselines (Pioneer, APEx with Cache). Nevertheless, since our cache entries only consist of 4 floating points (32 B), even a cache with $\approx 25k$ entries would be

TABLE 2.5: Indices of attributes included in each workload in the IDEBench task for the overhead evaluation (Figure ??). A total of eight attributes were spanned by all 22 queries in that task. We note that queries 12–15 correspond to a histogram query that uses the same attribute for the histogram bins as the value whose frequency count is reported, and is thus interpreted as a two-attribute workload.

Workload number	1–2	3	4	5–11	12–15	16–17	18	19	20	21	22
Indices of attributes in workload	1	2	3	1,2	1,1	1,3	4,5	5,6	5,7	6,8	1,4,5

reasonably small in size (≈ 800 kB). We claim that these overheads are reasonable, given the significant reduction in privacy budget over the baselines. We note that our cache size is limited by the number of nodes on our strategy tree. Thus, when the analysts continue querying each system, we expect that our cache size will be bounded, whereas in other baselines, the cache size would keep increasing with more workloads.

We now analyze how the cache size and runtime of CacheDP scales for multi-attribute queries. This experiment includes a total of 22 IDEBench SQL workloads; each workload query spans between one and three attributes and all 22 workloads span a total of eight attributes. We number these eight attributes based on the chronological order in which they appear in the workloads. For a given workload index, we list the indices of the attributes that are covered in that workload in ?. We consider multiple clients in Table ?, whereas we only consider one client in this experiment. We focus on the MMM module for this experiment and turn off our SE and PQ modules. We note that increasing the number of records in the database only impacts the size of the data vector in the pre-processing stage, and does not impact the size of the cache. We remind the reader that in a multi-attribute query, each marginal $w \in \mathbb{W}$ includes k RCQs, with one RCQ over each attribute (?).

We illustrate how the cache size and runtime of the MMM in CacheDP scales with the workload index in ?. We present the following hypotheses:

H5 Our cache size scales linearly with the number of *new* marginals introduced in a given query.

H6 Our runtime scales linearly with the number of elements in $\mathbf{WA}^+$.

We can verify Hypothesis ? from Figure ? and Table ?. The cache size increases significantly when the marginals in a workload span different sets of attributes than those encountered previously, such as on workloads 5 as well as on workloads 17–22.

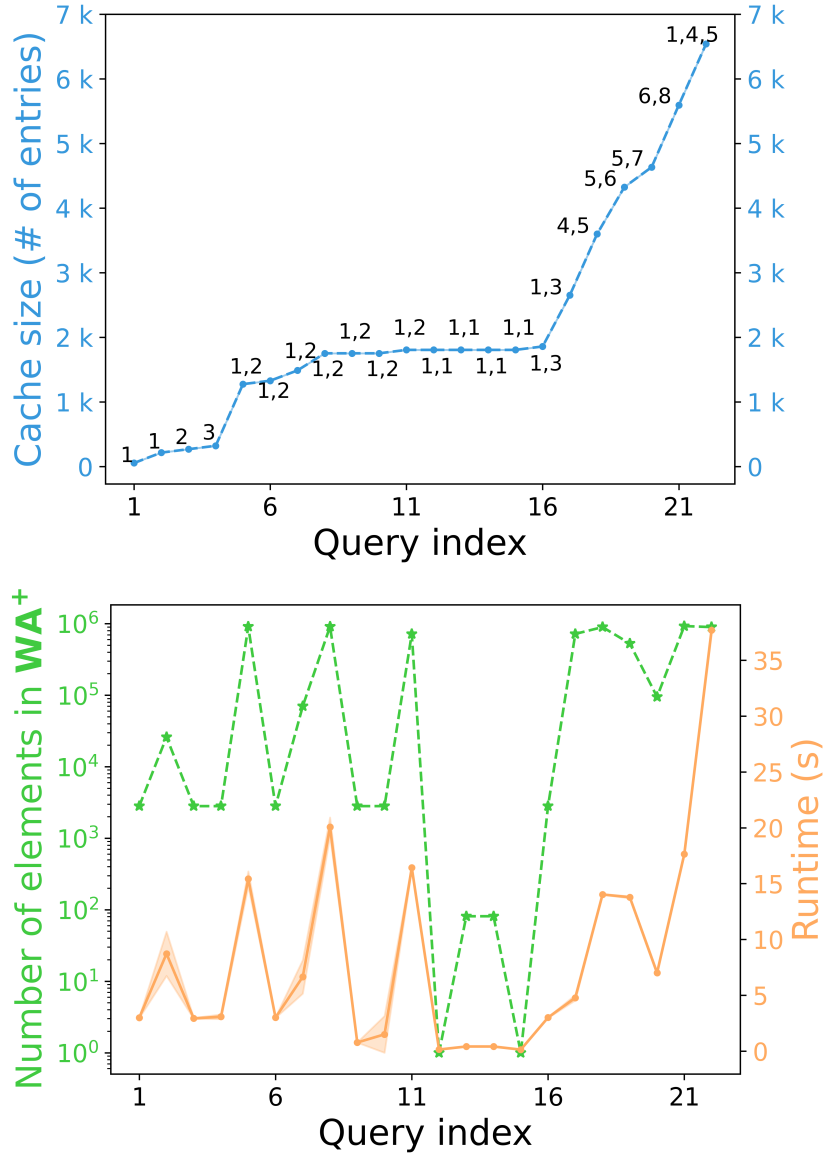


FIGURE 2.9: Cumulative cache size (blue) and average runtime (orange) of CacheDP, versus the workload index, for the IDEBench task. A total of eight attributes were spanned by all 22 queries in that task and each workload contains between one to three attributes. Each cache size is annotated with the indices of the attributes in the workload; Table ?? also presents the information within the annotations. 95% confidence intervals for the runtimes are computed across $N = 10$ runs and shown in light orange. The number of elements in the WA^+ matrix for each workload is shown in green.

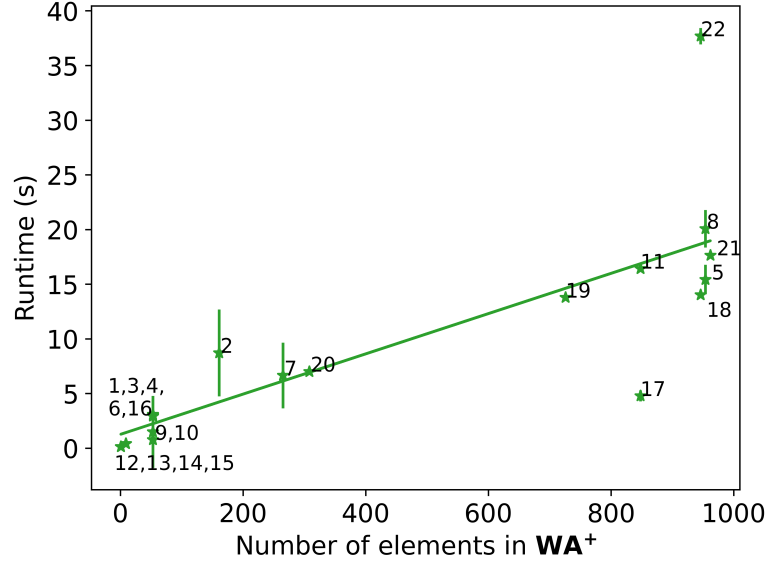


FIGURE 2.10: Line of best fit to measure the dependence of the runtime of the MMM with $\mathbf{WA}^+$. The line of best fit is shown in green. Each cache size is annotated with the indices of the attributes in the workload; Table ?? also presents the information within the annotations. 95% confidence intervals for the runtimes are computed across $N = 10$ runs and shown in green.

For example, although workloads 16 and 17 span the same sets of attributes, the later workload filters for a different set of values over the domain of one of the attributes. Concretely, the cache size can be limited by identifying inaccurate cache entries that can be replaced, while inserting new noisy responses, in each module.

Considering Hypothesis ??, we present the line of best fit between the runtime and the number of elements in $\mathbf{WA}^+$ in Figure ?. We can conclude from Figure ?? and Figure ?? that while the runtime of the MMM scales approximately linearly with the number of elements in $\mathbf{WA}^+$, there are other factors that contribute to an increased runtime, such as for the last workload, which has three attributes. We leave further analysis of these factors to future work. Concretely, we again find that the MC simulation forms the bottleneck for the runtime, when the number of attributes in the query remains small, whereas the matrix multiplication becomes the main bottleneck as the number of attributes in the query increases. The MC simulation can be avoided entirely, by expressing the desired accuracy through expected variances instead of the (α, β) requirements. Our implementation can be optimized with fast matrix multiplication algorithms to achieve smaller runtimes. We reiterate that the SE and PQ modules were

turned off for this experiment. The PQ module would further increase both the cache size and the runtime of CacheDP; instead of running the PQ module while answering a workload, we can delay fetching further strategy queries until the system is idle. The SE module would also increase the runtime of CacheDP; instead of setting the maximum number of cached strategy queries that can be added by the SE module (λ , Line ?? in Algorithm ??) to a constant, it can be a function of the size of the strategy matrix $\mathbb{A}$. We leave these optimizations to future work.

2.9 Related Work

Constrained inference techniques have been applied in the non-interactive DP setting to improve the accuracy of noisy query responses [ZXX16; QYL13] and in synthetic data generators [TWB+21; GMHI21; MSM19] to infer consistent answers from a data model built through noisy measurement queries. However, these systems do not provide any accuracy guarantee on the inferred responses. If the analyst desires a more accurate response than the synthetic data can offer, no privacy budget remains to improve the query answer [TMH+21]. Our work applies DP constrained inference in an interactive setting so that we can spend the privacy budget on queries that the analyst is interested in and meet their accuracy requirements. On the other hand, existing accuracy-aware DP systems for data exploration [GHIM19; MTS+12], releasing data [GHK+16; NBH+22], or programming frameworks [VRG19; XDW+21] do not exploit historical query answers to save privacy budget on a given query. We design a cache structure and inference engine extending one of these accuracy-aware systems, APEx [GHIM19].

Peng et al.’s Pioneer [PYZ+13] uses historical query answers to obtain accurate responses to upcoming single range queries. However, CacheDP can handle workloads with multiple queries. Second, it supports multiple, complex DP mechanisms and chooses the mechanism that uses the least privacy budget for each new workload. Third, our PQ module (Section ??) proactively fetches certain query responses that can be used later at no additional cost. Finally, CacheDP can answer multi-attribute queries through our extended ST module (Section ??).

Kostopoulou et al. developed Turbo [KTC+23] concurrently with our work and it was published shortly thereafter. Both Turbo and CacheDP are accuracy-aware and cache-aware interactive DP engines and share many similarities. Independently, both systems arrive at a tree-based cache structure. While in CacheDP each node of the tree contains a noisy response to the range query for specified values for one attribute (or a set of k attributes), in Turbo, each node of the tree contains a degree $d - 1$ histogram of

noisy responses to the range query for a vertical partition of a table with d attributes, over a specific value for an attribute.

Both deal with the problem of developing a differentially private cache that outperforms a naïve cache, albeit in different ways. Kostopoulou et al. extend the Private Multiplicative Weights algorithm (PMW) with a caching layer. The PMW algorithm maintains the aforementioned noisy histograms, starting from a uniform distribution to a differentially private version that is sufficiently accurate with respect to the ground truth. It determines, using the Sparse Vector technique, whether the noisy histogram is sufficiently accurate with respect to the ground truth,³ and if so, it is used directly to answer the incoming query, for free. Otherwise, privacy budget is spent for two tasks: to obtain the differentially private noisy response using the Laplace or Gaussian mechanism, and to adjust the weights of the distribution. Importantly, the second task can consume privacy budget that is multiple orders of magnitude more than that of the first task. Consequently, “training” the PMW algorithm in early stages is expensive, and thus, a naïve cache can easily outperform the PMW algorithm for small numbers of queries. Turbo opportunistically bypasses this algorithm as follows. It determines whether the SV test is likely to fail, and if so, directly fetches the DP noise and updates the histograms, so that in the future, the SV test may succeed with the updated histograms.

Both CacheDP and Turbo also exploit the DP parallel composition theorem over their respective cache structures. While Turbo arranges its tree-based cache such that it can fetch histograms for non-overlapping partitions, it does not *proactively* fetch noisy histograms for other non-overlapping partitions for free. On the other hand, the PQ module proactively fetches noisy responses to non-overlapping values of sets of attributes. As such, Turbo handles a larger class of workloads than CacheDP. For example, Turbo handles streaming workloads through time-based partitioning.

We conclude that while Turbo and CacheDP differ significantly in their designs, they nevertheless use similar design strategies to address similar challenges, such as extending existing mechanisms (MM or PMW) to outperform a naïve cache, structuring the cache and applying the parallel composition theorem over the cache. This shows that the techniques used in CacheDP can be generalized and extended to design better DP caches.

³Note that this is different from our CHECKACCURACY function, which simply checks whether the noise in the workload response meets the (α, β) -accuracy requirements, *without* consulting the ground truth.

2.10 Conclusion

In relation to privacy design strategies in ??, through our DP system, we realize the ABSTRACT design strategy. We address concerns of our users for this context, namely data analysts and data admins. CacheDP supports data analysts in answering data exploration workloads accurately, without requiring them to know DP. On the other hand, for data admins, CacheDP realizes privacy budget savings that were commonly seen only in the non-interactive DP model. We show that composing PETs, such as differential privacy, with components of the back-end web architecture, such as caches, involves non-trivial challenges. Our work provides researchers with a methodology to address common challenges while integrating DP mechanisms with a DP cache, such as, cache-aware privacy budget estimation (MMM), filling the cache at a low privacy budget (PQ), and maximizing cache reuse (SE).

While relational DBMSes remain widely used today as a back-end web architecture there is growing interest in peer-to-peer networks and distributed web applications built atop them, and we thus proceed to examine these back-end web architectures next.

Chapter 3

DHTPIR

In light of our thesis statement (??), we continue to examine privacy-enhancing technologies that realize privacy design strategies within back-end web architectures. We focus on peer-to-peer networks as our back-end web architecture in this and the following chapters. Structured peer-to-peer (P2P) networks support redundancy and do not suffer from a single point of failure. Even though individual nodes can store content cheaply, structured P2P networks remain relevant today, for the two aforementioned reasons, and are used to develop distributed file systems, such as IPFS [HSE19]. Structured P2P networks link peer nodes' identifiers to the content that they store and use distributed hash tables (DHTs) for routing search and insertion queries. DHTs, such as Kademlia [MM02] and Chord [SMK+01], are analogous to conventional hash tables, with the exception that the entire key-value store is systematically split across nodes in the P2P network, such that queries can be conducted efficiently.

Censorship-resistant systems allow users access to online content when direct access to such content is restricted by a nation-state adversary, namely a *censor*. For instance, Tor [DMS04] is an anonymity network that supports users in accessing websites that are censored by nation-states. Censorship-resistant *publishing* systems allow publishers to submit their work onto multiple servers, also called *nodes*, such that a censor cannot take down or tamper with the published content. Several censorship-resistant publishing systems have been proposed to date, such as Vasserman et al.'s CROPS [VHKH11], Waldman and Mazières' Tangler [WM01] and Stubblefield and Wallich's Dagster [SW02]. Censorship-resistant publishing systems are commonly built atop structured peer-to-peer networks, due to their built-in redundancy and lack of a single point of failure.

Node operators within the censor’s region of influence may be legally obliged to report any prohibited content that is stored on their machines, or that is requested by users in queries. Many node operators would thus rather not learn the content that they are storing or the content of the queries that they are receiving. For instance, the Signal messaging app is designed to minimize the types of users’ data and metadata that are stored in the Signal servers, so that obliging with legislative requests for users’ data can only cause minimal privacy harms to its users [Sig24; Sig21; Sig16]. The aforementioned systems support node operators in *plausibly denying* any knowledge of the content that they store. They do so by encrypting chunks of publishers’ documents and storing key material separately from these chunks, thereby realizing Hoepman’s SEPARATE privacy design strategy. However, these systems reveal the content of a user’s query.

In this work, we address the problem of hiding the content of a user’s query, by integrating a PET, namely PIR, into a back-end web architecture, namely peer-to-peer networks, as we outline below. We realize the MINIMIZE strategy in that we minimize the information available to node operators about a peer’s query. We do so by separating this information across multiple peers, such that a quorum of them need to collude in order to learn what was queried, and thus, we also realize the SEPARATE strategy.

3.1 Introduction

Censorship-resistant publishing systems have largely focused on securing the *supply* of documents for users. Frequently, the goal is to ensure that for each document, at least one server is available to provide any user a copy of that document. As long as node operators learn what document was requested, a censor may coerce them into revealing this information. Consequently, node operators may be discouraged from contributing their storage and bandwidth for such systems. Users who recognize this flaw may not allow themselves to even seek documents — a form of self-censorship known as a *chilling effect*. In these cases, censorship-resistant publishing systems have still failed to make the document available to all users, because they did not secure the *demand* of documents. Thus, exposing the contents of queries to node operators disincentivizes both node operators and users from using these types of systems.

Private information retrieval (PIR) is designed to solve the problem of retrieving a row from a database on a server, without revealing the index of that row to the server. Information-theoretic PIR (IT-PIR) in particular typically requires multiple replicas of the database. The user sends different queries to each server holding a replica, such that as long as less than a threshold fraction of the servers collude, they do not learn

anything about the contents of the query, and the user can reconstruct the desired row from the servers' responses [Gol07; CKGS98].

To motivate our key insights on using PIR within censorship-resistant publishing systems, we examine the structure of these systems. P2P networks have been known to suffer from attacks by malicious users who aim to prevent a legitimate user from obtaining a correct copy of their desired content [LNR06; SNDW06]. Related research [FSY05; BM07; AS09] shows that if only a small fraction of nodes in the P2P network are malicious, it is possible to group joining peers into *quorums* such that, with high probability, all quorums in the network will only contain a bounded fraction of malicious nodes. This result is known as the *goodness invariant*. Young et al. [YKGK10] develop protocols for efficient, robust communication across quorums that prevent spamming attacks, whereas Backes et al. [BGKT12] extend the former's protocols to support conducting DHT queries without revealing the query content to any in-path nodes.

Our key insight lies in observing that IT-PIR can be instantiated over quorums in the DHT. Carefully analyzing and setting the parameters involved, we leverage the DHT goodness invariant to satisfy the IT-PIR non-collusion assumption. We present an IT-PIR architecture that can be integrated with an existing censorship-resistant publishing system, such as CROPS, to enable clients to retrieve documents from the DHT privately. Specifically, neither the target node, nor any in-path nodes, learn which document was retrieved. In addition to our enhancement of using PIR to protect the nodes from learning which document is being retrieved, we use Young et al.'s protocols for robust communication across quorums, and Backes et al.'s protocols to hide the content of routing queries from intermediate nodes.

We begin with a description of the building blocks we use, namely PIR, DHTs, and censorship-resistant publishing systems, in ???. We discuss robust DHTs, including related work in query privacy over DHTs, in ??, and describe our threat model in ??. We then present the following contributions:

- We provide an architecture, similar to that of regular DHTs in structured P2P networks, such that a user can retrieve a document from the network without revealing what document was retrieved to the target node that stores the document, nor to any in-path nodes. We also illustrate an integration of this interface with that of an existing censorship-resistant publishing system (??).
- We detail the security parameters of our system and show in our analysis that our system works when the DHT quorums have less than $\frac{1}{4}$ malicious nodes (??).

- We include a message and communication complexity analysis of our system and find that it has a low communication complexity in comparison to the best-known scheme for privately retrieving a document from a node in a DHT [BGKT12] (??).
- We simulate the communication complexity of our system and find that it is explained by our complexity analysis. We also simulate the latency and throughput overheads of our system, to demonstrate its viability for deployment (??).

Finally, though we have presented our interface in the context of a censorship-resistant publishing system, we discuss general deployment of our interface in ??, while highlighting other use cases and addressing bootstrapping issues. We conclude in ??.

3.2 Background and Related Work

In this section, we describe the building blocks used in our system. We begin with a brief description of private information retrieval and factors affecting our choice of a PIR scheme. Second, we provide a brief overview of DHTs, including existing literature on attacks in DHTs. Third, we delve into censorship-resistant publishing systems, with a focus on plausible deniability, and motivate our interface, which prevents exposing node operators to the content of queries.

3.2.1 Private Information Retrieval

Private information retrieval (PIR) allows public databases, held by one or more untrusted servers, to be queried by clients while guaranteeing that the servers cannot learn which index of the database was accessed. Computational PIR (CPIR) schemes require the computational hardness assumptions of certain cryptographic problems, and can be used even when only one server stores the entire database. On the other hand, information-theoretic PIR (IT-PIR) schemes do not require any computational hardness assumptions; however, they can only be implemented when multiple servers have copies of the database [CKGS98].

In IT-PIR schemes, the client constructs one query for each server such that using the responses to these queries, it can reconstruct the desired row of the database. A certain threshold number of these servers must not share the queries they receive with each other, as otherwise, they could reconstruct the desired row and identify it, just as the client does. This is known as the *non-collusion assumption*. Chor’s IT-PIR scheme [CKGS98] requires all nodes to respond correctly to the client’s query, for the latter to be able to reconstruct the desired row, and thus does not tolerate any malicious

nodes. *Byzantine-robust* IT-PIR schemes, such as that of Goldberg [Gol07] support efficient reconstruction of the correct row, while allowing a fraction of servers to be offline and out of those that are online and provide responses, a fraction can provide incorrect responses. Thus, if one can reasonably argue that the aforementioned non-collusion assumption holds true, then IT-PIR schemes are preferable to CPIR schemes, since they do not require any computational hardness assumptions. We detail in ?? why the non-collusion assumption for IT-PIR can be reasonably made within our context, leading us to use IT-PIR. We use Goldberg’s IT-PIR scheme [Gol07] as it is robust against failing and Byzantine nodes, both of which exist in P2P networks.

3.2.2 Distributed Hash Tables

A file that is to be stored in a structured P2P network has a collision-resistant one-way hash, typically of its contents, as its identifier. A node on structured P2P networks also has a truncated one-way hash as its identifier. A common storage model for DHTs is that a node stores all files whose identifiers, when truncated, are equal to the identifier of the node. As no single entity stores all of the content on the network, peer-to-peer networks do not suffer from a single point of failure. Each node in a structured P2P network maintains a *routing table*, which consists of node identifiers and their routing information, such as network addresses, for a small number of other nodes in the DHT. In order to retrieve a file with a given identifier, a node would need to contact the target node that stores the file, based on the aforementioned storage model. The node would obtain routing information for the target node by first querying peers from its own routing table, which are known as its *neighbours*. The node would then continue with querying its neighbours’ neighbours and so on, until it finds the target node, through *iterative routing*. The node follows the same process to insert a file into the DHT. (*Recursive routing*, on the other hand, has the node’s neighbour itself looking up the next neighbour that is closest to the target node, and so on. We do not consider recursive routing as providing privacy guarantees over routing queries in recursive routing systems is fundamentally difficult, as Backes et al. [BGKT12] point out.) The maximum number of nodes that a given node needs to contact is known as the *path length* (ℓ). DHTs guarantee that the path length is logarithmic in the number of nodes in the network.

DHTs have been known to be vulnerable to many attacks that prevent a user from obtaining a copy of a file. Existing censorship-resistant publishing systems do not consider these attacks, and thus, do not integrate defenses that have been proposed against such attacks. A censor can easily exploit these attacks, which we motivate below, to break the availability of files stored on the overlay censorship-resistant publishing system.

If nodes are allowed to choose their own node identifiers, then a censor may simply join the system as a set of legitimate nodes, at various node addresses that would be storing prohibited documents. It can then simply drop all future requests that it obtains for storing or retrieving prohibited content. The censor can also direct its new nodes to join at addresses that are close to a target honest node, and cause this node's routing requests to be directed to its nodes, instead of other honest nodes. This is known as an *eclipse attack*, which restricts honest nodes' view of the network.

As a defence against these attacks, P2P networks can require existing nodes to assign a random identifier to incoming nodes. However, malicious nodes may still attempt to join and leave the network until they are placed at suitable network addresses, to influence honest nodes' routing table entries in a Byzantine join attack. Several researchers [BM07; FSY05] have proposed protocols to allocate joining nodes into quorums, such that by repeatedly rejoining, any malicious nodes do not increase their chances of being cast into a group with a majority of other such malicious nodes. We explicate Awerbuch and Schiedeler's cuckoo rule (CR) [AS09], as well as Sen and Freedman's improvement over this rule, namely the commensal cuckoo rule (CCR) [SF12], in ??.

To resolve routing requests across quorums, nodes in a quorum can reply back with the network addresses of the nearest quorum to the target address, akin to iterative routing in regular DHTs. However, malicious nodes in such quorums may respond back with incorrect routing table entries, or inundate honest nodes in other quorums with fake requests. Young et al. [YKGK10] present two robust communication protocols to support nodes in efficiently communicating across these quorums, while being able to either detect or prevent spamming attacks. However, these protocols only provide *integrity* guarantees over the routing information within a quorum-based DHT. They do not provide *confidentiality* of the routing query content; the node address being queried is known to nodes on the path to the target node and to the target node itself. The censor may thus compel all nodes in its region, as they may serve as *in-path* nodes to other nodes, to reveal the node addresses looked up by users in its region. It can then possibly harm these users for attempting to contact external nodes that serve prohibited content. Backes et al. [BGKT12] improve on Young et al.'s protocols by hiding the queried key from all in-path nodes, thereby providing query privacy (QP). As we discuss in the next subsection, honest node operators benefit from learning as little as possible about the key that was queried.

We describe Awerbuch and Schiedeler's CR strategy, Sen and Freedman's CCR strategy, Young et al.'s Robust Communication Protocols, and Backes et al.'s Query Privacy protocols in ??. Our interface uses these protocols to harden existing censorship-

resistant publishing systems against these attacks.

3.2.3 Censorship-resistant publishing

Censorship-resistant publishing (CRP) systems, such as Waldman and Mazières' Tangler [WM01], Waldman et al.'s Publius [WRC00], and Vasserman et al.'s CROPS [VHKH11], support publishing and retrieving documents in the face of a censor that may attempt to take down prohibited content. Censors may outright block any CRP systems within their administrative region. We assume that blocking the system entirely would impose an undesirably large social or economic cost to the censor. This is the collateral damage assumption, and we revisit it in ??.

CRP systems are built atop DHTs formed by machines that are located across different administrative regions, each of which contributes disk space and bandwidth. However, DHTs by themselves do not ensure the confidentiality, integrity, or availability of the content stored on them; CRPs include mechanisms to publish and retrieve documents while providing these three crucial properties. To increase the availability of the content stored on them, CRP systems store redundant copies of documents or of their parts across the underlying DHT. In Publius [WRC00], secret sharing is used to produce shards or *shares* of documents, such that the document can be reconstructed using a threshold number of shares. Publishing a single document in Tangler [WM01] necessitates replicating multiple other existing documents within the system, again through secret sharing. To store a document within CROPS, several replicas are created, and split into chunks using erasure coding, which is similar to secret sharing in terms of the aforementioned reconstruction guarantee. The lack of a single point of failure and support for redundant storage render DHTs attractive, with respect to the availability property, for censorship-resistant publishing.

All of the aforementioned systems encrypt documents to preserve their confidentiality, before breaking them down into shards for redundancy. Encryption supports node operators in *plausibly denying* knowledge of what content they store. Consider a censor that legally obliges all node operators in its region to report any prohibited content that they store. In Vasserman et al.'s CROPS, a node that stores an erasure-coded chunk of a document does not learn anything about it, based on the hardness assumption of the cryptographic hash used to index the chunk.

Importantly, CROPS helps users easily discover content on the system through keywords, while preserving plausible deniability over the stored content and ensuring its integrity. To publish a document, the publisher specifies relevant keywords for it. After encrypting the document and performing erasure coding on the ciphertext document,

the publisher creates replicas of two manifest files, namely the content and key manifests. A cryptographic hash of one important keyword, along with the replica number, is used to index the manifest replicas. Each manifest file is signed by the publisher to preserve its integrity. The content and key manifests separate the material required to locate the erasure-coded chunks (cryptographic hashes of the chunk content) from that required to obtain a plaintext document (symmetric key for decryption). (The manifests also contain cryptographic hashes to verify the integrity of the ciphertext document, which is reconstructed from erasure-coded shares, as well as of the plaintext document.) Nodes that store one of a key or a content manifest replica need to obtain the other one, and sufficient erasure-coded chunks, in order to obtain the plaintext document that the replicas refer to.

As the client’s query includes keyword hashes, the censor may legally oblige the node operator to reveal all queries that it received. The censor may then determine if any of the keyword hashes in a query match those of prohibited keywords [KCN+15; Hen15], using rainbow tables. It may also require the node operator to report the network addresses of all users, and possibly harm the users who attempted to fetch prohibited content. Thus, simply providing plausible deniability over *stored* content is insufficient; node operators should also be prevented from learning what content is being *retrieved*.

We address this problem by allowing a client to privately retrieve either erasure-coded chunks or manifests (collectively known as *files* from here on). Although we focus on CROPS as a use case for our system, we provide private file retrieval for censorship-resistant publishing systems in general. As we discuss in ??, our interface may also be used with other CRP systems and DHT-based applications.

3.3 Robust DHTs

In this section, we describe protocols to form quorums (Awerbuch and Schiedeler’s CR [AS09], Sen and Freedman’s CCR [SF12]), to route queries robustly across quorums (Young et al.’s RCP [YKGK10]), as well as to conduct these routing queries privately (Backes et al.’s QP [BGKT12]). The QP protocols can also be applied to conduct file retrieval queries privately, and so we also outline the advantages of our DHTPIR system over those.

Forming quorums: Awerbuch and Scheideler [AS09] propose a cuckoo rule joining strategy, wherein incoming nodes cause existing nodes at nearby addresses to be kicked

1CR. A joining node p is assigned a random address, which lies in some k -region of a quorum Q_T .

1CCR. Primary join: Along with step 1CR, quorum Q_T ensures that it has received at least $k - 1$ secondary joins before letting node p join.

2CR. All other nodes with addresses in this k -region are relocated to other random k -regions.

2CCR. Secondary join: Instead of step 2CR, $k' = k \cdot |Q_T| / \bar{s}$ nodes with addresses in the quorum Q_T are relocated to other random quorums (where $\bar{s}$ is the desired average quorum size).

FIGURE 3.1: Descriptions of cuckoo rule (CR) [AS09] and commensal cuckoo rule (CCR) [SF12].

out (or cuckooed) to other addresses, as shown in Figure ?? . A virtual address space of size n is split into *quorums*, which are split further into k -regions that span a fraction $\frac{k}{n}$ of the address space.

Awerbuch and Schiedeler show that for a given *global* bound on the ratio of malicious to honest nodes (say ϵ), their strategy results in approximately equally sized quorums of $s = O(\log n)$ nodes (the *balancing* condition), where the ratio of malicious to honest nodes in each quorum is upper-bounded by a value greater than ϵ (the *correctness* condition). However, their result is *asymptotic*, and so left an open question of whether it holds for networks with realistic numbers of peers.

We implemented a simulator for the cuckoo rule in Rust, and determined that for reasonable numbers of peers in the network (say 10 million), there is a severe tradeoff between ϵ and s : to maintain the correctness of relatively small quorums ($s < 100$), the network can only withstand a very small fraction of adversary-controlled nodes ($\epsilon < \frac{1}{1000}$). Having large quorums of 100 peers or more makes the distributed protocols required within Young et al.’s schemes expensive.

Sen and Freedman [SF12] performed a similar simulation and their findings echo ours. Additionally, they find that the aforementioned tradeoff worsens for large network sizes; that is, small quorums in larger networks can only withstand much smaller fractions of malicious nodes. To address this issue, they propose a *commensal cuckoo joining strategy*, which modifies each of the two aforementioned steps of the cuckoo rule, as shown in ??. In their joining strategy, the desired average quorum size is set beforehand (to $\bar{s}$), in contrast with other schemes that require it to increase with the network size. We use the commensal cuckoo rule strategy to form quorums in our join protocol. We

TABLE 3.1: Contrasting our scheme with existing ones for private and robust communication in a DHT. $\mathcal{D}$ is the total size (in bytes) of the files stored at each node in an s -node quorum. RR = robust routing; QP = In-path query privacy; PR = availability and, if so, communication complexity of private file retrieval.

System	RR	QP	PR
Base DHT	○	○	○
Young et al.’s RCP [YKGK10]	●	○	○
Backes et al.’s QP [BGKT12] w/o last hop	●	●	○
Backes et al.’s QP [BGKT12] w/ last hop	●	●	$\mathcal{D}$
DHTPIR (our system)	●	●	$2s\sqrt{\mathcal{D}}$

discuss its security implications within our threat model in ??.

Robust routing across quorums: Young et al. [YKGK10] propose two efficient robust communication protocols (RCP) that prevent spamming attacks from nodes in other quorums by using a threshold signature scheme. In order to communicate with a node in a target quorum, the client node must obtain a time-stamped proof of robust communication from one of the neighbours of that quorum, in the form of a threshold signature of the neighbouring quorum. Young et al.’s deterministic RCP-I scheme *prevents* spamming attacks by imposing a high communication cost for the client; the client communicates with each of the s nodes in each of ℓ quorums on the path to the target quorum, for $s \cdot \ell$ nodes in total. Their probabilistic RCP-II scheme enables the target quorum’s nodes to *detect* such an attack, and only requires the client to communicate with an expected $O(s + \ell)$ nodes. RCP schemes include distributed protocols for synchronizing the quorum’s cryptographic state whenever a new node joins the quorum.

Query privacy across quorums: Backes et al. [BGKT12] present two query privacy (QP) protocols, namely QP-I and QP-II, which build on Young et al.’s RCP-I and RCP-II protocols respectively to hide the node address key being queried from all nodes in all quorums in path to the target quorum. They develop a protocol based on oblivious transfer (OT) rather than PIR, in order to restrict a client from privately obtaining routing information from any intermediate quorum about *multiple* neighbouring quorums through a single valid query.

However, as they observe, their OT protocol requires the queried node to send the entry-wise encrypted data store over which the OT query is to be conducted back to the client. In their use case, this data store is simply the small routing table maintained at each DHT node, which contains $O(\log n)$ routing records for a network of size n . To

privately retrieve a file stored in the file store of the target quorum, Backes et al. suggest optionally extending this same OT protocol with one more hop to the target quorum. However, doing so would require sending the entire encrypted file store back to the client, resulting in a large communication complexity.

To retrieve a file from a quorum of s nodes, which holds a database of size $\mathcal{D}$ bytes, Backes et al.’s scheme incurs a communication complexity of $\mathcal{D}$, whereas our scheme only has a complexity of $2s\sqrt{\mathcal{D}}$. In ?? we contrast the properties provided by Young et al., Backes et al., and our schemes to DHT clients and compare the communication complexity for Backes et al.’s and our scheme in providing private file retrieval. We compute the range of values of $\mathcal{D}$ for which our system performs better in ??.

3.4 Threat model

As mentioned earlier, we build our system atop DHTs as they support redundant storage of documents and do not suffer from a single point of failure, which renders them suitable for censorship-resistant publishing. (Tor and Tor onion services are orthogonal to our work, as discussed in the non-goals below.) Our adversary is a single nation-state censor that has an administrative region of influence; other parts of the world are deemed safe. We consider the case of multiple such censors below. We assume that this nation-state cannot simply censor the entire system or host an alternative, as blocking the system entirely would impose an undesirably large social or economic cost to the censor. We recognize that this collateral-damage assumption may not hold for powerful nation-state adversaries: such an adversary could coerce its population to simply use a publishing system that is not privacy protective. However, we believe that it is a reasonable assumption that may hold for many nation-states. We require the same cryptographic hardness assumptions as Backes et al. [BGKT12], namely the GDH problem for threshold signatures and the DDH problem for the OT protocol.

Our adversary’s goals are to determine which document was retrieved in a particular content retrieval request and to prevent the operator from responding correctly to a request. Our adversary has several capabilities that it can exploit towards furthering its goals. First, the censor may observe the network traffic within its region of influence, possibly using deep packet inspection techniques on any plaintext data. Second, the censor can search for sensitive content by posing as a client and can map files to virtual addresses of nodes that store them. Third, the censor’s control over nodes in the network is bounded; in particular, for a network with n honest nodes, the censor may insert up to $\epsilon \cdot n$ malicious nodes, where $\epsilon \approx 0.02$. The censor may remove these nodes

from the network in the future, and have them rejoin the network, which would allow the node to join a different quorum. A censor could spawn more than $\epsilon \cdot n$ malicious nodes in a Sybil attack; deterring Sybil attacks through proof-of-storage mechanisms in IPFS is an ongoing line of work [GP25].

Multiple nation-state censors may be interested in censoring different content or content retrieval requests across nodes within their respective regions of influence. Since nodes operated by one nation-state censor may collude with those operated by another censor, we require that the ratio of nodes controlled by all these censors, to non-controlled nodes, is less than ϵ . These nodes may behave as Byzantine nodes by, for example, responding to routing requests with incorrect responses, in an attempt to misdirect a client’s routing request towards other Byzantine nodes. They may also collude and share the content retrieval requests that they obtain, or forward them to the censor. These censor-controlled nodes may also simply fail, by not replying to queries. Furthermore, they may attempt to inundate honest nodes with routing or content request messages at the application layer.

Finally, the censor cannot dynamically compromise existing nodes; that is, it cannot coerce honest nodes within its region of influence to act as Byzantine or failing nodes. It may not compel honest nodes to retain (and share) the content retrieval requests they receive. (Note that honest nodes in a given quorum would be geographically distributed in arbitrary administrative regions and may be out of the censor’s region of influence.) The censor may not block communication between honest nodes.

Our interface provides the following guarantees in the face of this adversary:

- Minimizing data within the content of document retrieval queries — *honest* server operator nodes and small coalitions of Byzantine nodes within a quorum do not learn which document was retrieved.
- Correct responses to document retrieval requests — a document retrieval request will be answered correctly, even if censor-controlled nodes provide incorrect or no responses.

Non-goals: We do not provide any additional protections over content *publishing* requests. Although we use RCP’s spam-limiting feature to prevent DoS amplification for the DHTPIR protocol messages, protecting nodes from general DoS attacks is orthogonal to this work. We do not aim to protect the client’s identity, and thus, we also do not prevent adversaries from targeting clients, such as by traffic analysis. Regarding

the first goal, although nodes controlled by the adversary that lie *within* the quorum from which the file is fetched also cannot learn which file was retrieved by colluding among themselves, they may collude with nodes from other quorums. By learning the set of quorums from which files were fetched, they may determine which document was fetched, in a quorum fingerprinting attack; we do not aim to protect against this attack, as DHTPIR aims to protect server nodes, and not directly clients.

DHTPIR may be instantiated over Tor onion services to provide client anonymity. However, Tor itself does not support the redundant storage of documents that is afforded by peer-to-peer networks like DHTs. Although Tor onion services may have multiple physical servers serving the same onion service, all servers must share a private key, putting them under shared administrative control. In this sense, Tor onion services suffer from a single point of failure, unlike DHTs. As node operators of Tor onion services learn which document was retrieved through a given document retrieval request, they do not have plausible deniability over the content of such requests. Tor exit node operators have been blamed, and even charged, for retrieving content that a client had retrieved through their node [Gal17]; privacy-enhancing technologies that prevent the operators from being exposed to the information they store or forward may decrease such risks.

3.5 Our Design

We first begin with an overview of our interface for integrating private file retrieval to DHT-based censorship-resistant publishing systems. We begin with a description of the building blocks that our interface requires, and then describe our implementation of this interface, while analyzing possible Byzantine behaviour. Through this analysis, we motivate the constraints that our instantiations of the building blocks must follow.

3.5.1 Building blocks

Our interface can be used within censorship-resistant publishing systems, such as Vasserman et al.’s CROPS [VHKH11], to insert files into quorums through the DHTPIR-PUTFILE function and to query quorums privately for files through the DHTPIR-FIND-FILE function, both provided by our DHTPIR API. Functions in the DHTPIR interface invoke existing protocols in layers, depicted in ???. A list of all API functions can be found for reference in ??.

We use Backes et al.’s query privacy (QP) protocol to privately fetch the desired quorum’s routing information; the corresponding function constitutes the QP API layer.

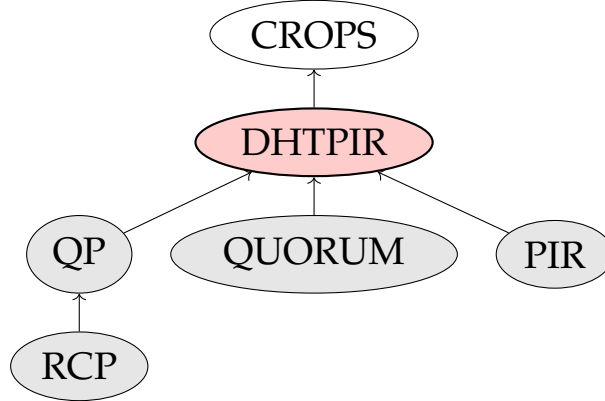


FIGURE 3.2: API interfaces required by, and provided by, our DHTPIR interface. The bold red ellipse represents our interface, and the grey ellipses show the layers of APIs required by our interface. The white ellipse represents an existing system within which our interface can be used in order to provide document retrieval privacy, in addition to the robust routing and in-path privacy provided by the lower layers.

(This protocol internally invokes Young et al.’s RCP schemes.) We provide a QUORUM API layer to support inter-quorum communication, given the target quorum’s routing information. We use Goldberg’s IT-PIR algorithm to privately fetch files from the target quorum; functions to generate, perform, and process PIR queries form the PIR API layer. We rely on Sen and Freedman’s commensal cuckoo-hashing technique [SF12] to establish quorums across the DHT; that is, to assign node addresses and quorum neighbours to new nodes joining it. We briefly describe functions at the QP, QUORUM, and PIR layers below.

QP layer: The client uses the QPGETTARGETQUORUM function to obtain the routing information of the target quorum while preserving the confidentiality of the file’s identifier from all nodes in the DHT. This function executes Backes et al.’s RCP-QP-I or RCP-QP-II algorithms for the input file identifier id . This function obtains the target quorums’ nodes’ network addresses and public encryption keys $[(A_1, PK_1), (A_2, PK_2), \dots, (A_s, PK_s)]$ from one of the quorum’s neighbours. It also obtains a proof ϕ of robust communication from that neighbour. The proof shows that the given client node sent a legitimate request to contact the target quorum, through a signature over the node’s network address (A), its public key (PK) and a timestamp ts (ϕ in our notation is equivalent to $S_{\ell-1}$ for a query path length ℓ in Backes et al.’s notation). Finally, this function returns a verification key VK , obtained from the target quorum’s neighbour, which allows the client to verify messages signed by a threshold number of nodes in the target quorum.

TABLE 3.2: Overview of the functions in each layer of the interfaces presented in Section ?? . The layer name is included as the first word of the function name.

Function(Arguments) → Output	Description
DHTPIRPUTFILE(File identifier id , File F) → None	<i>Privately</i> inserts a file F with the ID id . Replaces the native putFile function.
DHTPIRFINDFILE(File identifier id) → File F	<i>Privately</i> finds a file with the ID id . Replaces the native findFile function.
QPGETTARGETQUORUM(File identifier id) → ϕ , $[(A_1, PK_1), (A_2, PK_2), \dots (A_s, PK_s)], VK$	Obtain the following output for the closest quorum that stores the file with the ID id , without any intermediate node learning the ID: proof of robust communication: ϕ , target nodes' network addresses (A_i) and public keys (PK_i), quorum verification key: VK
QUORUMSENDQUERY(ϕ , $[(A_1, PK_1), (A_2, PK_2), \dots (A_s, PK_s)], [M_1 \dots M_s]$) → $[R_1 \dots R_s]$	Securely sends a message M_i to each of s peers in a target quorum using the proof of communication ϕ , the peer's address A_i , and public key PK_i . Receives a reply R_i from each peer.
QUORUMQUERYLISTENER (None) → None	Securely receives messages from nodes in other quorums and internally replies to them, possibly after coordinating with other nodes in own quorum.
PIRGETQUERYVECTORS(File identifier id , Perfect hash function f , Number of nodes s) → Query vectors $[q_1 \dots q_s]$	Computes s query vectors, one for each server that holds a database indexed by the output of a perfect hash function f , in order to obtain a file at index id .
PIRPERFORMQUERY(Query vector q_i) → Response block f_i	Queries own file store for the input query q_i , and returns a response block f_i .
PIRRECREATEFILE(File blocks $[f_1 \dots f_s]$) → F	Returns a file F that is reconstructed from the s input blocks.

QUORUM layer: The client uses the `QUORUMSENDQUERY` function for sending an encrypted message to each of the nodes in a target quorum and obtaining a reply from them, given the nodes' addresses and public keys $[(A_1, PK_1), (A_2, PK_2), \dots (A_s, PK_s)]$ as well as a proof of robust communication ϕ . This function can also be configured to send a single message to an arbitrary quorum node, retrying with different nodes until it obtains a valid reply. The recipient nodes in the target quorum listen for messages from other nodes in the `QUORUMQUERYLISTENER` function. Depending on the flags in the message, a recipient node may be delegated to forward different values back to its peers and send a response from these peers back to the client node. All honest nodes regularly execute a Byzantine agreement protocol among themselves to guarantee that they agree on the state of the database (including a perfect hash function as we discuss later) stored at other honest nodes.

PIR layer: We observe that all nodes in a quorum share the same database of files; we provide an overview of the join protocol that enables this shared state at the end of this section. The client uses the `PIRGETQUERYVECTORS` and `PIRRECREATEFILE` functions to generate and process IT-PIR queries while the target quorum's nodes use the `PIRPERFORMQUERY` function to perform the IT-PIR query itself. Perfect hash functions (PHFs) [BPZ13] are often used within PIR schemes to map a set of keywords for documents stored at server nodes to a set of indices, such that the client can simply compute the desired index if it knows the keyword. The client obtains the PHF f beforehand using the `DHTPIRFINDFILE` function, and then passes it as an input to the `PIRGETQUERYVECTORS` function, along with the desired file identifier id and the number of (server) nodes s . This function computes the index at which the desired file will be stored, by computing the value of f for the key id , and obtains an index i . It then creates s query vectors such that when responses to these vectors are combined in the `PIRRECREATEFILE` function, they will result in file i of the PIR database.

The `PIRRECREATEFILE` function reconstructs the file from s PIR query responses using Goldberg's `EASYRECOVER` algorithm [Gol07, Fig.3]. We show in ?? that the preconditions for using this efficient algorithm are met. The `PIRPERFORMQUERY` function computes a matrix multiplication of the query vector with a block-wise field representation of the database, using Goldberg's IT-PIR routine [Gol07, Fig.2].

3.5.2 Our interface

We present our functions for searching for a file, namely `DHTPIRFINDFILE`, and inserting a file, namely `DHTPIRPUTFILE`, shown in Algorithms ?? and ??, respectively. Both of these functions expect at least an identifier for the file to be inserted or searched.

Algorithm 10 DHTPIR file search function.

```
1: function DHTPIRFINDFILE( $id$ )
2:    $dest \leftarrow \text{QPGETTARGETQUORUM}(id)$ 
3:    $[(A_1, PK_1), \dots (A_s, PK_s)] \parallel \phi \parallel VK \leftarrow dest$ 
4:    $nodes \leftarrow [(A_1, PK_1), \dots (A_s, PK_s)]$ 
5:    $msg \leftarrow \text{SEARCH\_PHF\_PARAMS}$ 
6:    $reply \parallel sig \leftarrow \text{QUORUMSENDQUERY}(\phi, nodes, msg)$ 
7:    $PHF \parallel ts \leftarrow reply$ 
8:   if  $\text{VERIFYSIG}(reply, sig, VK) == \text{False}$  then
9:     return Error: "Could not verify PHF shares."
10:  if  $ts + \text{Current time} > T$  then
11:    return Error: "Got Expired PHF"
12:   $queries \leftarrow \text{PIRGETQUERYVECTORS}(id, PHF, s)$ 
13:   $msg \leftarrow \text{SEARCH\_PIR\_QUERIES} \parallel queries$ 
14:   $blocks \leftarrow \text{QUORUMSENDQUERY}(\phi, nodes, msg)$ 
15:   $file \leftarrow \text{PIRRECREATEFILE}(blocks)$ 
16:  return  $file$ 
```

These functions use this identifier to locate the file within the DHT (as in regular DHTs), with the exception that in our case, we wish to locate all nodes in a quorum; that is, a *set* of nodes in the DHT, all of which will store the file. Specifically, both functions execute the Query Privacy API function QPGETTARGETQUORUM with this identifier (id) to obtain network addresses and long-term public keys of the quorum's s nodes ($[(A_1, PK_1), \dots (A_s, PK_s)]$), a proof ϕ of robust communication from a quorum linked to the target quorum, and finally, a verification key VK to verify messages that are signed by a threshold of the target quorum nodes (lines 2–4).

We motivate the rest of these functions by starting with a simple protocol for each function and then optimizing its communication complexity. While designing the optimized protocols, we ensure that we meet the goals mentioned in ??, namely, we guarantee that an honest node, or a bounded-size coalition of Byzantine nodes in a quorum, cannot learn the content of a client's file retrieval query. We also meet the second aforementioned goal, that is, such a coalition of Byzantine nodes cannot cause a legitimate search or insertion query to be executed incorrectly or dropped. Additionally, we explore what actions a Byzantine *client* node can perform and consequently, we equip honest nodes with signature verification and rate-limiting mechanisms to detect Byzantine client nodes that attempt to overwhelm them with DHTPIR messages.

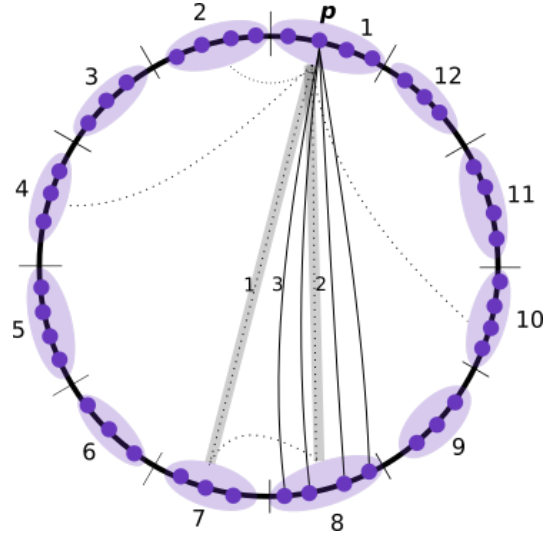


FIGURE 3.3: Layers of protocols used within DHTPIR. Quorums are numbered and shown in purple ovals. Node p from quorum 1 wishes to fetch a file that is stored by quorum 8. Routing table (RT) entries maintained by quorum 1 are shown by dotted lines. Quorum 1 uses the QP protocol to query quorum 7 (grey background lines), whose RT entry it knows. Quorum 7 then provides the RT entries for quorum 8, which is queried again through QP. PHF and PIR requests and responses are shown in solid lines; node p fetches the PHF from quorum 8, and then sends a PIR request to each node in quorum 8, from which it obtains a PIR response, to reconstruct the file.

File retrieval: We illustrate the file retrieval process in ???. To conduct a search query, the client needs to identify the index of its file in the target quorum's nodes' database. To do so, it obtains the perfect hash function (PHF) used in that quorum; we detail this step shortly. Through a call to the `PIRGETQUERYVECTORS` function, the client computes the index of the desired file identifier id using the PHF and constructs s PIR query vectors (lines 14–15). It sends one vector to each node and obtains all PIR responses through the `QUORUMSENDQUERY` function (line 16). The client reconstructs the file using these PIR responses through the `PIRRECREATEFILE` function (line 17). We parameterize the threshold for the IT-PIR scheme as follows (references for these and subsequent constraints to the relevant inequalities are included in ??).

Constraints for threshold of IT-PIR scheme:

1. Byzantine nodes in a quorum *cannot* reconstruct a PIR response through collusion (inequality ??).
2. Honest nodes can *efficiently* reconstruct the desired file, when a given fraction of nodes per quorum is Byzantine (inequality ??).

Naively, the client could obtain the PHF by simply querying each node in the target quorum and majority filtering the received responses. Since all target nodes must agree on a PHF for indexing the database, ostensibly the client could query one target node for the PHF. (We describe PHF computation while discussing our file insertion function.) However, this delegate node may be Byzantine and may simply not respond. In this case, through the QUORUMSENDQUERY function, the client retries sending the message to a different node until it receives a reply (lines 5–6). (Again, we observe that eventually the sending node will reach an honest node as only a minority of nodes in the quorum are Byzantine.)

A Byzantine delegate node may also collude with other Byzantine nodes to send a PHF that is built after excluding certain files' indices from the input set, or after swapping some files' indices around. Thus, the client should receive a confirmation from enough honest nodes in the quorum that a valid PHF was returned. Therefore, the delegate node obtains signature shares over a hash of the PHF and sends the threshold signature back to the client. As the client knows the quorum verification key VK , it can verify this signature (lines 6–10). Yet, a Byzantine node may simply return a valid PHF and signature shares that reflect the state of the database in the past. To guarantee freshness, we require the delegate node to concatenate a timestamp with the PHF, before computing the signature shares (lines 11–13). We parameterize the threshold for Young et al.'s threshold signature scheme as follows.

Constraints for Young et al.'s threshold signature scheme (inequality ??):

3. Byzantine nodes cannot produce enough valid signature shares through collusion.
4. Byzantine nodes cannot prevent the reconstruction of a signature by simply failing to respond.

Algorithm 11 DHTPIR file insertion function.

```
1: function DHTPIRPUTFILE( $id, F$ )
2:    $dest \leftarrow \text{QPGETTARGETQUORUM}(id)$ 
3:    $[(A_1, PK_1), \dots (A_s, PK_s)] \parallel \phi \parallel VK \leftarrow dest$ 
4:    $nodes \leftarrow [(A_1, PK_1), \dots (A_s, PK_s)]$ 
5:    $msg \leftarrow \text{"INSERT_FORWARD"} \parallel F$ 
6:    $reply \parallel sig \leftarrow \text{QUORUMSENDQUERY}(\phi, nodes, msg)$ 
7:   if  $reply \neq \text{Hash}(F)$  then
8:     return Did not get the correct hash of the file in response.
9:   if  $\text{VERIFYSIG}(reply, sig, VK) == \text{False}$  then
10:    return Could not verify the signature of the remote quorum.
11:  return
```

File insertion: We present our file insertion function in ???. We remind the readers that a file itself is typically a secret share of a document or a manifest, as is common in CRP systems (Section ??), and thus, plaintext files, by themselves, would reveal no information to the nodes in the quorum that store the file. Intuitively, for inserting a file into the target quorum, the client needs to send its copy to each node in the target quorum. To preserve the confidentiality and integrity of the file content, consider a baseline wherein the client encrypts the file to each of the target nodes, given their public keys PK_c . Again, as an optimization, it could send the file to only one of the nodes that acts as a delegate node and forwards the file to all other nodes in the quorum. We deploy the same technique as for the PHF, to ensure that these nodes get the correct file, namely, the delegate node should return threshold signature shares over a hash of the file to the client (line 6). The client ensures that the signature is over the expected hash of the file (lines 7–9) and verifies it using the quorum’s verification key VK (lines 10–12).

A Byzantine delegate node may send the file to only a minimum number of honest nodes in the quorum such that it can compose a threshold signature over the hash of the file, to reply seemingly honestly to the client. When the nodes synchronize their data store through a Byzantine agreement protocol, it can deny having obtained such a file, and thus render the honest nodes as malicious or de-synchronized [CL99]. We specifically parameterize the fraction of Byzantine nodes as follows.

Constraint for desynchronization attack [CL99]:

5. The number of honest nodes that retain this file at the end of the aforementioned attack is sufficient to run the Byzantine agreement protocol correctly (inequality ??)

Finally, we remark that a minority of Byzantine nodes that receive any file may not store it at all; instead, they may simply compute the requisite hashes for insertion queries and drop all search queries. Within our model, we assume that all honest nodes follow all protocols correctly and therefore, that they store the files. Thus, given our goodness invariant holds, enough honest nodes will reply back with correct response blocks to future search queries, such that the impact of the lazy Byzantine nodes is nullified.

PHF updates: The target quorum's nodes can recompute a PHF after each file insertion, or in occasional batches, at the cost of temporary file retrieval misses until the insertion protocol fully completes. We remark that the PHF computation cost is very small — approximately $0.25 \mu\text{s}$ per file in the database. Therefore, if nodes receive a file retrieval request while re-computing the PHF, the impact on latency due to the PHF computation is negligible, as evidenced in our latency simulation experiments. In other words, the PHF recomputation does not necessitate a period of quiescence or low network churn. After recomputing the PHF, each node should recompute its own signature share over the PHF and broadcast it to other nodes in the quorum. Any node that is chosen as a delegate node may then respond with the updated PHF and signature shares.

Byzantine client node: We have assumed so far that the client behaves as an honest node, that is, it runs the `QUORUMSENDQUERY` function correctly. However, the client may spam a target quorum's nodes through an incorrect execution of this function. We now describe rate-limiting mitigations for spamming attacks within the recipient node's `QUORUMQUERYLISTENER` function.

The receiving node expects each incoming message to contain a valid proof. This check extends to file insertion messages forwarded by a delegate node, thereby preventing a Byzantine delegate node from inundating other nodes with fake insertion messages. As mentioned within the QP layer description in ??, a valid proof should consist of a signature σ of a neighbouring quorum over the new node's network address A , its public key PK , and a timestamp ts . The receiving node only accepts a proof

that is obtained within a predetermined time interval T of the timestamp ts from the network address A .

With these restrictions, a Byzantine client can only spam a receiving node within a time interval of T units from the timestamp in the proof, and may only do so by sending messages with the network address that is in the proof. Furthermore, the receiving node ensures that a valid proof will only be processed once for each different flag in the accompanying message within the time interval T . This check allows for an honest node to use a valid proof to conduct one insertion query and one search query with a target quorum.

We note that the recipient node may belong to the target quorum or to any intermediate quorum in path to the target quorum. So, for instance, a malicious client who ran a million instances of the QUORUMSENDQUERY at the same time would be prevented from doing so by nodes in its own quorum, since the aforementioned check would fail. We note that in practice, these checks in the QUORUM layer would complement, and not replace, IP-based rate limiting that already exists in DHTs. Due to IP-based rate limiting, the recipient node may detect and prevent a denial-of-service attack from a malicious client with a given network address. Thus, our rate-limiting mitigations in the QUORUM layer (within the QUORUMQUERYLISTENER) ensure that a malicious client cannot use our QUORUMSENDQUERY function to *amplify* a denial-of-service attack.

Join protocol: A new node p interacts with multiple quorums as it joins the network; we illustrate the join protocol through the sequence diagram in Figure ???. At a high level, we use Sen and Freedman’s commensal cuckoo rule [SF12] to assign addresses to primary and secondary joining nodes. We use threshold signatures within Young et al.’s robust communication protocols [YKGK10] to securely notify quorums of these joins, and to securely transfer the stored database to these nodes. As we run Kate and Goldberg’s distributed key generation (DKG) scheme [KG09] within the join and leave protocols to redistribute a secret throughout the quorum, we must ensure that its prerequisite is met. We thus obtain the following constraint:

Constraint for DKG [KG09]:

6. The fraction of Byzantine nodes per quorum should satisfy the upper bound for the DKG protocol (inequality ??).

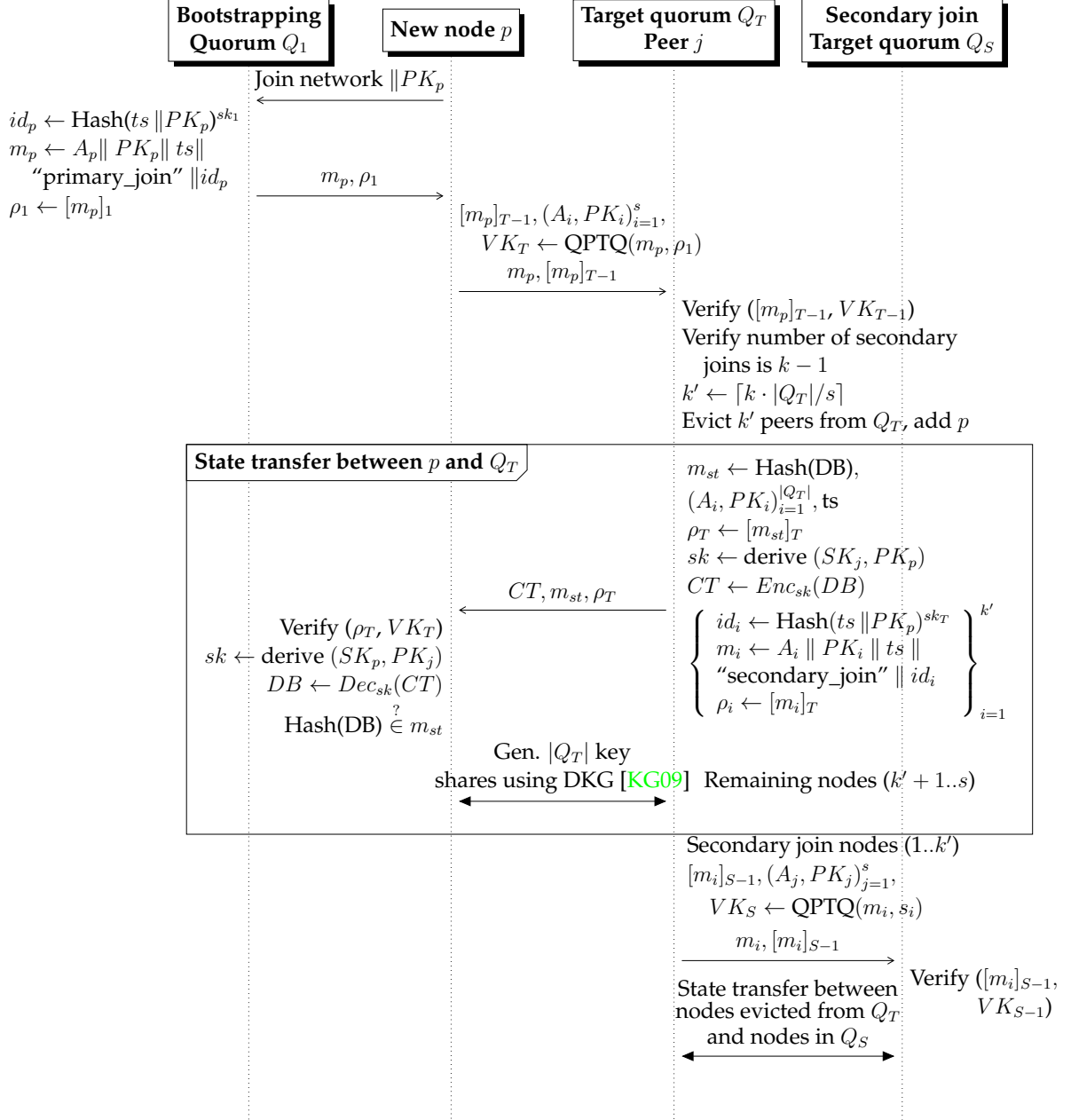


FIGURE 3.4: DHTPIR Join protocol. Here QPTQ refers to the QPGETTARGETQUORUM function.

We now detail the join protocol. When a new node joins the network, it first contacts a bootstrapping peer q , which is a part of quorum Q_1 for instance, and sends its public key PK_p . Peer q then sends network addresses and public keys of its peers, signed by quorum Q_1 , thereby allowing node p to send this bootstrapping message to node q 's peers. Nodes in the quorum Q_1 collectively generate a random address using their shares of the quorum's shared secret s . They generate a cryptographic hash of the new node's public key PK_p , and include a timestamp for freshness. Treating this hash as a group element, they raise it to the power of their share, and send it to the new node. Node p combines these shares to obtain a random address id_p .

To contact the target quorum Q_T that spans address id_p , node p obtains a valid proof of communication by running the QPGETTARGETQUORUM function. This function may be run with either RCP-I or RCP-II as the underlying communication protocol, though there are two small changes when it is run for primary or secondary joins. First, the lookup message m_p also includes its securely assigned virtual address id_p and a string to demarcate the message as a primary (or secondary) join, on top of the node's network address A_p , public key PK_p and a timestamp ts . We need to distinguish primary joins from a secondary join, as a new Byzantine node could masquerade as a secondary join to avoid causing the eviction of other Byzantine nodes in its target quorum. Second, for a primary join, as node p does not belong to a quorum, the bootstrapping peer's quorum Q_1 produces a threshold signature over the message m_p , which is denoted by $[m_p]_1$ within our diagram. Through this function, node p then proceeds to contact multiple quorums, to ultimately obtain the usual response; that is, a threshold signature over m_p by the nearest neighbour of the target quorum, namely quorum Q_{T-1} , as well as network addresses and public keys of nodes in the target quorum $((A_i, PK_i)_{i=1}^s)$, and the verification key of the target quorum VK_T . Node p then forwards the message m_p and the threshold signature $[m_p]_{T-1}$ to the target quorum.

The target quorum verifies the threshold signature $[m_p]_{T-1}$, using the verification key of its neighbour quorum $VK_{Q_{T-1}}$, as usual. When the signed message contains the primary join flag, a node in Q_T must first vet the join, as proposed within the commensal cuckoo rule (step 1CCR in Figure ??). That is, each node first ensures that the quorum has had at least $k - 1$ secondary joins since the last primary join into the quorum. Here, k is set such that an average of k nodes are cuckoo-ed out in a k -region that spans a fraction $\frac{k}{n}$ of the address space. (In case this check fails, the target quorum replies back with a signed failure message that is bound to the input message m_p . Node p repeats the aforementioned process with the bootstrapping peer, until it reaches a new target quorum that allows the primary join.) Nodes in Q_T then compute the number k' of its nodes that must be evicted from it and rejoin another quorum, following Sen and Freed-

man’s commensal cuckoo rule. They then recompute the size of its quorum based on k' . Finally, each node in Q_T generates a message m_{st} that consists of the database hash, the network addresses and public keys of the *remaining* peers, and a timestamp. Each node computes its signature share over this message, and forwards it to one delegate node, say node j . This delegate node derives a shared secret with node p , using the latter’s public encryption key and its own private key SK_j and uses this shared secret to encrypt the database to node p . Node j merges the threshold signature shares to a threshold signature ρ_T and forwards it along with the message m_{st} and the encrypted database, to the new node p .

The new node verifies the threshold signature ρ_T using the quorum verification key VK_T that it obtained earlier. It ensures freshness of the message m_{st} by checking its timestamp. It uses the peer node j ’s public encryption key PK_j that it obtained earlier to establish the same shared secret sk , and decrypts the encrypted database in the message. It computes a hash of this database and checks it with the hash contained within the message, to detect if peer j maliciously sent an incorrect copy of the database. (In this case, it sends $(m_p, [m_p]_{T-1})$ to another node of the target quorum.) The new node runs Kate and Goldberg’s distributed key generation (DKG) scheme [KG09], in conjunction with the *remaining* nodes of the target quorum, to derive a signing key share for itself and verification key shares for these nodes. We note that within the DKG scheme, the new node can use the network addresses and public keys of the remaining nodes in the target quorum $((A_i, PK_i)_{i=1}^{|Q_T|})$, which were obtained within m_{st} . This scheme requires that the fraction of Byzantine nodes per quorum should be less than a third, as explicated in constraint 6. The threshold for this threshold signature scheme should be maintained to satisfy constraints 3 and 4. Thus, after a certain number of new node insertions, a new signing/verification key may need to be generated and the neighbouring quorums would need to be informed of the verification key. The target quorum also updates its routing table entries to include the network address A_p and public key PK_p of the new node, and excludes addresses and keys of the k' old nodes. The RCP-II protocol involves informing the quorum’s neighbours of these updates in the routing table entries; we refer the reader to Section IV-C in Young et al.’s paper for details [YKGK10].

The nodes to be evicted from the target quorum do not participate in the DKG. After computing these threshold signatures, each node in the target quorum proceeds to evict k' of its nodes as follows. They generate a random node address, in a similar manner as before, compute a cryptographic hash of it, and expand this hash to construct k' new random addresses for the nodes to be evicted. They then identify which nodes in the quorum should be evicted, based on their current addresses and the new

TABLE 3.3: Notation for security constraints for DHTPIR

n	Number of peers in the network
ϵ	Network-wide ratio of Byzantine nodes to honest nodes
s	Total number of peers in a quorum
h	Honest peers in a quorum that reply to an IT-PIR query
b	Byzantine peers in a quorum (includes failing nodes)
t	Max colluding peers for IT-PIR queries [Gol07]
τ	Threshold for Young et al.’s threshold signature scheme [YKGK10]

set of addresses. For each evicted node, the target quorum generates a message m_i and threshold signature $[m_i]_T$ over it; m_i is identical to the message m_p in structure for all fields, but differs in that its string flag indicates a secondary join. Each evicted node follows a similar process as the new node did with the bootstrapping quorum, in order to communicate with its new target quorum. (Such nodes may now safely delete their databases.) It runs the QPGETTARGETQUORUM function as node p did earlier in the protocol, and forwards m_p as well as a threshold signature by the neighbour of its new target quorum, $[m_i]_{S-1}$ to the new target quorum Q_S . The quorum Q_S accepts all incoming secondary joins and secondary joins do not result in further cuckoos. So, with the exception of the corresponding check for primary joins and the generation of IDs, and signed join messages (id_i, m_i, ρ_i) , this quorum and the secondary joining node repeat the process of sharing the database, and generating $(|Q_S| + 1)$ new key shares.

Leave protocol: If a node decides to leave a quorum (outside of a primary join), then it can send a time-stamped message which is signed by its own private signing key. The remaining nodes may then need to re-run the DKG protocol, or regenerate the quorum’s signing/verification key, depending on the current value of the threshold for the threshold signature scheme.

3.6 Security Parameters Analysis

We now compose all constraints that we have identified in the last section. We show that as long as the fraction of Byzantine nodes per quorum is less than $\frac{1}{4}$, constraints 1–6 hold. We present the notation used in our analysis in ???. We first establish some invariants using this notation.

Invariants: Out of s peers in a quorum, a large number of h peers provide honest replies whereas the rest of the peers, say b , are Byzantine and either fail to provide a

reply or provide incorrect replies. Thus, we have that $s = h + b$. This equation holds in each quorum of s nodes in the network, which may have varying sizes. However, the particular value of s is irrelevant for the following calculations, and thus we normalize our equations by dividing by s and denote the resulting fractions by appending a subscript 0 to the original symbols, yielding the following, where $b_0 < h_0$:

$$h_0 = 1 - b_0 \quad (3.1)$$

RCP constraints: The threshold for Young et al.'s threshold signature scheme, namely τ , is bound by constraints 3 and 4, as follows.

$$b_0 < \tau_0 \leq h_0 = 1 - b_0 \quad (3.2)$$

The DKG protocol used by Young et al.'s scheme imposes the following bound on b_0 (constraint 6):

$$s \geq 3b + 1 \Leftrightarrow 1 \geq 3b_0 + \frac{1}{s} \quad (3.3)$$

These threshold signature shares are used to indicate a change of the state of the quorum upon insertion of a file. As a node is delegated to forward file insertion requests to all other nodes, we must prevent the desynchronization attack that was described in ?? to motivate constraint 5. We detail this attack here.

A Byzantine node might only forward such a request to its fellow Byzantine nodes and the minimum number of honest nodes required to obtain a threshold signature ($\tau - b$ nodes in our case) and then collude with other Byzantine nodes to pretend that it never obtained such a file. In this case, only $\tau - b$ nodes retain a copy of the new file whereas the remaining $s - (\tau - b)$ do not have such a copy. For the Byzantine agreement protocol that synchronizes the state of the data store across different nodes to be successful, a majority of nodes must retain a copy of the newly inserted file. That is, we must have $\tau - b > s - (\tau - b)$. Equivalently, we obtain $\tau_0 > b_0 + \frac{1}{2}$.

As this inequality tightens the lower bound in inequality ??, we obtain the following range for τ_0 :

$$b_0 + \frac{1}{2} < \tau_0 \leq 1 - b_0 \quad (3.4)$$

This inequality is satisfiable as long as:

$$b_0 < \frac{1}{4} \quad (3.5)$$

Note that any admissible value of b_0 that satisfies inequality ?? also satisfies inequality ??

for any quorum with at least $s \geq 4$ members.

IT-PIR: In our notation, at most t nodes can collude without being able to reconstruct the desired row of a database from an IT-PIR query. As all of the b Byzantine nodes in a quorum *may* collude, constraint 1 requires that t can only be as small as the number of Byzantine nodes:

$$t \geq b \quad (3.6)$$

Second, Goldberg's EASYRECOVER algorithm [Gol07, Fig.3] shows that if the following condition holds, a client can guarantee efficient reconstruction of the desired row from the responses of the s peers, b of which may be Byzantine and thus send incorrect responses. So, to satisfy constraint 2, we need:

$$h > \frac{s+t}{2} \Leftrightarrow h_0 > \frac{1+t_0}{2} \quad (3.7)$$

Substituting equation ?? and simplifying, we get:

$$1 > 2b_0 + t_0 \quad (3.8)$$

Therefore, as expected, with an increase in the fraction of Byzantine peers, the IT-PIR threshold should decrease in order to reconstruct the message, without any other changes. Inequalities ?? and ?? constrain the range of this threshold, as follows.

$$1 - 2b_0 > t_0 \geq b_0 \quad (3.9)$$

For a value of t_0 that satisfies inequality ?? to exist, the difference between the upper and lower bounds of that inequality should be strictly greater than $\frac{1}{s}$, as follows.

$$1 > 3b_0 + \frac{1}{s} \quad (3.10)$$

Any admissible b_0 that satisfies the strictest inequality for Young et al.'s protocols, that is, inequality ??, also satisfies the strictest inequality for Goldberg's IT-PIR scheme, that is, inequality ??, for any quorum with at least $s \geq 4$ members. Our requirement that the fractions of Byzantine nodes in each quorum be strictly less than $\frac{1}{4}$ is therefore sufficient for the requirements of Young et al.'s robust quorums, and also Goldberg's IT-PIR scheme.

Quorum size/ ϵ trade-off: For a network with at most n nodes, the desired average size of quorums formed through the commensal cuckoo rule can be set beforehand to

$\bar{s}$. The adversary can only insert at most $\epsilon \cdot n$ nodes into this network and re-join these nodes, as per the join protocol in ??, in an attempt to overwhelm a quorum. Increasing the target $\bar{s}$ allows us to increase ϵ at the expense of a greater intra-quorum communication cost.

We extend Sen and Freedman’s [SF12] simulation and we estimate the maximum value of ϵ that can allow only $b_0 < \frac{1}{4}$ of each quorum to be malicious. We conduct the simulations across a range of network sizes (n) and over 100K rounds of re-join attempts. We find that small networks ($n = 6400, 12800$) can sustain ϵ up to 0.03, while maintaining a desired average quorum size of $\bar{s} = 25$ and reaching a maximum observed quorum size of $s = 74$. When larger networks ($n = 10 \cdot 2^\lambda \mid \lambda = 18, 19, 20$) were configured with a slightly smaller target average quorum size of $\bar{s} = 20$, the maximum observed quorum size was less than 75, and can withstand ϵ up to 0.02. Therefore, even for large network sizes, we conclude that the commensal cuckoo rule can construct reasonably sized quorums that satisfy the $b_0 < \frac{1}{4}$ constraint.

3.7 Complexity analysis

In this section, we analyze the performance of our system in terms of its message and communication complexity for the client and server nodes, for file insertions and retrievals. In terms of computation complexity, our PIR-based [Gol07] file retrieval function only requires efficient Lagrange interpolation for the client. Recent improvements in optimizing Lagrange interpolation [TCZ+20] can be utilized in an implementation of our scheme, to allow for fast reconstruction of files from blocks returned by the s nodes of the target quorum.

Before the client can send search or insertion queries, it needs to obtain the routing information of the target quorum through the QPGETTARGETQUORUM function. The message complexity incurred by the client in this call depends on whether RCP-I or RCP-II is chosen; as discussed in ??, RCP-I has a deterministic message complexity of $O(s \cdot \ell)$ whereas RCP-II has a randomized expected message complexity of $O(\ell + s)$.

First, we show that the message complexity of our insertion and search algorithms is $O(s)$. As ℓ will be $O(\log n)$ in expectation and quorums are of size $s = \Theta(\log n)$, $O(s) = O(\ell)$, the message complexity to run the whole protocol is determined by which of the two aforementioned protocols is chosen. Second, we compare the message complexity for the target quorum to participate in our PIR-based protocol to the case when the quorum participates in Backes et al.’s OT-based one.

When an honest node in the target quorum is delegated to insert a file or re-

spond with a PHF, it exchanges a round of messages with other nodes in the quorum ($2 \cdot (s - 1) = O(s)$ messages) to obtain threshold signatures over the file or PHF hashes. Consider the baseline discussed previously in ?? wherein for file insertion, the client simply sends the file to all nodes, without requiring any acknowledgements back, or, for PHF retrieval, the client expects a PHF back from all nodes. As compared to this baseline, the target quorum incurs an identical message complexity, namely $O(s)$ for the entirety of file insertion, or for PHF retrieval; the messages are sent within the quorum, instead of to the client. However, the client may send the file or PHF request messages to multiple Byzantine nodes that drop them, and thus keep resending them until it reaches an honest node. Modelling the number of times that a node needs to be sampled, without replacement, until a *single* honest node is found, as a negative hypergeometric distribution, the expected number of nodes contacted is upper bounded by $\frac{1}{h_0}$. Thus the client incurs a lower message complexity (by a factor of h_0), for file insertion and PHF retrieval, than the baseline.¹

In the next step for file retrieval, the client sends s PIR query vectors, each of size $\sqrt{D}$ [Gol07]. The client thus sends a total of $\frac{s}{h} + s$ messages and thus the message complexity for the client for file retrieval is $O(s)$. As each PIR query vector is of size $\sqrt{D}$, the total communication complexity of the messages sent by the client will be $s \cdot \sqrt{D} + s = O(s \cdot \sqrt{D})$.

Suppose that each node in the target quorum stores $\mathcal{F}_Q$ document chunks, each of which is L bytes long, that is, a total of $\mathcal{D} = \mathcal{F}_Q \cdot L$ bytes. Botelho et al.’s perfect hash function construction [BPZ13] for indexing these chunks allows the delegate node to send the PHF in $\kappa \cdot \mathcal{F}_Q$ bytes, where $.243 < \kappa < .3375$. Each PIR response is of size $\sqrt{D}$ bytes [Gol07]. When all Byzantine nodes send PIR responses, along with the honest nodes, the quorum incurs a worst-case communication complexity of $s \cdot \sqrt{D}$ bytes. Note that the large PIR responses outweigh the small PHFs for the communication complexity of the target quorum for reasonably sized quorums and numbers of stored files per quorum.

In contrast, consider Backes et al.’s proposed extension to their OT-based QP-RCP protocols, so that OT is used to retrieve the file itself privately from the data store in the target quorum. In this case, a delegate node responds back with the entry-wise

¹As the client waits to obtain a response before retrying with another node, it incurs a higher expected latency in our new scheme than the baseline. The client may send the file in parallel to multiple delegate nodes for insertion queries, significantly increasing its chances of reaching an honest node, and reducing the latency at the cost of a higher message complexity. Similarly, it may query several nodes in parallel for the PHF, each of which will return the signature over the PHF, for search queries.

encrypted data store that is $\mathcal{F}_Q L$ bytes long as well as a total of $2 \cdot \mathcal{F}_Q + 1$ OT parameters, each of a small constant length γ bytes. For our scheme to incur a lower communication complexity for the target quorum than Backes et al.'s, we must have:

$$s \cdot \sqrt{\mathcal{D}} + \kappa \cdot \mathcal{F}_Q < \mathcal{D} + (2\mathcal{F}_Q + 1) \cdot \gamma$$

As $\kappa \ll 2 \cdot \gamma \ll L$, we drop the terms containing κ and γ , to obtain:

$$\mathcal{D} = \mathcal{F}_Q \cdot L > s^2 \quad (3.11)$$

The expected number of files per quorum (expected value of $\mathcal{F}_Q$) will be the total number of files in the network (say $\mathcal{F}_N$) divided by the number of quorums ($\frac{n}{s}$), yielding $\frac{s\mathcal{F}_N}{n}$. Substituting this value for $\mathcal{F}_Q$ into inequality ??, we get the following lower bound on the number of files in the network for our scheme to have a lower communication complexity for quorums, than Backes et al.'s:

$$\mathcal{F}_N > \frac{sn}{L} \quad (3.12)$$

As $s \ll L$, our scheme requires less communication complexity for quorums than Backes et al.'s private file retrieval variant, when any reasonably large numbers of files are stored in the network.

3.8 Evaluation

We evaluate the efficiency of DHTPIR in terms of the communication and computation overheads to retrieve a file using it. We measure these overheads as the number of nodes or users that connect to the P2P network (n) increases, as well as when the number of files that are stored in the network (F_N) increases. We simulate our system in order to evaluate these costs for the client and target quorums' nodes in our system. We briefly describe our simulation, and then we focus on experiments that demonstrate the scalability of our system with F_N , although our simulations show that these costs for our system scale well with both n and F_N . We also remark that the communication and computation costs for publishing documents remain reasonably low. Our simulation code and output, as well as additional graphs, can be found at <https://git-crysp.uwaterloo.ca/dhtpir/simulations>.

Simulation setup: In our simulation, one node serves as the client and creates F_N randomly generated document chunks, each of size 1 KiB. It stores each chunk on the

quorum identified by the ID of that chunk. The client then attempts to retrieve each chunk from the network. We individually simulate each of the layers used within our system: we start with a simple DHT-based P2P network — a Base DHT that implements file insertion and retrieval. For all other layers in our simulation, we instantiate $q = 128$ quorums, each of which has $s = 64$ nodes. (We simulated our interface for $q \in \{16, 32, 64, 128\}$; we discuss results for $q = 128$ as it is representative of this range.)

The next layer is a simulation of Young et al.’s probabilistic robust communication protocol (RCP-II) to efficiently route queries across quorums. That is, our RCP simulation supports file insertion and retrieval queries by replacing the routing protocol in the Base DHT simulation with the RCP-II protocol. (We remark that none of the nodes in our simulation behaves in a Byzantine manner.) We then simulate Backes et al.’s QP extension to Young et al.’s RCP-II protocol, to conduct private routing queries, in the RCP+QP simulation. Finally, we include two protocol simulations that enable clients to privately retrieve files. We simulate our own DHTPIRFINDFILE and DHTPIRPUTFILE functions within the DHTPIR simulation. We also simulate the closest related work, namely extending Backes et al.’s OT protocol for the *last hop* to the target quorum, in order to privately retrieve a chunk from it; we refer to this protocol simulation as LastHop. We delegate one node from the target quorum to conduct the OT-based protocol for LastHop, and use threshold signatures just as with RCP-QP-II.

For our performance simulation, we use the following estimates of network and computation speeds. We set the network bandwidth to 50 Mb/s, and the round-trip time (RTT) to 150 ms. We estimate the PIR computations to run at the rate of 0.25 s per GB of database, whereas encryption operations run at 1 GB/s (about 3 cycles/byte for AES-NI). For all experiments, we increased F_Q roughly exponentially, that is, $F_Q \in \{100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000\}$. (A total of $F_N = F_Q \cdot q$ chunks are stored in the network.)

Communication complexity: We plot the total communication complexities over 25 runs for the DHTPIR, LastHop and RCP+QP simulations in ???. For the LastHop simulation, as explained earlier, the OT request sent by the client is constant in size with respect to increasing F_Q , whereas the OT setup and response sent by the delegate node is proportional to F_Q in size. Thus, the total communication complexity of the LastHop simulation increases linearly in F_Q in ??.

In contrast, DHTPIR provides a smaller communication complexity for file retrieval *responses*, at the cost of a larger communication complexity for the corresponding *requests*. The total communication complexity overhead of DHTPIR over RCP includes the cost for the PHF request, PIR request and PIR response. This overhead increases by

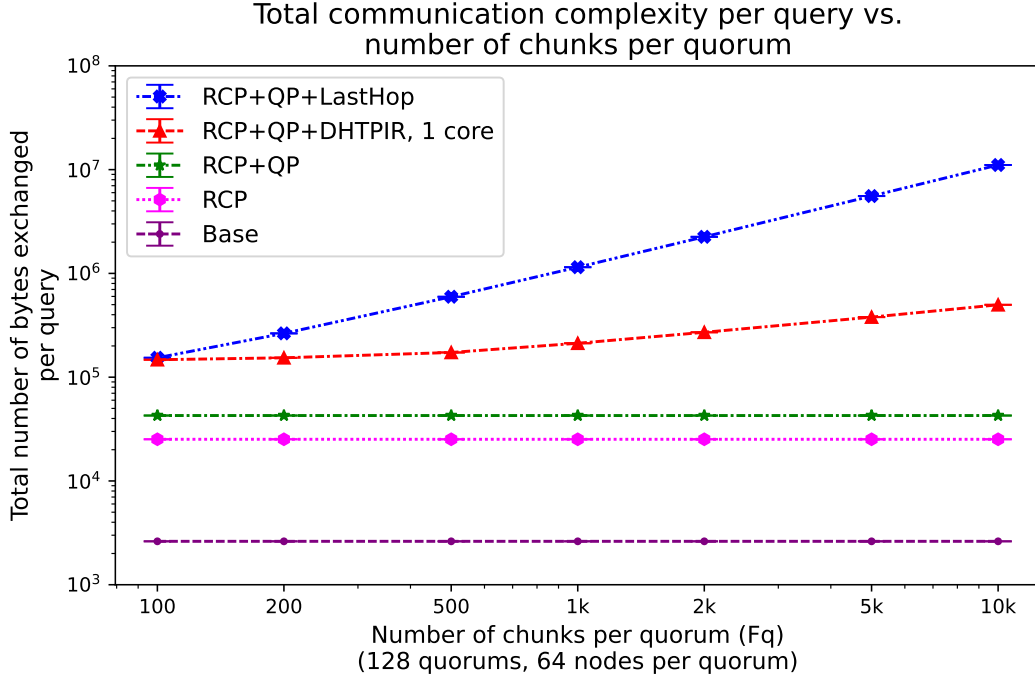


FIGURE 3.5: Measurement of the total communication complexity of one query for one chunk, including the request and the response, as the number of chunks stored in each quorum increases roughly exponentially. This plot shows the average communication complexity over 25 runs of the simulation. Only RCP+QP+DHTPIR and RCP+QP+LastHop provide the desired privacy properties; other lines are plotted in order to show the overheads incurred by each layer of DHTPIR. Note the logarithmic scales. Error bars are shown, but they are too small to see.

approximately the asymptotic amount ($s \cdot \sqrt{F_Q \cdot L}$) of bytes discussed in ??, as expected.

Performance simulations: We measure the latency of DHTPIR and LastHop simulations to retrieve a file. We also measure their throughput in terms of number of simultaneous clients' file retrieval requests that can be handled by the entire system. (We measure the latency and throughput of *routing* requests for RCP+QP, as routing requests are the bottleneck for that protocol, which has no privacy protection for the actual retrieval.) We plot the latencies of these three systems as F_Q increases in ?. Both LastHop and DHTPIR incur two additional round trips over RCP+QP: to obtain PHF or OT parameters and then to obtain a PIR or OT response. However, as LastHop includes the entire database (up to ≈ 10 MB in the figure) within the OT response, we can see in ? that as the number of chunks per quorum (F_Q) increases, LastHop incurs a linearly

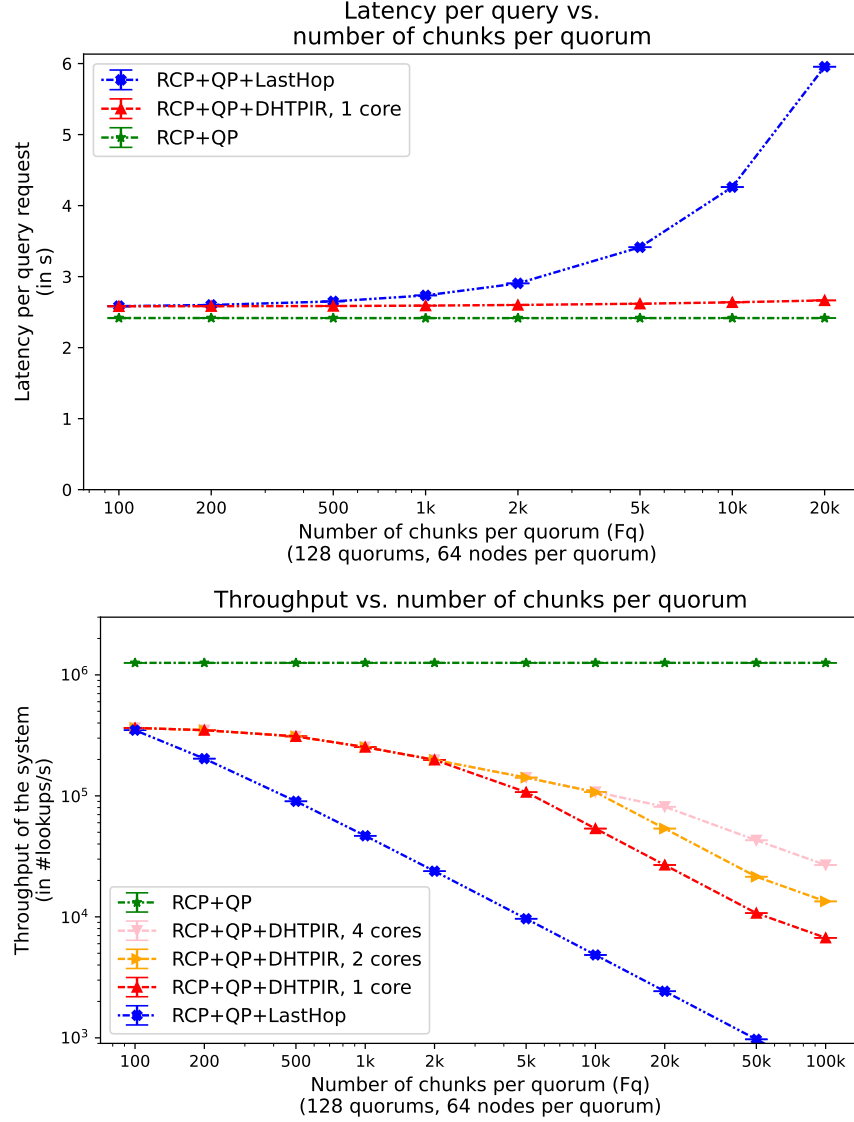


FIGURE 3.6: Measurement of the (a) latency and (b) throughput, as the number of chunks stored in each quorum increases roughly exponentially. These plots show the average latency and throughput over 25 runs of the simulation. Note the logarithmic y-axis in (b). Error bars are shown, but they are too small to see. We emphasize that the RCP+QP layer does *not* provide the desired privacy properties; it is used to demonstrate a lower bound for the latency plot in (a) and an upper bound for the throughput plot in (b).

increasing latency overhead over RCP+QP. On the other hand, for DHTPIR, as the PIR requests and responses only grow proportional to $\sqrt{F_Q}$, we do not observe a significant increase in latency due to computational costs or communication costs, over the range of F_Q in ??.

We plot the throughput of RCP+QP, LastHop, and DHTPIR as F_Q increases in ??. For LastHop, as only a single delegate node in the target quorum may compute the OT response (and computing threshold signatures is much faster), multiple nodes can process distinct OT requests in parallel. For DHTPIR, *each* node participates in computing the PIR response for a single PIR request; adding another core allows for parallel processing of multiple *distinct* PIR requests. (Adding more cores has quickly diminishing effects, however, as the network speed soon becomes the bottleneck.)

We can see in ?? that DHTPIR experiences a small constant drop in its throughput over RCP, until $F_Q \approx 2000$ — it stays network-limited in this range — beyond which, adding another core results in a significant improvement in throughput, which can be extrapolated to expand with database size. Thus, DHTPIR affords opportunities to optimize its throughput for nodes that store large databases, through multi-core processing. (We do not consider processing a single PIR request across multiple cores or threads; though efficient parallelizable matrix multiplication algorithms can be used to achieve even greater throughput for DHTPIR [LG15].) Even though each *node* in LastHop can process multiple distinct OT requests in parallel, it incurs a large reduction in throughput as compared to RCP, proportional to the size of the database ($F_Q \cdot L$). As LastHop is significantly network limited, adding more cores would not increase its throughput. DHTPIR thus achieves a throughput that is about two orders of magnitude larger than LastHop and this gap widens with database size.

Our performance simulations show that DHTPIR incurs almost no change in latency and only a modest drop in throughput over the baseline RCP+QP simulation (which again does not protect the privacy of the queries from the target quorum at all), across increasing database sizes. DHTPIR shows a marked improvement in latency and throughput across larger databases over LastHop, and scales significantly better in comparison. Our simulation thus provides us reasonable grounds to believe in the efficiency of a DHTPIR implementation.

3.9 Deployment

In the CRP setting, we rely on the developers of the underlying CRP system to provide methods to bootstrap a user with the complete client-side software, which would

include a DHTPIR implementation. For instance, a service like GetTor [Pro20] can be used to deliver the software. To join the network, DHT-based systems typically require a new node to contact a node from a small list of permanently online nodes [SWF20], which is included in the software package. These nodes are trusted to provide correct routing information; that is, to not conduct an eclipse attack. For a CRP system that uses DHTPIR, including the quorum verification keys of these nodes' current quorums with this list allows a new node to detect and prevent an eclipse attack, through the robust communication protocols. Similarly, we also rely on popular adoption of the underlying CRP system. For instance, the system can be seeded with popular, non-prohibited content, thereby inflicting a high collateral damage onto the censor if the entire system were to be banned.

Although we have focused on CROPS in order to motivate our interface, an implementation of our interface can be used to publish and privately retrieve a block of data within other publishing systems, such as in Tangler. Our design can also support other applications that are built atop DHTs, such as distributed health and medical data repositories. A system like ours is also useful to integrate into DHTs used for commercial settings, to minimize the amount of potentially sensitive data nodes can collect.

3.10 Conclusion

Censorship-resistant publishing systems are built atop DHTs and enable users to store sensitive documents onto multiple server nodes in different administrative regions, such that a censor cannot easily take down or tamper with the published content. Server node administrators may be compelled by censors to reveal the documents retrieved by a given user. In this work, we address this problem and consequently, realize the MINIMIZE privacy design strategies for distributed hash tables.

We propose an interface that uses information-theoretic private information retrieval (IT-PIR) to prevent node operators from being exposed to information about which document was retrieved. We integrate existing work on hardening peer-to-peer networks so that quorums of nodes only contain at most a certain fraction of Byzantine nodes; we allow the censor to run such colluding Byzantine nodes. Our key insight lies in using quorums as a coalition of server nodes that store a set of files over which IT-PIR queries can be performed. In other words, DHTPIR can be seen as realizing the SEPARATE strategy even further, by separating out information known to each server peer, such that a quorum of them need to collude in order to learn what was queried. We measure the efficiency of our system by simulating it. In particular, we find that its throughput

is two orders of magnitude higher than the closest related work and scales reasonably with the database size.

Designers of an upcoming distributed file storage system, namely the Interplanetary File System (IPFS), have proposed establishing quorums [jbe15] as well as privately fetching content from nodes using Young et al. and Backes et al.'s systems [gpe18], among other privacy features [mgo18]. Since our design integrates the aforementioned systems, we revisit and extend it in the subsequent chapter, with the goal of providing privacy protection for IPFS.

Chapter 4

Peer2PIR

In this chapter, we continue with peer-to-peer networks as our back-end web architecture. The InterPlanetary File System (IPFS) is a distributed file system, hosting over 190,000 peers spanning 152 countries. Emerging decentralized technologies such as IPFS enable building distributed back-end web applications on top of DHTs. Despite its prominence, the privacy properties that IPFS offers to peers are severely limited. Any query within the network leaks to other peers the content for which a peer is querying. We address IPFS' privacy leakage across three functionalities (peer routing, provider advertisements, and content retrieval) in our system, namely Peer2PIR, ultimately enabling peers to privately navigate and retrieve content in the network.

Similar to DHTPIR, PeerPIR minimizes the amount of information that a peer is exposed to, in the process of answering another peer's queries, and thereby operationalizes Hoepman's MINIMIZE strategy (??). Although we address similar problems in DHTPIR and Peer2PIR, we use different approaches. Since DHTPIR used information-theoretic PIR schemes, we faced difficulties incorporating these schemes into IPFS, as we detail in ???. Consequently, in this chapter, we rely on computational PIR (CPIR) schemes. We hide the content of the query from other peers through these schemes, which use homomorphic encryption, and thereby, Peer2PIR realizes the HIDE strategy. Our work highlights and addresses novel challenges inherent to integrating CPIR into distributed systems. We present our new, private protocols and demonstrate that they incur minimal overheads compared to the status quo in IPFS.

Our work is beneficial to PETs researchers and privacy engineers through two additional contributions. First, we systematically compare state-of-art PIR protocols in the context of distributed systems; our comparison may be of independent interest to re-

searchers developing PIR schemes. Second, we glean privacy engineering takeaways through our experience integrating these protocols into the IPFS codebase. These takeaways may be useful to privacy engineers and software engineers who work on distributed systems.

4.1 Introduction

As a distributed file system built on top of a P2P network, IPFS provides three high-level functionalities to its peers. First, it arranges peers in an overlay topology such that each peer can contact and communicate with any other peer efficiently. IPFS implements this functionality of *peer routing* using distributed hash tables (DHTs). Second, peers can advertise to other peers that they provide a file. Analogously, any other peer on the network can discover which other peers provide a desired file. IPFS refers to this functionality as *provider advertising*. Third, a peer can directly contact another peer that provides a file and retrieve the file from them. The two peers engage in a *content retrieval* protocol to share that file. Each of these three functionalities is performed through protocols involving queries between different pairs of peers.

Consider one peer acting as a client and another acting as a server in IPFS, where the server peer is storing content and assumed to be a passive adversary. We observe that in each of the three aforementioned functionalities, the client reveals what it is querying for to the server. Peer routing reveals which peer the client is attempting to contact. Moreover, a server learns which file the client wishes to retrieve in a provider advertisement query and in a content retrieval query.

Prior work addressed only one of these problems in isolation, such as by enabling confidentiality for the target file in content retrieval [DET23; DT23] or obfuscating the target peer in peer routing [Mic23a]. Replacing one aspect of the system with a private version is insufficient if the remainder of the system leaks the queried content. For example, a server peer can learn the target file through a provider routing query, even if a content retrieval query hides the target file.

This work proposes an end-to-end solution for IPFS that hides the *content a client queries for and retrieves in a response* from a server. Ours is the first work to address this problem holistically across IPFS protocols for peer routing, provider routing, and content retrieval. This provides a comprehensive solution for advancing the privacy of users within IPFS — not only is the content of their queries protected from intermediate peers, but also from the final peer from whom they retrieve the content. In developing our solution, we handle challenges inherent to DHTs, such as a dynamic network with

churn, and variable amount of content stored at each peer. Towards developing this solution, we present the following building blocks:

- a novel routing algorithm that enables an end-to-end procedure for privately contacting a peer (??);
- generic binning and hashing techniques to adapt datastores to be amenable to PIR, and apply them to the provider advertisement and content retrieval stores (??);
- a scalable and private adaptation of an interactive content retrieval protocol (??); and,
- a systematic comparison of state-of-the-art PIR protocols in the context of DHTs (??).

Furthermore, we find that for our use cases in DHTs, existing PIR protocols do not provide a reasonable communication-computation tradeoff. We present new PIR protocols, namely PAILLIERPIR and RLWEPIR, which combine techniques from cryptographic folklore and PIR literature, to provide a reasonable communication-computation tradeoff (??), particularly over small databases. We prove the security of our PIR protocols and demonstrate that they outperform the state of the art in this setting.

Our performance evaluation demonstrates the feasibility of our system in terms of communication and computation costs. Our private peer routing and provider advertisement queries can be answered in less than 100 ms and 1 s, respectively. We believe that these are reasonable latencies for the IPFS system, given the benefit of added privacy. Furthermore, our private protocols do not incur additional rounds, and our private peer routing protocol does not significantly increase the number of hops over the original routing algorithm. Our system is modular in that it enables substituting PIR protocols for each of the three functionalities, and thus Peer2PIR directly benefits as PIR protocols become more efficient. We integrate our protocols into the IPFS codebase, highlighting and addressing privacy engineering challenges faced along the way.

4.2 Background

We first present the IPFS design. We then describe PIR schemes and outline the main challenges in adapting them to the distributed hash tables setting.

4.2.1 IPFS

IPFS is a distributed file system built on top of the Kademlia DHT [TRT+22; Pro23b; Pro24b]. We describe below the three main functionalities provided by IPFS, namely

contacting another peer in the DHT, discovering which peers provide desired content, and retrieving content from a peer. We conclude with the privacy problem inherent in naively providing each functionality. Though each peer can both respond to and conduct queries, throughout this chapter, we designate peers as *clients* when they are conducting queries, and as *servers* when they are responding to queries.

Contacting a peer. We remind the reader of the iterative routing process in DHTs, which was described in ??; we summarize this process below in the context of IPFS and the Kademlia DHT. Each peer in IPFS has a peer identifier (peer ID) that is the cryptographic hash of the peer’s public key. Peers also maintain a *routing table* consisting of peer IDs and their routing information, for a small number of other peers in the DHT. The routing information consists of *multiaddresses* which encode IP addresses and ports, as well as peer IDs. To reach a target peer, a client obtains relevant routing information by querying one of its neighbours for the target peer ID, and then querying one of that neighbour’s neighbours, and so on, through the aforementioned iterative routing process. Kademlia, similar to other DHTs, guarantees that each peer can reach any other peer in a network of size n , without a trusted central party, through approximately $\log(n)$ routing queries. Importantly, each such routing query will reveal the target peer ID to the in-path peers. The IPFS routing algorithm, in the context of the Kademlia DHT, is discussed in ??.

Discovering peers providing content. IPFS decomposes each file into multiple content blocks, using either fixed or adaptively sized chunks [Pro24a; Gre23]. Each content block in IPFS is addressed by a *content identifier* (CID), which includes a collision-resistant one-way hash of the block. CIDs remove the need for a central party to address content and are also used to verify the integrity of a content block. A peer who provides a content block in IPFS is known as a *provider peer*. DHTs support redundancy in that a given content block can be provided by multiple provider peers. Provider peers advertise the fact that they are storing a particular content block, to the DHT, through provider advertisements. A provider advertisement maps a given CID to its providers’ peer IDs. Each peer maintains a provider store with provider advertisements. A client who wishes to discover the peers who provide a given CID should query the server peers’ provider store for the target CID. (This query occurs simultaneously with the aforementioned routing query, as we describe later.) Importantly, through this query, the server peer learns the target CID that the client wishes to access. We describe IPFS provider advertisements in ??.

Retrieving a content block. Once a client determines the provider peer for a target CID and contacts the peer, it can proceed to retrieve the content block. Content retrieval in IPFS is accomplished using Bitswap [DDP21], a content exchange protocol for retrieving a content block with a given CID from a connected peer, given their routing information. It is likely that the same provider peer stores all of the content blocks of a file. A client would then repeatedly query the same peer to reassemble the file. We detail the Bitswap protocol in ?? and note that the server learns the target CID that the client wishes to fetch as well as the content block that it returns.

4.2.2 Private Information Retrieval

We recall, from ??, that Private information retrieval (PIR) protocols allow a client to retrieve an item from a table, held by one or more servers, such that the servers do not learn which item is retrieved. We refer the reader to that section for a discussion on PIR. We adapt the following definition of a single-server PIR scheme from Henzinger et al. [HHC+23].¹ We define the correctness and security properties for PIR schemes in ??.

Definition 4.2.1 (PIR scheme). A single-server PIR scheme is a triplet of algorithms (Query, Response, Extract) over record space $\mathcal{D}$ and a table with n rows, satisfying the following:

- $\text{Query}(i) \rightarrow (\text{st}, \text{qu})$: takes as input an index $i \in [n]$ and outputs a client state st and a query qu .
- $\text{Response}(D, \text{qu}) \rightarrow \text{ans}$: given the table D and query qu , outputs an answer ans .
- $\text{Extract}(\text{st}, \text{ans}) \rightarrow d$: given client state st and an answer ans , outputs a record $d \in \mathcal{D}$.

We begin with outlining symmetric PIR, which extends PIR with an additional property. We also classify PIR schemes depending on whether they are designed for tables or key-value stores. Within DHTPIR, we addressed the problem of private content retrieval and private peer routing through information-theoretic PIR (IT-PIR), which requires multiple, non-colluding servers. We discuss challenges in directly incorporating the DHTPIR scheme, and specifically, multi-server PIR, within IPFS; these challenges led to our choice of single-server computational PIR (CPIR) schemes. We conclude with our design goals and challenges in integrating existing single-server CPIR schemes for our use cases.

¹We observe that the functions in this definition are similar to those in DHTPIR; Query corresponds to PIRGETQUERYVECTORS, whereas Response maps to PIRPERFORMQUERY and Extract maps to PIRRECREATEFILE. However, the arguments differ since DHTPIR involves multiple servers in its PIR schemes, whereas Peer2PIR builds on top of single-server PIR schemes, along with other semantic differences.

Symmetric PIR. While PIR schemes ensure that the server not learn which record is retrieved, SPIR schemes additionally guarantee that the client learns only the record for which they query. This is similar to oblivious transfer (OT); however, SPIR schemes aim to have sublinear communication cost. We avoid OT schemes with high communication cost, and instead use SPIR schemes.

IndexPIR vs. KeywordPIR. PIR schemes are often tailored for tables, wherein the client knows the index of the desired row; such schemes are known as IndexPIR schemes. Whereas, KeywordPIR schemes enable a client to retrieve a value from a key-value store, given the key. KeywordPIR schemes privately match keywords, and thus incur communication overhead over IndexPIR schemes [MK22; AAAG22]. Since modern DHTs involve key-value stores, we could use KeywordPIR schemes. However, we can also an IndexPIR scheme, to conduct PIR over key-value stores with an extra round of communication and a small probability of error. The server constructs a hash table, mapping each key to an index in the table using a hash function. It places the associated value into the indexed row of the hash table and sends the chosen hash function to the client. The client, using the hash function and its desired keyword, calculates the index for its desired row in the hash table and then uses IndexPIR to retrieve that row. We adopted this approach in DHTPIR, wherein the server sends a perfect hash function [BPZ13] to the client in our file retrieval scheme (??). While the perfect hash function eliminates errors, namely, multiple keywords being mapped into the same index, this approach nevertheless adds an extra round of communication. We exploit the fact that the CIDs themselves include hashes, to enable the client to compute the index for a CID, which lies within the server’s hash table, without an extra round.

Challenges in incorporating DHTPIR into IPFS. Directly incorporating DHTPIR into IPFS requires realizing IT-PIR. We find two main challenges in operationalizing IT-PIR within IPFS, namely forming quorums and requiring peers in quorums to share content. First, in DHTPIR, we use the Commensal Cuckoo Rule (CCR) [SF12], as outlined in ??, to form quorums that satisfy the non-collusion assumption for IT-PIR. However, CCR requires peers to repeatedly leave and join the network until they join a sufficiently large quorum. Concretely, the CCR evicts an average of k peers per quorum during its secondary join step, and thus, in expectation, for each new node joining the DHT, the CCR induces node churn by a factor of k . Sen and Freedman parameterize $k \in \{4, 8, 12\}$, and as such, CCR drastically increases node churn. We conclude that it was infeasible to incorporate CCR into IPFS while minimizing impact on the node churn.

Second, DHTPIR and IPFS operate in different storage models. In DHTPIR, each peer stores content that is related to its peer ID (??). For example, a peer stores all content blocks whose CIDs, when truncated, are equal to the peer’s own peer ID. However, in IPFS, peers decide which content blocks to store. IT-PIR requires peers to provide more storage, by a factor of the number of peers within each quorum. For example, for an average quorum size of 25 peers, as discussed in ??, each peer in the quorum would need to provide approximately 25 times more storage. Moreover, the amount of storage that each peer needs to provide depends on the maximum storage provided by any single peer. As we discuss in ??, while most peers in IPFS store few content blocks, a few peers in the network store thousands of content blocks; we refer to these as heavyweight peers. All peers that are within the quorum of a heavyweight peer would need to provide as much storage as this peer. As such, it is unreasonable for peers in IPFS to provide up to 1000 times or even 25 times more storage. Moreover, every time a peer changes quorums, they would also need to copy content from other peers in their destination quorum, which introduces a large communication overhead on top of the aforementioned node churn. Due to these challenges, we opt for single-server PIR schemes over multi-server PIR schemes. Since single-server information-theoretic PIR schemes are impossible [CKGS98], we use single-server computational PIR schemes.

Challenges in incorporating CPIR into IPFS. Integrating computational PIR schemes towards our use cases in IPFS is also non-trivial. First, the key-value stores we are concerned with have varying numbers of records and record sizes. While the routing table is upper-bound in size, the number of provider advertisements and content blocks on each peer follows a long-tail distribution, as we discuss in ??. Our peer routing use case involves a table with a few small rows, whereas our provider advertisements use case targets a large table with many small rows, and our content retrieval use case addresses a content store with many large rows of content blocks. Since computational PIR schemes are typically tailored for one table size, one scheme thus cannot fit all of our three use cases.

Second, we seek to minimize the communication overhead of PIR queries, while maintaining reasonable computational overheads. Our rationale is that peers can compensate for a larger computational overhead than desired, by adding more cores to their machines, whereas improving latency requires upgrading links across the network. Existing computational PIR schemes commonly assume that clients repeatedly query the same server, and thus amortize large communication overheads over many queries. However, by design, across all types of queries that we consider, a peer acting as a

client routinely contacts different peers to retrieve content.² Thus, directly applying state-of-the-art CPIR protocols, when optimizing for communication overhead, is not possible for our use cases. In ??, we analyze existing CPIR protocols, elucidating different types of communication and computation overheads that they incur, and assess them for applicability to each of our use cases.

4.3 Related Work

We recall that related work on private retrieval of content and routing information in DHTs was discussed in ??. In this section, we outline literature on privately retrieving content for each of our three use cases in IPFS, namely peer routing, provider advertisements and content retrieval.

Private peer routing and provider advertisements. The recently proposed Double Hashing approach [Mic23a; Mic22a] aims to improve client privacy when querying for content in IPFS. The approach requires that a client query for a *prefix* of the CID as opposed to the entire CID, thus providing the client with plausible deniability about which content it is querying for. A server returns provider advertisements corresponding to any CID matching this prefix. The client then obtains k -anonymity where k is the number of CIDs with a matching prefix. We provide stronger guarantees to both the client and the server. The client hides the entire CID, instead of just the suffix, from the server. Additionally, the server is guaranteed that the client does not learn other CIDs that the server stores.

Private content retrieval. Daniel et al. [DET23] provide plausible deniability to peers by integrating a gossip protocol into Bitswap, in which peers forward messages on each other’s behalf. While their work obscures the source of a query, we protect the content of a query. Daniel and Tschorsch [DT23] incorporate probabilistic data structures and Private Set Intersection (PSI) algorithms to protect the content *discovery* step of Bitswap, while leaking the target CID during the content *retrieval* step. These protocols either leak the entire list of CIDs to the client or require it to run computationally expensive operations. Keyword search schemes for IPFS enable a client to retrieve a file, given a

²For the use case of content retrieval, the reader may recall from ?? that the same server peer may provide multiple content blocks of the same file. However, if this provider peer went offline, the client may need to contact multiple peers for retrieving all blocks of this file. In addition, the client will still need to contact different peers for different files.

TABLE 4.1: Privacy properties offered by Peer2PIR across all three functionalities in IPFS, namely peer routing, provider routing and content retrieval. By client privacy, we mean that the content of the client’s query (Peer ID or CID) is hidden from the server. By server privacy, we mean that other than the content requested by the client, remaining contents of the server’s table (routing table, provider advertisement store or content store), are hidden from the client.

Property	Peer routing (Over Peer ID)	Provider routing (Over CID)	Content retrieval (Over CID)
Client privacy	✓	✓	✓
Server privacy	✗	✓	✓

representative keyword rather than its CID [CL22]. They similarly reveal the keyword to server peers.

4.4 Threat Model

We consider a peer acting as a client and an adversarial peer acting as a server. We aim to hide the *content the client queries for and retrieves* from the server(s), across three types of queries.

- Peer routing queries: the client fetches routing information for a *target peer ID* from a server.
- Provider advertisement queries: the client fetches which peers provide a *target CID*, as well as the routing information to reach these provider peers.
- Content retrieval queries: the client fetches a block from a server peer, given its *target CID*.

We hide the target peer ID from each intermediate server peer during peer routing queries. We hide the CIDs during provider advertisement queries and content retrieval queries. Additionally, the server peer cannot determine these identifiers based on its response to a client’s query. We strictly improve over the current IPFS implementation, by hiding these identifiers, as IPFS currently only encrypts communication between peers at the transport layer [Pro23c]. We summarize the privacy properties provided by Peer2PIR across all three functionalities in Table ??.

We assume that all peers within the network are honest but curious, meaning that they correctly follow our prescribed protocols but may passively observe their execution

to break the aforementioned guarantees. An honest-but-curious peer acting as a server may try to determine what a client has queried or what has been returned in response. We rely on the following hardness assumptions in our work: the decisional composite residuosity assumption for the Paillier cryptosystem [Pai99] and the Ring Learning With Errors (RLWE) assumption [LPR10]. We briefly outline non-goals before contextualizing our guarantees for provider advertisements and content retrieval.

Non-goals. We do not aim to protect the client’s network identity. Instantiating a distributed file system such as IPFS over Tor onion services to provide client anonymity is outside of our scope. We do not protect against a network adversary that observes the client. Based on the volume of (encrypted) traffic to other IPFS peers, such an adversary can narrow down the search space of possible target peer IDs or CIDs that the client is looking for. We do not protect against such traffic correlation attacks. Similarly, peers who are acting as servers can not collude to share the fact that they received a request from a given client (peer ID), as they can then conduct this attack. We do not protect content *publishing* requests. In other words, peers know that a server peer is storing a given CID as it inserts provider advertisements, or they may simply query the server for the content block. We do not build post-quantum schemes; the current IPFS implementations use only classically secure protocols for communication between peers. Since CPIR schemes add significantly more computational overheads for the server than for the client, a malicious client can exploit our CPIR-based schemes to amplify a DoS attack against any peer. Standard rate-limiting based on the client’s peer ID and network identity can be used to deter such DoS attacks.

Peer routing. Servers may cause an honest client to experience a restricted view of the network through an eclipse attack. A server peer may respond incorrectly to a client’s routing request, causing it to be directed to other malicious server peers. Prünster et al. [PMZ22] recently demonstrated an eclipse attack against IPFS, which has since been fixed. Importantly, the client’s routing table serves as feedback to a server peer in conducting their attack. While we do not prevent eclipse attacks, we do not make IPFS peers more vulnerable to these attacks, e.g. by releasing a client’s routing table.

Provider advertisements and content retrieval. We provide *server privacy* for these two use cases. We ensure that a client can only retrieve a provider advertisement or a content block if it knows the corresponding CID. In other words, the client should not be able to retrieve information stored by a server peer corresponding to CIDs that

are unknown to the client. Separately, we observe that the privacy guarantees afforded to a client depend on the amount of content stored by each server peer it encounters. The amount of provider advertisements stored by each peer in IPFS follows a long tail distribution [Tra22b], as is typical for DHTs. So it is highly likely that a client peer will query a heavyweight content provider for both provider advertisements and content blocks, and its target CID will be indistinguishable from millions of others. Assuming that a given peer stores files of different sizes, the number of content retrieval queries made by the client would leak which file was being retrieved. We cannot prevent both of these attacks without changing how content blocks of a file are duplicated in IPFS.

4.5 Private Peer Routing

In this section, we focus on how a server answers a client’s query for a target peer ID without revealing it to the server. We first outline the routing process used by the Kademlia DHT, highlighting how a server uses its routing table. We then propose an algorithm to adapt the routing table to enable hiding target peer IDs from the server and demonstrate the correctness of our algorithm. Finally, we integrate a PIR scheme to privately retrieve a row from our modified routing table, leading to a private peer routing algorithm.

4.5.1 Kademlia and the IPFS Routing Table

Kademlia DHT. DHTs map both the content identifiers and peer identifiers to a common virtual address space, through a collision-resistant one-way hash. IPFS uses SHA-256 to form the virtual address space for its underlying Kademlia DHT, and thus peer identifiers are 256 bits long. Kademlia measures *distance* between two addresses by computing an XOR between them and interpreting the result as an integer. For example, the addresses `0b00011` and `0b00101`, in an example 5-bit address space, are `0b00110=6` units apart. They share the first 2 bits out of 5 and are said to have a *common prefix length* (CPL) of 2. Addresses with higher CPLs are closer to one another.

Routing table structure. The routing table (RT) is indexed by the CPL. When a peer is to be added to the RT, the peer first computes the CPL between itself and the new peer. Each row of the RT contains multiaddresses for at most k peers, whose CPL is given by the row index. In other words, higher indices of the RT contain multiaddresses to closer peers. In IPFS, each row (also called *bucket*) can have up to $k = 20$ peers’ multiaddresses. When a peer first joins the network, it contacts a set of bootstrapping

peers to populate its RT. Whenever a peer comes into contact with another peer in the DHT, it may create a new bucket for the new peer or add an entry to an existing bucket, depending on the CPL between the two peers. Thus, buckets can be created statically, that is, all $L = 256$ buckets are initialized to empty buckets, where L is the length of the peer IDs in bits, and then these buckets fill up over time.³ In contrast, buckets can also be created dynamically; that is, we start with one bucket, and then split it, depending on the peer ID of the newly encountered peers, in order to create more buckets. Specifically, a bucket with index i is created only when the peer has met at least k additional peers whose CPL is greater than or equal to i . In ??, we contrast how the RT is populated when buckets are created statically versus dynamically. As we illustrate below, there are two differences between buckets created statically and dynamically: first, we have fewer buckets as expected, and second, the last bucket may have peers whose CPL is *greater than* the bucket index. Kademlia prefers long-lived peers, in that if a peer's RT already has k peers whose CPLs equal to the CPL of a new peer, then this new peer is not stored in the RT. Irrespective of whether the buckets are created statically or dynamically, the RT can have buckets of unequal sizes.

Example 4.5.1. Consider a peer whose address is `0b00000` encounters three peers with addresses `0b00010`, `0b00101` and `0b10100` consecutively. Suppose that our $k = 2$; that is, each bucket can have at most two peers. These three peers' CPLs with our given peer are three, two and zero respectively. If buckets were created statically, then we would have five buckets and each peer would simply be added to a bucket whose index is given by the CPL. When buckets are created dynamically, `0b00010` is added to the initial bucket, that is, the bucket with index zero. The next peer, *irrespective of its address*, is also added to the same bucket, since this bucket has fewer than k peers whose CPL is greater than or equal to the bucket index (zero). However, we cannot add the next peer to this bucket since it is now full; again, this is true irrespective of the peer's address. Thus, we split the last bucket into two buckets, and distribute its peers across the two buckets, such that the bucket with the *lower* index only has peers whose CPLs equal that bucket index, while the bucket with the *higher* index has all other peers. Thus, bucket zero is initially empty, while bucket one now has peers `0b00010` and `0b00101`. We continue to split the last bucket until it has fewer than k peers, introducing new empty buckets if necessary. So, in the second split, peers `0b00010` and `0b00101` are moved from bucket one to bucket two. In the third split, peer `0b00101` is kept in bucket two but peer `0b00010` is moved to bucket three. No further splits are executed since the last bucket, namely bucket three, has only one peer. Thus, all buckets, other than

³For our private provider advertisements algorithm, we require $L \geq 2\lambda$ where λ is the security parameter, as discussed in ??.

the bucket with the highest index, only have peers whose CPLs match their bucket indices. The new peer, namely `0b10100`, has a CPL of zero, and we can now insert it into bucket zero, as desired. In this example, peers are assigned to identical buckets when the buckets are created dynamically and statically, with the only difference being that in the dynamic case, we only have four buckets instead of five.

Original Kademlia lookup. We describe how a server peer uses its RT in Kademlia, in order to answer a client peer's query for a target peer ID. If the bucket in the RT given by the CPL between the target and the server's peer IDs is full, then the server returns that bucket. Otherwise, it selects peers from other buckets to return a set of k closest peers. It does so by computing the distance between the target peer ID and other peer IDs in the RT to select the k closest peers, as detailed below in ???. As long as the RT has at least k peers in total, the server returns k peers' multiaddresses for every target peer ID. The client then uses the multiaddresses to contact these peers, as follows. The client initializes and maintains a list of the top k peers that it has not queried yet. A concurrency parameter in Kademlia ($\alpha = 3$) dictates how many server peers are queried concurrently in a round, after querying the initial peer. If a round fails, that is, all (up to $k \cdot \alpha$) responses received from these α server peers do not return a peer that is closer than the closest peer seen, then the client queries the remaining peers from its list. The client thus iteratively queries different server peers, until it reaches the target peer. Each peer refreshes the buckets within its routing tables periodically. The peer samples a peer ID at random within the range of virtual addresses spanned by that bucket, and conducts a peer routing query for that address. Maymounkov and Mazières establish that after refreshing the buckets, every bucket will either contain k valid nodes or else contain every node in its range if fewer than k exist [MM02].

Challenges in hiding the target peer ID. Our goal is to enable the server to respond to the client's query, while hiding the target peer ID. Suppose that the server only learns the bucket index, namely the CPL between its own peer ID and the target peer ID. (We ultimately hide this index as well, via PIR, as we discuss later.) If the server's bucket for this CPL is full, then it simply returns this bucket. However, if it is not full, then only knowing the CPL is insufficient to complete the original Kademlia lookup. In particular, the suffix of the target peer ID is used to compute distances to peers in other buckets, and select the closest ones. A naive approach might return the bucket corresponding to the CPL, even if it is not full; however, non-full buckets would be insufficient for the client to reach the target peer. To solve the problem of non-full buckets, the server can compute the k closest peers for every target peer ID. This approach blows up the RT,

which originally has at most 256 rows, to 2^{256} rows. We propose *normalizing* the RT, that is, ensuring that each bucket has exactly k peers, which are closest to any target peer ID whose CPL is given by the bucket index.

Before we describe our normalization algorithm, we study the original Kademlia lookup. The routing response for a given target peer ID can be justified by simply computing the distances of all peer IDs in the RT to the target peer, and picking the k closest peers. However, to motivate our private peer routing algorithm, we examine when Kademlia *needs* the entire target peer ID, including the least significant bits (LSBs), to compute the closest peers, or when simply knowing the CPL suffices.

Example 4.5.2. Continuing ??, suppose that our server peer (0b000000) encounters new peers, and then receives a query for a target peer ID 0b001111. In ??, we consider five cases of different routing tables maintained by the server peer, and we depict the routing response for this target peer ID, under Kademlia as well as Peer2PIR, for each case. The CPL of the target peer ID with the server, namely t , is two, and in all five cases, the bucket at index two in the routing table is *not* full, and thus, Kademlia must choose peers from other buckets in order to respond back with $k = 3$ peers. Given *only* the CPL between a target peer ID and the server, its $t + 1$ -th bit would be the opposite of that of the server, and thus the server knows the $t + 1$ -bit prefix of the target peer ID; in our example, given that $t = 2$, the server knows that the target peer ID would be of the form 0b001XX. We go through each case below. We denote a bucket at index i by B_i .

In the first case, there are only two peers in B_2 , and so, Kademlia could either pick a peer from a lower (farther) bucket (0–1) or a higher (closer) bucket (3–4). Now, a peer in B_t would share a common prefix of length at least $t + 1$ with other peers in the same bucket. The CPL between a peer in B_t and peers in *closer* buckets ($B_j | j > t$) is always t , whereas its CPL with peers in *farther* buckets ($B_j | j < t$) is given by the index of the farther bucket (j). For example, consider our target peer ID 0b001111, which lies in B_2 of our server peer and thus $t = 2$. Its CPL with peers in B_3 and B_4 is of length two, whereas its CPL with peers in B_1 and B_0 is one and zero respectively. Thus, a peer in a given bucket is closer to peers in buckets that are closer to the server and farther from peers in buckets that are farther from the server. So, the server would prefer filling a bucket with peers from closer buckets over farther ones. Indeed, in our example, the target peer is further away from the peer in bucket zero, which is at a distance of 0b11101 = 29, than the peer in bucket three, which is at a distance of 0b00101 = 5.

The second case demonstrates that when the target bucket is empty and the closer buckets (3–4) contain exactly k peers, Kademlia prefers to pick peers from closer buckets over farther buckets (0–1). So far, the server peer can compute the routing response

TABLE 4.2: Examples of Kademlia and Peer2PIR routing responses (C_1 and C_2 respectively) for the target peer ID $p = 0b00111$, under various routing tables held by the server peer $0b00000$. The routing response should contain $k = 3$ peers. We also show the CPL and the distance between each peer in the respective routing response and the target peer ID, for both Kademlia and Peer2PIR. We emphasize that the CPLs are identical for both Kademlia and Peer2PIR routing responses. We underline the peer IDs in the routing response for Peer2PIR that differ from the one in Kademlia and also underline their distances from the target peer ID.

	Bucket no.	Case 1	Case 2	Case 3	Case 4	Case 5
	0 (0b1XXXX)	0b11010				
	1 (0b01XXX)	$\emptyset$			0b01010 0b01110 0b01100	
	2 (0b001XX)	0b00101 0b00100	$\emptyset$	0b00101 0b00100	0b00101	0b00101 0b00100
	3 (0b0001X)	0b00010	0b00011 0b00010		$\emptyset$	
	4 (0b00001)	$\emptyset$	0b00001		$\emptyset$	
Kademlia	Routing response C_1	0b00101 0b00100 0b00010	0b00011 0b00010 0b00001	0b00101 0b00100 0b00011	0b00101 0b01110 0b01100	0b00101 0b00100 0b01110
	CPL	{3, 3, 2}	{2, 2, 2}	{3, 3, 2}	{3, 1, 1}	{3, 3, 1}
	Distance(C_1, p)	{2, 3, 5}	{4, 5, 6}	{2, 3, 4}	{2, 9, 11}	{2, 3, 9}
Peer2PIR	Routing response C_2	0b00101 0b00100 0b00010	0b00011 0b00010 0b00001	0b00101 0b00100 <u>0b00010</u>	0b00101 0b01110 0b01100	0b00101 0b00100 <u>0b01100</u>
	CPL	{3, 3, 2}	{2, 2, 2}	{3, 3, 2}	{3, 1, 1}	{3, 3, 1}
	Distance(C_2, p)	{2, 3, 5}	{4, 5, 6}	{2, 3, <u>5</u> }	{2, 9, 11}	{2, 3, <u>11</u> }

using only its CPL with the target peer ID; the LSBs of the target peer ID were not required.

The third case differs from the second case in that the closer buckets, combined with the target bucket, have more than k peers. Here, the server peer does need the LSBs in order to pick the remaining peer(s) from these closer buckets. Importantly, we highlight that the closest peer(s) to any given target peer ID may lie in a bucket not adjacent to the target bucket. While in this example, the peer in B_4 (0b00001) is further from the target than the peers in B_3 , for a different target peer ID with the *same* CPL of two, such as 0b00101, 0b00001 is closer than the peers in B_3 .

Considering the fourth and fifth cases, the closer buckets are empty and thus the server peer must pick one or more peers from farther buckets (buckets 0–1). We further

split peers in the farther buckets, into subbuckets, based on a $t+1$ -bit prefix of their peer IDs. We can split B_1 into subbuckets based on a three bit prefix: subbucket B_1^0 contains peers of the form $0b010XX$, that is, $B_1^0 = \{0b01010\}$, whereas subbucket B_1^1 contains peers of the form $0b011XX$, that is, $B_1^1 = \{0b01110, 0b01100\}$. We can now order the aforementioned subbuckets based on their peers' distance from the $t+1$ -bit prefix of the target peer ID ($0b001XX$). Subbucket B_1^1 is closer than B_1^0 to *any* target peer ID in bucket two. Peers in B_1^1 would have a distance of $0b011XX \oplus 0b001XX = [8, 12)$ from any peer in bucket two, whereas peers in B_1^0 would be at a distance of $0b010XX \oplus 0b001XX = [12, 16)$. In the fourth case, since our target bucket has only one peer, we need $k - 1 = 2$ peers from the farther buckets, and we thus return with all peers in subbucket B_1^1 . In the fifth case, our target bucket has two peers, and we only need one more peer from the farther buckets. We now use the LSBs of the target peer ID to pick the closest peer from subbucket B_1^1 , namely $0b01110$, which is at a distance of 9, as compared to $0b01100$, which is at a distance of 11. A bucket whose CPL with the server i is greater than the CPL of t , also has a CPL of i with peers in bucket t , as described above. Thus, we observe that buckets that are farther from the server peer, are also farther from the target bucket. When picking peers from buckets whose indices are greater than t , Kademlia picks them strictly in order of decreasing bucket index. Consequently, for the fourth and fifth cases, Kademlia prefers picking peers in bucket 1 over peers in bucket 0. We thus find that while knowing the CPL is sufficient for some cases (1, 2 and 4), in other cases (3 and 5), the LSBs of the target peer ID are required to find the closest peers to it.

From ??, we glean the following for our private peer routing algorithm:

1. When a target bucket has fewer than k peers, Kademlia chooses peers in closer buckets, that is, buckets corresponding to higher CPLs with the server, over peers in lower buckets, as follows.
 - 1a If the closer buckets and the target bucket together have at most k peers, Kademlia outputs all of these peers.
 - 1b Otherwise, Kademlia chooses peers from *all* closer buckets, using the LSBs of the target peer ID to compute peers at the smallest distances.
2. If the server still does not have k peers for the target peer ID, then Kademlia chooses peers from each successively farther bucket, as follows.
 - 2a The server partitions peers in this bucket into subbuckets, based on their $t+1$ -bit prefix, and orders these subbuckets in terms of their distance to the $t+1$ -bit prefix of the target peer ID. Starting with the closest subbucket, Kademlia

chooses all peers from a subbucket one at a time. If doing so would result in an output of more than k peers, Kademlia does not include that subbucket and proceeds to the next step.

- 2b Kademlia uses the LSBs of the target peer ID to choose the remaining peers from *within* a subbucket.

Now, if the server peer only knew its CPL with the target peer ID and did *not* know the LSBs of the target peer, then steps ?? and ?? proceed as usual. However, the best that the server can do for steps ?? and ??, is to pick at random. With this in mind, we present an algorithm for normalizing the RT buckets in ?? and describe it next.

4.5.2 Routing Table with Normalized Buckets

The client wishes to retrieve a target bucket indexed by t , where t is the CPL between the target and the server's peer IDs. As in the original lookup algorithm, if the RT has at most k peers *in total*, all peers are returned (Line ??). Since buckets are created dynamically, the RT may have $r < t$ buckets, in which case the server returns the (normalized) last bucket, containing the closest k peers that it knows (Line ??). If the target bucket is full, it is returned directly as normalization is not required (Line ??).

Reiterating Step ??, the server prefers filling a bucket with peers from closer buckets over farther ones (Line ??). Importantly, without knowing the suffix of the target peer ID, the server cannot order peers in these closer buckets in terms of their distance to the target. So, the server picks the remaining number of peers at random from all closer buckets, taking all closer peers if the total is less than k (Lines ??–??). If bucket t remains non-full, the server then adds farther buckets one at a time (Lines ??–??). If the server cannot add a bucket completely, then it partitions peers in the bucket into subbuckets, based on a $t + 1$ -bit prefix of their peer IDs, as illustrated in ?. The server then adds each subbucket one at a time (Line ??–??), selecting peers at random if it cannot include the entire subbucket (Line ??). Importantly, we note that the CPLs of each peer returned for bucket t are identical for both the original Kademlia lookup, as well as for a lookup on a normalized routing table; the reader may verify this for all cases in ?.

In practice, we fix a seed for randomness in our implementation, thus making the selection of peers within a particular bucket actually deterministic. This has no effect on the efficiency of the algorithm given that without knowing the target peer ID, all peers within that bucket are equally likely to be a given distance from the target peer. Using a fixed seed also has no effect on the privacy of the client since the seed does not rely on the target peer ID.

Algorithm 12 Routing Table Normalization Algorithm

```
1: For  $i \in [0, r = \text{len}(\text{RT})]$ , bucket  $B_i$  corresponds to CPL  $i$ .
2: Input: Target index  $t$ 
3:  $R \leftarrow \cup_{i=0}^r B_i$ 
4: if  $|R| \leq k$  then  $\triangleright$  RT has less than  $k$  peers
5:   return  $R$ 
6: if  $r < t$  then  $\triangleright$  If  $r < t$ , return last bucket
7:    $t \leftarrow r$ 
8:  $R \leftarrow B_t$ 
9: if  $|R| == k$  then
10:  return  $R$ 
11: else  $\triangleright R$  is not full ( $|R| < k$ )
12:   if  $t < r$  then  $\triangleright$  Closer buckets can be used
13:      $C \leftarrow \cup_{i=t+1}^r B_i$ 
14:     if  $|C| \leq k - |R|$  then
15:        $R \leftarrow R \cup C$ 
16:     else append  $k - |R|$  peers from  $C$  to  $R$ , at random.
17:   if  $|R| < k$  then  $\triangleright$  Must go through farther buckets
18:      $\ell \leftarrow t - 1$ 
19:     while  $|R| + |B_\ell| \leq k$  do
20:        $R \leftarrow R \cup B_\ell$ 
21:        $\ell \leftarrow \ell - 1$ 
22:      $\nabla$  PARTITIONBUCKET sorts peers in  $B_\ell$  into  $2^{t-\ell}$  subbuckets. Peers in the same
    subbucket have the same  $t + 1$ -bit prefix.
23:      $\{B_\ell^j | j = \{0, 1, \dots, 2^{t-\ell}\}\} \leftarrow \text{PARTITIONBUCKET}(B_\ell, t)$ 
24:      $m \leftarrow 0$ 
25:     while  $|R| + |B_\ell^m| \leq k$  do
26:        $R \leftarrow R \cup B_\ell^m$ 
27:        $m \leftarrow m + 1$ 
28:     append  $k - |R|$  peers from  $B_\ell^m$  to  $R$ , at random.
29: return  $R$ 
```

A peer routing algorithm is said to *converge* when the client peer ultimately contacts the target peer. We use the term *convergence* to refer to the expected number of hops needed to reach a target peer. Our peer routing algorithm uses the target CPL to look up the normalized RT. It is possible that the client peer would require more hops to converge when using our algorithm, in comparison to the original Kademlia lookup.

We bound the difference in the number of hops taken by Peer2PIR and Kademlia to converge.

4.5.3 Convergence under Normalized Routing Tables

We prove and experimentally verify that our normalized algorithm does not significantly increase the number of hops required to reach a target peer. At a high level, the argument follows from the fact that the only cases in which the normalized algorithm diverges from the original algorithm still provide the *same guarantee* of closeness to the target peer. In the following lemma, KADEMLIA refers to the original Kademlia routing algorithm and PRIV-KADEMLIA refers to our normalized routing algorithm.

Proposition 4.5.1 ([MM02]). In the original (non-normalized) routing table, every bucket will either contain k valid nodes or else contain every node in its range if fewer than k exist.

Proof. We defer the reader to their paper for their discussion. We remind the reader that each peer refreshes its buckets periodically (??). In short, Maymounkov and Mazieres argue that the aforementioned invariant is true as long as k is chosen to be appropriately large, such that the likelihood of any k peers all failing between two consecutive refreshes of any bucket is negligible. $\square$

Proposition 4.5.2. Given a target peer ID t , a peer ID p chosen uniformly at random and an integer c^* with $0 \leq c^* < L$ then the probability that $\text{CPL}(p, t) = c^*$ equals the probability that $\text{CPL}(p, t) > c^*$, both of which are given by $q = 2^{-(c^*+1)}$, whereas the probability that $\text{CPL}(p, t) < c^*$ is given by $1 - 2q$.

Proof. We consider the probability that any peer ID chosen uniformly at random will have a CPL of exactly c^* . The first c^* bits of the peer ID would be identical to that of t with a probability 2^{-c^*} . The next bit of the peer ID is equally likely to match t (and thus have a CPL larger than c^*) or to mismatch t (and thus have a CPL equal to c^*). Thus, the peer would have equal probability $q = 2^{-(c^*+1)}$ of having a CPL equal to c^* or larger than c^* . The probability that the peer would have a CPL smaller than c^* is then simply given by $1 - 2q$. $\square$

Corollary 4.5.1. For a network with n peers, consider a server peer p and a target peer t , such that $\text{CPL}(p, t) = c^*$. Then the probability that there are m peers across the network whose CPL with t is c^* , given that bucket B_{c^*} within the routing table of p is empty, is given by $\binom{n}{m} \cdot q_1^m \cdot (1 - q_1)^{n-m}$ where $q_1 = 1/(2^{c^*+1} - 1)$.

Proof. Since the peers in bucket B_{c^*} have a CPL larger than c^* with peer t , we wish to compute the probability that there are m peers whose CPL with t is exactly c^* , given that there are no peers whose CPL with t is larger than c^* . This conditional probability is equal to the ratio of the probability that there are m peers of CPL exactly c^* and zero peers of CPL larger than c^* , to the probability that there are zero peers of CPL larger than c^* .

If there are zero peers whose CPL with t is larger than c^* , then the CPL of all n peers with t is less than or equal to c^* . By Proposition ??, the probability that there are n peers whose CPL with t is less than or equal to c^* , is simply given by $(1 - q)^n$, where $q = 2^{-(c^*+1)}$ as defined in that proposition. Applying the same proposition, we note that the probability that a given set of m peers has a CPL of exactly c^* , is q^m , and the probability that the remaining $n - m$ peers have a CPL that is less than or equal to c^* , is $(1 - 2q)^{n-m}$. There are $\binom{n}{m}$ ways of choosing these m peers, and thus, the probability that there are m peers with a CPL of exactly c^* and the remaining $n - m$ peers with a CPL that is strictly less than c^* , is simply $\binom{n}{m} \cdot q^m \cdot (1 - 2q)^{n-m}$. Thus, the desired probability is given by:

$$\frac{\binom{n}{m} \cdot q^m \cdot (1 - 2q)^{n-m}}{(1 - q)^n} = \binom{n}{m} \cdot \left(\frac{q}{1 - q}\right)^m \left(\frac{1 - 2q}{1 - q}\right)^{n-m}$$

We denote $q/(1 - q) = 1/(2^{c^*+1} - 1)$ by q_1 and obtain the desired result. $\square$

Theorem 4.5.2 (Convergence of Private Routing). Defining convergence as the expected number of hops to reach the closest peer to a target peer ID t , PRIV-KADEMLIA converges in at most a single expected additional hop over KADEMLIA.

We note that KADEMLIA converges in $\Theta(\log_2(n))$ hops [MM02, Section 3], and therefore the theorem implies that PRIV-KADEMLIA does as well.

Proof. We denote the set of all peers that have a CPL of c with a given peer p by $S(c, p)$; that is, $S(c, p) = \{q \mid \text{CPL}(p, q) = c\}$. Restating Proposition ??, when querying any peer p for a target ID t , such that $\text{CPL}(p, t) = c$, as long as $S(c, p) \neq \emptyset$, this proposition guarantees that bucket B_c within the RT of peer p would be non-empty and $B_c \subseteq S(c, p)$ with $|B_c| \leq k$. In other words, B_c will contain at least one peer as long as $S(c, p) \neq \emptyset$.

Suppose that the closest peer, in the entire network, to the target is τ and $\text{CPL}(\tau, t) = c^*$. Since τ is closer than any other peer to t , then for a peer p such that $\text{CPL}(p, t) = c < c^*$,

we observe that peer τ would share the first c bits with peer p , and its $c + 1$ -th bit would be the same as that of t and would differ from that of p . Thus $\tau \in S(c, p)$. Since $S(c, p) \neq \emptyset$, the precondition for Proposition ?? holds true, and for any peer p , its bucket B_c will be non-empty. When the target bucket is non-empty, we observe that the output of PRIV-KADEMLIA includes all peers in that bucket (Line ??), and specifically, the closest peer output by PRIV-KADEMLIA and KADEMLIA would be identical. Again, since any peers in the c -th bucket have a CPL of at least $c + 1$ with peer t , this closest peer output in each hop will be closer to t than the queried peer p . Then as long as $CPL(p, t) = c < c^*$ the client contacts an identical sequence of peers, irrespective of whether it uses PRIV-KADEMLIA or KADEMLIA. The CPLs of the peers in this sequence, with the target peer, increase by at least one bit in each hop, and thus the distance from these peers to the target reduces in each hop. This process continues until the client reaches a peer θ such that $CPL(\theta, t) = CPL(\tau, t) = c^*$. If t is online, then, by definition, the closest peer in the entire network to the target is $\tau = t$ and additionally, we have that $\theta = t$, thus $c^* = CPL(\tau, t) = CPL(t, t) = L$. Thus, both PRIV-KADEMLIA and KADEMLIA would contact an identical sequence of peers, and reach the target in an identical number of hops.

We now consider the case that there is no online peer with ID t . When the client searches for a peer that hosts a provider advertisement for a CID, the target peer ID t is given by the hash of the CID. Thus, it is likely that no peer exists in the network whose peer ID exactly matches the hash of the CID. Nevertheless, there will always exist an online peer τ whose peer ID is closest to t , and let $c^* = CPL(\tau, t)$. Then bucket B_{c^*} at peer θ would be empty, since any peer in it would have its CPL with t be strictly larger than c^* . Let $CPL(\tau, \theta) = j$. We note that $j > c^*$, and thus peer τ would be inserted into a bucket at a higher index than c^* within the routing table of peer θ . We denote the set of all peers that have a CPL of c^* with t by $S(c^*, t)$, and we note that this set is non-empty since $\tau \in S(c^*, t)$. If θ knew of any peer $q \in S(c^*, t)$, then, similar to peer τ , peer q would be inserted into a bucket with a higher index than c^* . We now consider three cases depending on the size of $S(c^*, t)$.

If $|S(c^*, t)| \leq k$, then by Proposition ??, peer θ would indeed know of each such peer $q \in S(c^*, t)$, including τ . Additionally, when the desired bucket B_{c^*} is empty and the higher buckets have at most k peers in total, then PRIV-KADEMLIA includes all peers in the higher buckets in its output (Lines ??–??), just as KADEMLIA would. Thus, the output of both algorithms would include the set $S(c^*, t)$, and thus both algorithms would necessarily converge to τ in precisely one hop from peer θ .

We now consider the case that $|S(c^*, t)| > k$. Let the set of peers across peer θ 's buckets $c^* + 1$ to $L - 1$ be C ; that is, $C = \cup_{i=c^*+1}^{L-1} B_i$. Evidently, $C \subseteq S(c^*, t)$; we denote

$|S(c^*, t)|$ by m . Let C_k denote the subset of size k of C that is output by KADEMLIA. We pessimistically assume that θ knows τ , and thus $\tau \in C_k$, though this need not be the case, so that KADEMLIA will converge to τ in one hop after reaching θ . PRIV-KADEMLIA would select k elements at random from C . When the client queries any node $p \in S(c^*, t)$, it will learn τ with probability $k/|C| \geq k/|S(c^*, t)| = k/m$ since $C \subseteq S(c^*, t)$. Otherwise, the client queries the peer in C that is closest to t , and so on. Thus, the expected number of hops that the client would need to reach τ , after reaching θ , is at most m/k . As mentioned above, we assume pessimistically that KADEMLIA converges in only one hop. Therefore, for a given value of m , the expected number of *additional hops* for PRIV-KADEMLIA over KADEMLIA is simply $m/k - 1 = (m - k)/k$. We observe from Corollary ?? that the probability that there are m peers in $S(c^*, t)$, when bucket B_{c^*} is empty, is given by the following binomial distribution:

$$\text{Binomial}(n, q_1)[m] = \binom{n}{m} \cdot q_1^m \cdot (1 - q_1)^{n-m}$$

where $q_1 = q/(1 - q)$ with q as defined in Proposition ??, so that $q_1 = 1/(2^{c^*+1} - 1)$. Thus, the expected number of *additional hops* for PRIV-KADEMLIA over KADEMLIA is given by:

$$\begin{aligned} Y &= \sum_{m>k} \text{Binomial}(n, q_1)[m] * (m - k)/k \\ &= 1/k * \sum_{m>k} \text{Binomial}(n, q_1)[m] * (m - k) \\ &< 1/k * \sum_{m>k} \text{Binomial}(n, q_1)[m] * m \\ &< 1/k * \sum_m \text{Binomial}(n, q_1)[m] * m \\ &= 1/k * n * q_1 \text{ by the expectation of a Binomial distribution} \\ &= 1/k * n/(2^{c^*+1} - 1) \text{ by the definition of } q_1 \text{ in Corollary ??} \end{aligned}$$

Since in expectation $c^* = \log_2(n)$, and so $2^{(c^*+1)} = 2n$, we conclude that $Y < 1/k * n/(2n - 1) \ll 1$ for $k = 20$, which is used in IPFS. We reiterate that the CPLs of peers returned by PRIV-KADEMLIA have an identical CPL with the target peer, as compared to the CPLs of the peers returned by KADEMLIA. Thus using PRIV-KADEMLIA to conduct peer routing queries to refresh the buckets themselves would continue to uphold the invariant in Proposition ?. $\square$

Experimental analysis. We also experimentally evaluated our normalized routing algorithm against the Trie-based routing algorithm that is currently deployed in IPFS. Our goal is to determine the difference, on average, in the number of hops required to reach a target peer. To do so, we simulated a network based on a crawl of IPFS and then queried for different target peer IDs, tracking the number of hops required by each algorithm.

To capture a meaningful network topology to test our claim, we crawled the active IPFS network using the Nebula crawler [Tra25] and collected an adjacency list of all online peers and their neighbours. Our crawl collected peer IDs from 13692 peers in the network. Only peers' IDs and their routing tables are gathered via this crawler and so no peers' network addresses were revealed. For each peer, we set up a Trie-based RT using the current Kademlia implementation and a normalized RT based on our new algorithm. We generate 5000 target peer IDs at random, while the client peer was fixed for the experiment. For each step of the routing process in IPFS, the client would concurrently query three closest peers it knows to the target peer ID, and retrieve $k = 20$ nearest peers from them. We simulate each step of the routing process as follows; no peers are actively queried. Given the target peer ID, we simply look up the server peer's Trie RT following Kademlia's original implementation. We also look up the server's normalized RT given the target peer's CPL with the server peer. The process is repeated iteratively until the target peer is found.

All queries took an average of two hops to reach the target, with a maximum of six hops. We did not find a difference in the number of hops using the Trie-based and the normalized routing algorithms, that is, both algorithms required an identical number of hops in every iteration of our test. Our experiments verify Theorem ?? and show that for the value of k used in IPFS, normalizing the routing table introduces no difference in the number of hops.

4.5.4 Private Algorithm Integration

Integrating our normalization algorithm into IPFS generates privacy engineering challenges, which we outline and address below. We also justify our choice of PIR to privately retrieve a bucket from the RT. We then integrate our normalization algorithm with PIR to develop a private peer routing solution.

Dynamic networks. IPFS is a dynamic network with high churn, meaning that nodes often join and leave the network. Consequently, clients' RTs are regularly updated in order to reflect new or failing peers. We require that the RT be normalized every time a

new peer is added to it. Since routing tables are relatively small and the normalization algorithm is not computationally intensive, this does not impose any prohibitive overhead on the clients. We discuss the impact of node churn on our private peer routing in ??.

Joining peer IDs and addresses. We have thus far assumed that our RT stores multi-addresses for peers. In practice, the RT in IPFS stores only the *peer IDs*, and an additional table acting as an address book maps these peer IDs to their routing information, that is, *multiaddresses*. This design allows multiaddresses to be stored only once and used for both peer and provider routing. A non-private lookup uses the value fetched from the RT as a key to look up the address book; however, such a lookup is not possible when we privately retrieve buckets from the RT. Therefore, in our implementation, the server joins the two key-value stores, namely the RT which stores peer IDs and the address book which stores multiaddresses, before responding to a private query. This enables PIR to be computed over a single table rather than two. For freshness, this join must be recomputed whenever either store is updated, which is determined by the network churn.

Selecting a PIR scheme. Following normalization and join steps, we must hide the target CPL from the server. The RT is limited in both its number of rows (≤ 256) and the size of each row (multi-addresses for $k = 20$ peers). Under a trivial PIR approach, the server would provide the entire RT to the client, foregoing normalization and join; however, trivial PIR makes the server peer susceptible to eclipse attacks [PMZ22].

An alternative strategy might employ an OT algorithm, which would restrict the client to retrieving only one row and prevent leaking other rows' contents. However, the total communication cost of OT algorithms is often greater than the size of the table. This introduces a trade-off between information leaked to clients about other rows in the routing table and communication cost of a given primitive. Given the fact that IPFS does not restrict a client from repeatedly querying a server for different target peer IDs, or specifically, CPLs that correspond to different rows in the routing table, we argue that the minimal leakage guarantee provided by OT does not justify the communication overhead. Rate-limiting on the server peer can reduce the susceptibility of server peers to eclipse attacks. We thus opt for PIR with rate-limiting over OT.

We design an IndexPIR scheme based on RLWE encryption, namely RLWEPIR, catered to our private routing use case. This scheme and our rationale for choosing it are detailed in Section ??.

PIR integration. A client first computes the CPL between the target peer ID and the server peer ID, which it uses to construct a PIR query for the server. The server uses the PIR query and its normalized routing table joined with multiaddresses, to compute a PIR response, which is returned to the client. The client processes the PIR response, to obtain multiaddresses of k peer IDs close to the target peer ID. Effectively, the use of PIR hides the target CPL from the server.

We note that the server builds the RT dynamically as discussed in ??, and consequently, it typically has $r < 256$ buckets. Our normalization algorithm returns the r th bucket when asked to normalize a bucket for any index $t > r$ (Line ??, as discussed in ??). As an optimization in our implementation, the server can compute a PIR response without instantiating the normalized RT for indices $t > r$. Our algorithms simply take the sum of indices $t > r$ in the encrypted indicator vector, and then compute the PIR response over the normalized RT with r buckets. The client will obtain the correct output for these indices, without the server learning which index was retrieved.

Security. Interactions between the client and the server now consist only of PIR queries and responses, so the security follows directly from that of RLWEPIR, discussed in ??.

4.6 Private Provider Advertisements

We outline the provider advertisement functionality in IPFS, namely, how clients determine which peers are providing the desired content block, and introduce our private alternative. We refer the reader to ?? for relevant notation for Sections ?? and ??.

4.6.1 IPFS Provider Advertisements

A provider advertisement maps a CID c to a set of peer IDs $\{p_1, p_2, \dots, p_j\}$ of peers that provide c , namely provider peers. Each peer maintains a key-value store of provider advertisements, known as a provider store. When a provider peer stores a content block addressed by c , it advertises the block to the IPFS network as follows. The provider peer contacts its k closest neighbours to c , following the aforementioned iterative routing process. (A routing query for a CID operates similarly to that for a peer ID, in that both identifiers get hashed to the SHA-256 virtual address space of the Kademlia DHT, as observed in Section ??.) The provider peer then requests each such neighbour to add a provider advertisement for this CID to their provider store.

TABLE 4.3: Notation for private provider advertisements and content retrieval for Peer2PIR

c	A content identifier (CID) for a content block.
$c[: \ell]$	An ℓ -bit prefix of the CID c .
D	List of content blocks stored at a peer, ordered by the CID of the block.
J	Map of CIDs to their associated indices in the content store.
P	Provider store: maps a CID to a set of provider peers' IDs $\{p_1, p_2, \dots, p_j\}$.
M	Address book: maps a peer ID to its multiaddress.
B	Number of bins for private provider advertisements.
$\text{SE.Enc}(k, x)$	Symmetric encryption of x with key k .
KDF	Key Derivation Function which outputs a key for use by $\text{SE.Enc}(k, x)$.

Consider a client who wishes to discover the provider peers for c . The client first runs a routing query for that CID. This routing query will ultimately lead the client to one of the k closest peers to c . We note that our private peer routing algorithm from Section ?? can be used to determine the k closest peers to the desired CID, without revealing the CID to the server peer in the routing query. As described above, these peers will store the provider advertisement, which would map CID c to the set of provider peers. So, along with the routing query for c , the client must also query these peers' provider store for c . We refer to this latter query as a provider advertisement query.

A server peer processes a provider advertisement query non-privately as follows. The server queries its provider store for c , and fetches the set of provider peers. In practice, the provider store is keyed by the CID concatenated with the provider peer's ID, since a block with a given CID can be provided by multiple peers. Each value in the provider store is the expiry time of the advertisement. Thus, upon receiving a provider advertisement query, a server peer looks up the provider store for keys prefixed by the given CID. The server checks if each such provider advertisement has not expired, based on the value, and if so, extracts the provider peer's ID from the key suffix.

Recall that the server peer also maintains an address book, which maps peer IDs to their multiaddresses. Similar to the peer routing query, to process the provider advertisement query, the server uses the provider peers' IDs from the provider store as a key to the address book, and fetches their multiaddresses. We observe that the client's desired CID continues to leak to the server in the provider advertisement query, since the server looks up its provider advertisement store based on this CID.

4.6.2 Private Algorithm Integration

We present our algorithms for private provider advertisement queries in Algorithms ?? and ?. They enable a client to send a PIR query while hiding the CID that it is querying for. The server returns a PIR response that allows the client to learn the providers' peer IDs, if they exist in the server's provider store.

Join provider store and address book. Similar to the non-private peer routing query, the non-private provider advertisement query uses the provider peers' IDs from the provider store as a key to the address book which contains multiaddresses. We join the two key-value stores such that for a given key (CID), the value contains the provider peer's multiaddresses (??-?). In practice, we include each provider advertisement's expiry time along with the multiaddress, though we omit these details from our algorithm. While these two key-value stores can be joined to answer *each* provider advertisement query, they only need to be joined whenever either store updates. Provider advertisements have a short default expiry period (of 24 hours), and are typically reinserted after expiration.

Binning for PIR. Given that we now have a joined key-value store, we could directly apply a KeywordPIR scheme, wherein the desired CID acts as the keyword that the server can privately lookup. However, this approach incurs high communication costs of KeywordPIR schemes. So, we transform our joined key-value store to one which enables IndexPIR schemes, which are more efficient. This follows the approach described in ?, with optimizations. We exploit the fact that the CIDs can be used to directly map each key-value pair from the joined key-value store to a hash table. CIDs in IPFS are computed using SHA-256, so we can treat them as indices of a hash table. All key-value pairs whose CIDs have the same ℓ -bit prefix are grouped into a single row or bin of the table.

We refer to this process as *binning*; this effectively corresponds to a balls-in-bins problem, with the joined provider advertisements serving as the balls. The bins would fill up non-uniformly as the number of joined provider advertisements increases. So the server pads incomplete bins to the size of the largest bin. This provides a hash table with B consistently-sized bins over which we can perform PIR. While the server could vary the number of bins with the number of provider advertisements that it stores, this would require the client to know the number of bins in advance to formulate its query. Instead, we fix B beforehand, based on the parameterization of the chosen PIR scheme,

Algorithm 13 Private provider advertisements: QUERY, PREPROCESS and RESPONSE functions.

```

1: State:  $(sk, pk)$   $\triangleright$  Peer maintains the keypair for use by all three functions.
2: State:  $(P_C, T_C)$   $\triangleright$  A preprocessed table  $T_C$  for a provider store  $P_C$ .
3: function PRIVATEPROVIDERADVERTISEMENTQUERY(CID  $c$ )
4:    $q \leftarrow c[: \log_2 B]$   $\triangleright q \in \{0, 1, \dots, B-1\}$ 
5:    $(sk, (pk, ct)) \leftarrow \text{PIR.QUERY}(q)$   $\triangleright sk$  is stored for use in PIR.EXTRACT.
6:   return  $(pk, ct)$ 
7:  $\nabla$  Preprocess  $P$  and cache the resulting table in  $T_C$  in the global state.
8: function PRIVATEPROVIDERADVERTISEMENTPREPROCESS( $P$ )
9:    $v \leftarrow 0$   $\triangleright v$  represents the size of the largest bin.
10:  for  $i \in \{1, 2, \dots, |P|\}$  do  $\triangleright$  Iterate through the provider store.
11:     $(c, \{p_1, p_2 \dots p_j\}) \leftarrow P[i]$   $\triangleright$  Get the CID and set of provider peers' IDs.
12:     $m \leftarrow \phi$ 
13:    for  $k \in \{1, 2, \dots, j\}$  do  $\triangleright$  Iterate through each provider peer ID for CID  $c$ .
14:       $m \leftarrow m \parallel M[p_k]$   $\triangleright m$  stores multiaddresses for all provider peers for  $c$ .
15:       $j \leftarrow c[: \log_2 B]$   $\triangleright$  Compute bin index using the CID prefix.
16:       $k \leftarrow \text{KDF}(c[: L - \log_2 B])$   $\triangleright$  Derive a key using the suffix of the CID.
17:       $\Psi \leftarrow \text{SE.Enc}(k, m)$   $\triangleright$  Encrypt provider peers' multiaddresses under  $k$ .
18:       $T[j] \leftarrow T[j] \cup \Psi$   $\triangleright$  Add the ciphertext multiaddresses into the  $j$ -th bin of  $T$ .
19:      if  $|T[j]| > v$  then  $\triangleright T[j]$  has more ciphertexts than the biggest bin so far.
20:         $v \leftarrow |T[j]|$ 
21:  for  $\beta \in \{1, 2, \dots, B\}$  do  $\triangleright$  Iterate through each bin.
22:     $\text{PADBIN}(T[\beta], v)$   $\triangleright$  Pad  $T[\beta]$  to have exactly  $v$  SE ciphertexts.
23:   $(P_C, T_C) \leftarrow (P, T)$   $\triangleright$  Cache  $T$  for future use.
24: function PRIVATEPROVIDERADVERTISEMENTRESPONSE( $((pk, ct), P)$ )
25:  if  $P \neq P_C$  then  $\triangleright P_C$  was preprocessed to get  $T_C$  (Line ??).
26:    PRIVATEPROVIDERADVERTISEMENTPREPROCESS( $P$ )
27:   $\text{ans} \leftarrow \text{PIR.RESPONSE}((pk, ct), T_C)$   $\triangleright$  Compute PIR response over  $T_C$ .
28:  return  $\text{ans}$ 

```

as we describe later. To retrieve a provider advertisement for a given CID, c , the client uses its $\log_2 B$ -bit prefix as their desired index in the hash table (??). It then constructs its PIR query based on this index (??). The performance of the PIR protocol is relative to the size of the rows.

Algorithm 14 Private Provider Advertisements: EXTRACT function.

```
1: function PRIVATEPROVIDERADVERTISEMENTEXTRACT(sk, ans, c)
2:    $\nabla$  Each  $\psi_i$  is the ciphertext of provider peers' multiaddresses for a single CID.
3:    $(\psi_1, \psi_2 \dots \psi_n) \leftarrow \text{PIR.EXTRACT}(\text{sk}, \text{ans})$ 
4:    $k \leftarrow \text{KDF}(c[1 : L - \log_2 B])$   $\triangleright$  Derive the key using the CID.
5:    $\text{res} \leftarrow \perp$   $\triangleright$  Stores plaintext provider advertisements, if any.
6:   for  $i \in \{1, 2, \dots, n\}$  do  $\triangleright$  Iterate through each ciphertext.
7:      $\text{res} \leftarrow \text{SE.Dec}(k, \psi_i)$   $\triangleright$  Attempt to decrypt each ciphertext.
8:     if  $\text{res} \neq \perp$  then  $\triangleright$  Successful only for one ciphertext.
9:       break
10:  if  $\text{res} = \perp$  then  $\triangleright$  No provider advertisements could be decrypted.
11:    Server peer does not store any provider advertisements for CID  $c$ .
12:   $(m_1, m_2, \dots, m_j) \leftarrow \text{res}$   $\triangleright$  Multiaddresses of provider peers for  $c$ .
13:  return  $(m_1, m_2, \dots, m_j)$ 
```

Encrypted entries. Immediately applying PIR to the binned table may enable a client to learn more than it asked for, since any row in the table contains multiple provider advertisements. To prevent this leakage, all entries can be encrypted under a key derived from the corresponding CID. Effectively, the CID acts as a *pre-shared secret* between a client who desires the provider peers for that CID and a server who stores them. The server uses a Key Derivation Function (KDF) to compute a key for each CID c in the joined store (??). In particular, the server uses the last $L - \log_2(B)$ bits of the CID; note that the first $\log_2(B)$ bits of the CID are identical for all CIDs that fall into the same bin. The server encrypts the multiaddresses for each CID under this key, using a symmetric encryption scheme (??). The server then bins the *encrypted* multiaddresses into T , using the $\log_2(B)$ -bit prefix (??). While creating the bins, the server tracks the maximum number of symmetrically encrypted ciphertexts in any bin (v in Lines ??,??–??). If any bin has fewer than this maximum number of ciphertexts, then the server pads them with random ciphertexts (Lines ??,??). For all semantically secure symmetric encryption schemes, we can generate a random ciphertext element by encrypting a constant plaintext, such as a block of all zeroes, with a randomly generated key. We note that the joining, symmetric encryption, and binning operations (Lines ??–??) are preprocessing steps that are only executed when the provider advertisement store P changes. Once a server receives a PIR query, it computes the PIR response over the table of the aforementioned symmetrically encrypted provider advertisements (??).

When a client receives a PIR response, they process it as we describe in Algorithm ??. The client first extracts the PIR response (??), and obtains a set of ciphertext provider

advertisements, which are symmetrically encrypted under keys derived from CIDs. The client can derive the key from their desired CID and use it to decrypt each ciphertext (??). Under a *robust* symmetric encryption scheme only the ciphertext encrypted with the key derived from c will decrypt to a valid plaintext (??), as we discuss below. If no ciphertexts could be decrypted under this key, then, an honest server peer does not store any provider advertisements for the given CID (??). We discuss optimal choices for PIR schemes for this use case in ??.

Security. In the context of modern symmetric encryption schemes, and specifically authenticated encryption (AE) schemes, the *key commitment* property ensures that one ciphertext can only decrypt to a valid plaintext under the key that was used for encryption. Many modern authenticated encryption (AE) schemes, such as AES-GCM [Mor24], are *not* key committing; one ciphertext can be decrypted successfully under more than just the key with which it was encrypted [ADG+22]. In our case, this corresponds to a client being able to decrypt a ciphertext, which was encrypted under a key derived from a given CID c_1 , using a key derived from another CID c_2 . The plaintext resulting from the latter decryption (say p_2) would, in general be gibberish. As such, the client will not learn multiaddresses of providers of CIDs that it does not know, and thus, server privacy is not impacted. However, p_2 may not follow the same format as the original plaintext; for instance, a valid plaintext encrypted under a key derived from c_2 would first consist of the CID c_2 followed by a multiaddress. Following a successful decryption, the client could check the format of the plaintext. Instead of relying on format checks in the application layer, we can use a robust encryption scheme to ensure that a successful decryption only occurs under the same key as that used for encryption. For example, a robust encryption scheme can be instantiated using AEZ [HKR15]. Alternately, we can reuse existing AE schemes that have been widely implemented, such as AES-GCM, after modifying them to support key commitment [ADG+22].

The security of the chosen PIR scheme ensures that server does not learn the CIDs for which the client is querying. Meanwhile, the client only receives a set of encrypted provider advertisements. The *symmetric* property of our protocol requires that the client only learns provider advertisements for the CID that they queried for, and learns nothing about other provider advertisements stored by the server. In our protocol, this reduces to the IND-CPA security of the symmetric encryption scheme and the security of the KDF computed over a CID. We defer the proof to ??; we note that we require the CIDs to be sufficiently long in comparison to the security parameter ($L \geq 2\lambda$) for this proof.

4.7 Private Content Retrieval

IPFS employs the *Bitswap* [DDP21] content retrieval protocol, which, as in the provider routing protocol, leaks the CID retrieved to the server. We describe two steps of the Bitswap protocol (WANTHAVE and BLOCK) and introduce private equivalents.

4.7.1 Bitswap Protocol

The WANTHAVE Step. The client concurrently sends a message to multiple candidate peers, asking if they possess a block with a given CID. If the peer has the requested CID in its table, it responds with a *Have* message. Otherwise, it may respond with a *DontHave* message. Candidate peers can be obtained through provider advertisement queries, as discussed in Section ???. Importantly, IPFS clients can use the WANTHAVE step to first query multiple long-lived peers in its RT, thereby assessing whether they have the content block, *before* searching for provider advertisements for that block. So, the set of candidate peers can either correspond to a set of peers in the clients' RT that are queried opportunistically, or a set of peers that are output from provider advertisement queries.

The BLOCK Step. In this step, the client asks a peer, who has responded with *Have* in the previous step, to transfer the desired block. Since it is likely that the same peer hosts multiple content blocks of the same file, as an optimization, a client can run the WANTHAVE step for multiple content identifiers with a given server peer. The server peer would respond back with the *Have* or *DontHave* message for all blocks. If the server peer holds multiple blocks, then the client runs the BLOCK step for all content identifiers held by the server. By doing so, the client avoids multiple rounds of the WANTHAVE and BLOCK steps with a server who stores multiple content blocks of the same file. We note that the total communication overhead of Bitswap is approximately the same, irrespective of whether the client queries for multiple content identifiers in the same step or not.

4.7.2 The Private Bitswap Protocol

While translating the Bitswap protocol into a private version, we address the design question of retaining the same number of rounds. We could develop a single-round, relatively heavyweight protocol, or a double-round protocol with a lightweight PRIVATEWANTHAVE step and a relatively heavyweight PRIVATEBLOCK step. We remind the reader that the client may run the Bitswap protocol opportunistically with peers

from its routing table. Thus, under the first option, the communication overhead will be that of the single-round protocol, multiplied by the number of such peers, whereas under the second option, only the overhead of the lightweight first step would matter. So, we opt for the second option. We provide additional information in the first step to reduce the overhead of the PRIVATEBLOCK step. While Private Set Intersection (PSI) schemes can be used in the first step, they tend to incur more communication overheads than PIR schemes. For the remainder of this section, recall the notation introduced in ??.

PRIVATEWANTHAVE. The goal of this step is for the client to learn the following: if the server has a block with a given CID; and if so, at which index of the server’s table does this block lie. A trivial approach would have the server send to the client a map L of each CID stored at the server to its respective index. The client can then search the map for the given CID c , and obtain its index i . This approach would reveal to the client other CIDs that are stored by the server peer. The server can resolve this leakage by computing the hash of every entry in the list using a public hash function, and generating a random salt. The server can then send a map of hashed CIDs to indices, along with the salt, to the client. Then the client uses the salt to compute the hash of the CID they desire and determine its index using the map. Under the assumption that the hash function is one-way, the client does not learn which other CIDs are stored by the server. The salt prevents the client from learning whether two provider peers store the same content block.

Alternatively, we can reuse the process for provider advertisements to transmit this information to the client via non-trivial PIR. In particular, for the PRIVATEWANTHAVE step, the client and server can engage in ??, using the map J from CIDs to the indices of their blocks, in place of the joined provider store. We first map the CID c to an index $i = J[c]$. Instead of encrypting the provider peers’ multiaddresses, we encrypt the index of the content block and thus Line ?? in ?? changes to $T[b] \leftarrow T[b] \cup \text{SE.Enc}(k, i)$.

In the PRIVATEWANTHAVE step, the server also returns the optimal PIR protocol to use in the PRIVATEBLOCK step, depending on the number of blocks they hold. This enables an adaptive approach to efficiently retrieve the block in the next step. Optimal choices of PIR schemes for the PRIVATEWANTHAVE step are the same as those for the private provider advertisement functionality (??), since we apply the same algorithm. These schemes are discussed in ??. Similarly, the security of the PRIVATEWANTHAVE is equivalent to the security of the private provider advertisement algorithm.

Algorithm 15 PRIVATEBLOCK using PIR

```
1: function PRIVATEBLOCKRESPONSE( $qu, D$ )
2:   for  $j \in \{1, 2, \dots, m\}$  do
3:      $k \leftarrow \text{KDF}(L[j])$ 
4:      $D[j] \leftarrow \text{SE.Enc}(k, D[j])$ 
5:   return PIR.RESPONSE( $q, D$ )
```

PRIVATEBLOCK. Following the PRIVATEWANTHAVE step, the client knows an index, j , of the desired block in the table D held at the server. Thus, it can issue an IndexPIR query to retrieve the desired content block. The optimal choice of PIR protocol for this step depends on the number of blocks that the peer possesses. The optimal PIR protocol is sent to the client as part of the PRIVATEWANTHAVE response, enabling the client to follow that protocol to construct a PIR query for the PRIVATEBLOCK step.

Directly applying a PIR scheme to the table D of content blocks leaves the server peer vulnerable to an index enumeration attack. A client could query for an arbitrary index and retrieve a block of content without knowing its corresponding CID. Thus, we require some cryptographic pre-processing of the data in D . We reuse a technique that we have previously identified: using the CID as a pre-shared secret between clients and servers. Each block in the table is encrypted under a key derived from the corresponding CID, so that a client receiving a PIR response can only retrieve a content block if they know its CID. We summarize how the server handles a client's PIR query q over D in ??.

In ??, we discussed that Bitswap can query for multiple content identifiers in each of the WANTHAVE and BLOCK steps. By doing so, when the server stores multiple content blocks of the same file, Bitswap eliminates multiple rounds of each step. Similarly, within our private counterpart, the client can construct and bundle up multiple PIR queries within our PRIVATEWANTHAVE and PRIVATEBLOCK steps. For each step, the server can also send PIR responses for each query, within its message back to the client. Thus, we would replace multiple pairs of round trips between these two peers with just one pair of round trips; one for the PRIVATEWANTHAVE step and one for the PRIVATEBLOCK step. Additionally, *unlike* Bitswap, we can achieve a significantly lower communication overhead when bundling up multiple PIR queries in one message to the server, as we discuss in ??.

Security. The server only obtains a PIR query from the client, and thus it can only break the security of the PRIVATEBLOCK step if it can break the security of the underly-

ing PIR scheme. On the part of the client, the security argument is equivalent to that for the private provider advertisements algorithm. We note that the same key should not be used in each application of the SE.Enc function. The implementation should disambiguate whether the key encrypts provider advertisements (for the PRIVATEPROVIDER-ADVERTISEMENTRESPONSE function), the content block index (for the PRIVATEWANTHAVE step), or the content block itself (for the PRIVATEBLOCKRESPONSE function). For example, in the HKDF Extract/Expand paradigm [Kra10], a reasonable approach would be to include a label in the *context info* field of the Expand function to distinguish these three cases.

4.8 PIR Protocols

In this section, we outline how we select PIR protocols for our algorithms from Sections ??–??. Our goal is to determine a scheme with the smallest overall communication cost, while retaining a reasonable computation cost for each of the following use cases:

1. *Few rows with small payloads*: PIR over a table of 256 rows with roughly 1.5 KB entries for private peer routing.
2. *Bins with varying size*: PIR over bins for private provider advertisements and PRIVATEWANTHAVE steps.
3. *Many rows with large payloads*: PIR over 256 KB content blocks fetched during PRIVATEBLOCK.

We wish to maintain the same number of rounds as the non-private protocols, to adhere to the same interface. Hence, the protocol chosen for each use case should only require a single round. Additionally, the PIR protocol should be efficient even if only one PIR query is issued by a client to a specific server, which is the case for our applications. In other words, we cannot amortize the communication or computation overheads over many PIR queries.

We examine existing PIR protocols in ??. While these protocols are reasonable for our PRIVATEBLOCK step, applying any of them to the prior two use cases leads to infeasible costs. To sufficiently address our use cases, we propose two PIR protocols, PAILLIERPIR and RLWEPIR, in ??. Both are tailored for the IPFS setting but may be of independent interest in other distributed systems. We evaluate the costs of our proposed and existing PIR schemes for each use case in ??.

4.8.1 Existing PIR protocols

Modern PIR protocols typically consist of offline and online phases. In the offline phase, the client and server exchange content that is independent of the PIR query, such as cryptographic keys and database-dependent hints, in order to accelerate the online phase. In the online phase, the client sends a PIR query, to which the server provides a PIR response. To adhere to having only one round of communication, we require that any offline phase that requires interaction with the client be conducted concurrently with the online phase.

A body of PIR literature aims to minimize the query processing time to sub-linear in the size of the database [ZPZS23; CK20; LMW23; CHR17; OPPW25]. However, this is achieved through an offline phase which introduces an extra round of communication to store client-specific state at either the client [ZPZS23] or the server [LMW23]. Moreover, the communication and computation costs of client-specific states are justified by the amortization assumption. Thus, these schemes do not fit both of our constraints. On the other hand, public-key doubly efficient PIR constructions can maintain this client-specific state without an additional round of communication; however, they are not yet practical as demonstrated in the literature [CHR17; OPPW25].

PIR protocols with database-dependent hints and client-specific cryptographic keys can be modified to operate in a single round. Database-dependent hints can be sent along with the PIR response, whereas client-specific cryptographic keys can be sent along with the PIR query. However, the hints add a large communication overhead that is incurred every time the database changes. Moreover, this overhead can only be amortized when a large number of queries are made to the same server, which is not applicable for our use cases.⁴ For example, for a 1 GB database, SimplePIR and DoublePIR have hints of 121 MB and 16 MB, respectively. This constraint excludes protocols such as FrodoPIR [DPC23], SimplePIR [HHC+23], and DoublePIR [HHC+23] from our evaluation.

Thus, we conclude that PIR protocols which use client-specific cryptographic

⁴Within our content retrieval use case (??), multiple content blocks of a file can be stored at the same server. The reader may reason that PIR protocols with database-dependent hints may be feasible in this case, since the cost of a large database-dependent hint may be amortized over retrievals of multiple content blocks. However, it is preferable to use PIR protocols that do not involve database-dependent hints, since the content store may change between the client's queries, necessitating sending another large hint. As we discuss below and show in our evaluation, PIR protocols such as Spiral, which do not involve database-dependent hints but do involve client-specific cryptographic keys, are preferable for this use case.

TABLE 4.4: Lower bounds of communication costs in various PIR protocols. The second column denotes the size of client-specific cryptographic keys used in each approach.

Protocol	Client KeyGen	Smallest Encrypted Query	Smallest Response Plaintext	Smallest Encrypted Response
SealPIR [ACLS18]	1.6 MB	90 KB	10 KB	181 KB
FastPIR [AYA+21]	0.67 MB	64 KB	10 KB	65 KB
OnionPIR [MCR21]	5.4 MB	64 KB	30 KB	128 KB
Spiral [MW22]	13 MB	28 KB	7.5 KB	20 KB
PAILLIERPIR	1.14 KB	0.38 KB	0.38 KB	0.76 KB
RLWEPIR	750 KB	64 KB	7.5 KB	65 KB
RLWEPIR3	192 KB	64 KB	7.5 KB	65 KB
RLWEPIR2	128 KB	64 KB	7.5 KB	65 KB

keys are the only suitable candidates, i.e., protocols such as SealPIR [ACLS18], FastPIR [AYA+21], OnionPIR [MCR21], and Spiral [MW22]. In Table ??, we see that Spiral provides small queries and responses but requires the client to send a large 13 MB cryptographic key. In contrast, SealPIR, FastPIR, and OnionPIR involve smaller cryptographic keys but require larger queries and/or responses. Importantly, we deem all existing schemes to be insufficient for our first and second use cases, as they involve much smaller payloads. As shown in ??, our PAILLIERPIR scheme supports smaller payloads, which is relevant for our peer routing use case. RLWEPIR2 and RLWEPIR3 produce the smallest client-specific cryptographic keys, in comparison to the aforementioned schemes, which is useful for the private provider advertisements and PRIVATE-WANTHAVE steps. We do consider existing PIR schemes that support 256 KB payloads, for use in the PRIVATEBLOCK step and evaluate them in ??.

4.8.2 Tailored PIR Protocols: PAILLIERPIR and RLWEPIR

Our protocols are based on Paillier or RLWE encryption and use existing techniques from cryptographic folklore. The server stores a table D that has n rows and ℓ columns. Each element in the table is encoded as one plaintext in the corresponding protocol. Suppose that the client wants to retrieve the i -th row from the database. We first outline the non-private protocol. The client generates an indicator vector that has n elements, and contains a one at index i and a zero at all other indices. The server computes an inner product between the indicator vector and the table. The product is the desired row $D[i]$, since based on our definition of the indicator vector, its product with each

column of the table results in the i th element of that column. In both PIR protocols, the client homomorphically encrypts each element of the indicator vector, and thus, the PIR query includes a ciphertext indicator vector. To compute the PIR response, the server multiplies the *ciphertext* indicator vector with the table.

PAILLIERPIR. This protocol is based on the Paillier cryptosystem [Pai99] and uses homomorphic additions (PAILLIER.Add) and scalar multiplications (PAILLIER.ScalMult). We define the encryption, decryption and homomorphic operations for the Paillier cryptosystem in ??.

We present the Query function for this protocol in ?. The client first generates a public-private keypair (?). The public key pk consists of the modulus M , which is defined as a product of two primes, along with a generator for $\mathbb{Z}_{M^2}^*$ (?). As discussed above, naively, each PIR query would include one ciphertext for each row in the table. To reduce the size of the PIR query in PAILLIERPIR, we employ a technique from Beck [Bec15]. Instead of directly sending the indicator ciphertext vector, the client seeds a cryptographically secure pseudorandom number generator (CSPRNG; ?) and the CSPRNG generates a random ciphertext vector ct (?). The client then decrypts this ciphertext vector to obtain a randomized plaintext vector (?). It uses this plaintext vector to mask each element of the indicator vector (?). The client then sends the masked plaintext vector p to the server, its seed for the CSPRNG, as well as its public key. The communication cost for one query is then the sum of the sizes of the seed, a public key and a vector of *plaintexts*. Since a Paillier plaintext is half the size of a ciphertext, this technique roughly halves the communication cost of a query.

We present the Response and Extract functions for this protocol in ?. The server provides a PIR response using the client's query (Line ?). The server initializes the CSPRNG with the same seed as the client (Line ?). This enables the server to regenerate the vector ct of n ciphertexts, just as the client did (Line ?). The server then homomorphically adds the plaintext p sent by the client to the regenerated ciphertext ct (?). As we show in our proof of correctness for PAILLIERPIR in ?, the resulting sum ct' simply encrypts the indicator vector, that is, $b_j[i] = \text{PAILLIER.Dec}(\text{sk}, ct'[i])$. The server then computes a matrix product of this ciphertext indicator vector with the table, using PAILLIER.ScalMult (Lines ?, ?) and PAILLIER.Add (Line ?). Proofs of correctness and security for PAILLIERPIR are provided in ?. In our implementation, we choose a 3072-bit modulus M for 128-bit security [Dam20; Bar20]; our ciphertexts are in $\mathbb{Z}_{M^2}^*$ and are thus 6142 bits long. The public key in PAILLIERPIR is only about 1 KB for secure parameters.

Algorithm 16 Operations in the PAILLIER cryptosystem [Pai99].

```

1: function PAILLIER.KGen( $1^\lambda$ )
2:    $\nabla$  For  $\lambda$  bit security, set the length of the Paillier modulus ( $M$  in Line ?? below).
3:    $\lambda_P \leftarrow \text{GETSECUREPAILLIERPARAMS}(\lambda)$   $\triangleright$  For  $\lambda = 128$ ,  $\lambda_P = 3072$ .
4:    $p, q \leftarrow \$\mathbb{P}$   $\triangleright$  Sample two primes of length  $\lambda_P/2$  bits.
5:    $M \leftarrow p \cdot q$ ,  $g \leftarrow M + 1$ 
6:    $\lambda \leftarrow \text{lcm}(p-1, q-1)$ ,  $\mu \leftarrow (((p-1) \cdot (q-1))^{-1}) \bmod M$ 
7:    $\text{pk} \leftarrow (M, g)$ ,  $\text{sk} \leftarrow (\lambda, \mu)$ 
8:   return ( $\text{pk}, \text{sk}$ )

9: function PAILLIER.Enc( $\text{pk}, p$ )  $\triangleright \text{pk} = (M, g)$  (Line ??)
10:   $r \leftarrow \$Z_M$ 
11:  return  $ct \leftarrow g^{p \bmod M} r^M \bmod M^2$ 

12: function PAILLIER.Dec( $\text{sk}, ct$ )  $\triangleright \text{sk} = (\lambda, \mu)$  (Line ??).
13:  return  $p \leftarrow (((ct^\lambda - 1)/M) \bmod M^2) \cdot \mu \bmod M$ 

14:  $\nabla$  Given ciphertexts  $ct_1 = g^{p_1} r_1^M \bmod M^2$  and  $ct_2 = g^{p_2} r_2^M \bmod M^2$ , encrypts  $p_1 + p_2$ .
15: function PAILLIER.CAdd( $\text{pk}, ct_1, ct_2$ )  $\triangleright \text{pk} = (M, g)$  (Line ??)
16:  return  $ct \leftarrow ct_1 \cdot ct_2 \bmod M^2$   $\triangleright ct = g^{p_1+p_2} (r_1 \cdot r_2)^M \bmod M^2$ .

17:  $\nabla$  Given ciphertexts  $ct_1 = g^{p_1} r_1^M \bmod M^2$  and  $ct_2 = g^{p_2} r_2^M \bmod M^2$ , encrypts  $p_1 - p_2$ .
18: function PAILLIER.CSub( $\text{pk}, ct_1, ct_2$ )  $\triangleright \text{pk} = (M, g)$  (Line ??)
19:   $\nabla ct'$  is the multiplicative inverse of  $ct_2$  in  $Z_{M^2}^*$ ;  $ct = g^{p_1-p_2} (r_1 \cdot r_2^{-1})^M \bmod M^2$ .
20:   $ct' \leftarrow ct_2^{-1} \bmod M^2$ 
21:  return  $ct \leftarrow \text{PAILLIER.CAdd}(\text{pk}, ct_1, ct')$ 

22:  $\nabla$  Given a ciphertext  $ct_1 = g^{p_1} r_1^M \bmod M^2$  and a plaintext  $p_2$ , encrypts  $p_1 + p_2$ .
23: function PAILLIER.Add( $\text{pk}, ct_1, p_2$ )  $\triangleright \text{pk} = (M, g)$  (Line ??)
24:   $\nabla ct$  is of the form  $g^{p_1+p_2} (r_1 \cdot r_2)^M \bmod M^2$ , and encrypts  $p_1 + p_2$  with  $r_2 = 1$ .
25:  return  $ct \leftarrow ct_1 \cdot g^{p_2} \bmod M^2$ 

26:  $\nabla$  Given a ciphertext  $ct_1 = g^{p_1} r_1^M \bmod M^2$  and a plaintext  $p_2$ , encrypts  $p_1 - p_2$ .
27: function PAILLIER.Sub( $\text{pk}, ct_1, p_2$ )  $\triangleright \text{pk} = (M, g)$  (Line ??)
28:   $\nabla ct$  is of the form  $g^{p_1-p_2} (r_1 \cdot r_2)^M \bmod M^2$ , and encrypts  $p_1 - p_2$  with  $r_2 = 1$ 
29:  return  $ct \leftarrow ct_1 \cdot g^{-p_2} \bmod M^2$ 

30:  $\nabla$  Given a ciphertext  $ct = g^x r^M \bmod M^2$  and a plaintext  $p$ , encrypts  $x \cdot p$ .
31: function PAILLIER.ScalMult( $\text{pk}, ct, p$ )  $\triangleright \text{pk} = (M, g)$  (Line ??)
32:  return  $ct' \leftarrow ct^p \bmod M^2$   $\triangleright ct' = (g^x r^M)^p \bmod M^2 = g^{x \cdot p} (r^p)^M \bmod M^2$ , which is
    an encryption of the plaintext  $x \cdot p$  with a randomization of  $r^p$ .

```

Algorithm 17 PAILLIERPIR: Query function.

```

1: function PAILLIERPIR.Query( $i$ )  $\triangleright i \in \{1, \dots, n\}$ 
2:    $(pk, sk) \leftarrow \$$  PAILLIER.KGen( $1^\lambda$ )  $\triangleright 1^\lambda$  is the security parameter.
3:    $(M, g) \leftarrow pk$   $\triangleright$  The public key includes the modulus  $M$  and a generator  $g$ .
4:   CSPRNG.Seed( $s_0$ )
5:    $\triangleright$  The CSPRNG generates  $n$  ciphertexts in  $\mathbb{Z}_{M^2}^*$ .
6:    $ct \in (\mathbb{Z}_{M^2}^*)^n \leftarrow$  CSPRNG.Generate( $n$ )
7:   for  $j \in \{1, \dots, n\}$  do
8:      $b_j = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{if } j \neq i \end{cases}$ 
9:      $r_j \leftarrow$  PAILLIER.Dec( $sk, ct[j]$ )  $\triangleright r_j \in \mathbb{Z}_M$ 
10:     $p[j] = b_j - r_j \bmod M \triangleright p \in \mathbb{Z}_M^n$ 
11:   return  $(sk, (s_0, p, pk)) \triangleright$  The client retains  $sk$ , and sends  $(s_0, p, pk)$  to the server.

```

RLWEPIR. Our second construction is based on the Ring Learning With Errors (RLWE) assumption [LPR10] and related additive homomorphic schemes [Bra12; FV12; BGV12]. RLWE plaintexts are polynomials $p(x) \in R_p = \mathbb{Z}_p/(X^N + 1)$. Plaintexts are polynomials $p(x) \in R_p = \mathbb{Z}_p/(X^N + 1)$. We use the homomorphic addition (RLWE.Add) and scalar multiplication (RLWE.ScalMult) operations offered by these schemes. Our construction combines common techniques in the literature for RLWE-based PIR protocols [ACLS18; CLS24; AYA+21], and tailors them to our setting. Compared to PAILLIERPIR, RLWEPIR has lower computation overhead but higher communication costs. We present the Query function for the RLWEPIR protocol in ??, and the Response and Extract functions in ?. If the client wants to request row i , it generates the plaintext polynomial $p(X) = X^i$, and sends an encryption of this polynomial to the server as the PIR query (Line ??–Line ??).

Upon receiving the PIR query, the server derives an encrypted indicator vector from the query using an oblivious expansion technique [ACLS18; CLS24; CCR19] (?). The server then multiplies the ciphertext indicator vector with the database (Line ??–Line ??), just as within the PAILLIERPIR.Response function, and returns the response to the client. The client decrypts the desired plaintext row from the PIR response using their secret key, just as within the PAILLIERPIR.Extract function (?, ?). We detail the oblivious expansion technique next, as it influences the communication and computation overhead of the protocol.

Oblivious expansion. Given an input ciphertext c , which encrypts $p(X) = a_0 + a_1X + \dots + a_{N-1}X^{N-1}$, oblivious expansion outputs N ciphertexts, $\{c_i\}_{i=0, \dots, N-1}$ such that c_i

Algorithm 18 PAILLIERPIR: Response and Extract functions.

```

1: function PAILLIERPIR.Response( $(s_0, p, \text{pk}), D$ )  $\triangleright D \in \mathbb{Z}_M^{n \times \ell}$ 
2:   CSPRNG.Seed( $s_0$ )
3:    $ct \in (\mathbb{Z}_{M^2}^*)^n \leftarrow \text{CSPRNG.Generate}(n)$ 
4:   for  $j \in \{1, \dots, n\}$  do
5:      $ct'[j] \leftarrow \text{PAILLIER.Add}(ct[j], p[j])$ 
6:    $\text{ans} \leftarrow \{0\}^\ell$   $\triangleright$  Initialize ans to a vector of  $\ell$  zeroes.
7:   for  $k \in \{1, \dots, \ell\}$  do
8:      $\text{ans}[k] \leftarrow \text{PAILLIER.ScalMult}(ct'[1], D[1][k])$ 
9:     for  $j \in \{2, \dots, n\}$  do
10:       $t \leftarrow \text{PAILLIER.ScalMult}(ct'[j], D[j][k])$ 
11:       $\text{ans}[k] \leftarrow \text{PAILLIER.Add}(\text{ans}[k], t)$ 
12:   return ans
13: function PAILLIERPIR.Extract(sk, ans)
14:   for  $k \in \{1, \dots, \ell\}$  do
15:      $D[k] = \text{PAILLIER.Dec}(\text{sk}, \text{ans}[k])$ 
16:   return  $D$ 

```

encrypts the scalar a_i . An automorphism key, for a given $k \in \mathbb{Z}_{2N}^*$, allows the server to compute the encryption of $p(X^k)$ from the encryption of $p(X)$; this is known as an automorphism. Angel et al. first proposed an oblivious expansion scheme that required $\log_2 N$ automorphism keys to be sent from the client to the server [ACLS18]. In particular, they generate automorphism keys for all $k = 1 + N/2^i$ for $i \in [\log_2 N]$. Our RLWEPIR variant uses Angel et al.'s approach to send all $\log_2 N$ keys from the client to the server. Since the routing table only has 256 rows, only $\log_2 256 = 8$ automorphism keys are required for RLWEPIR for peer routing. However, since provider routing involves more rows, more automorphism keys are required.

More recently, de Castro et al. [CLS24] proposed a change in oblivious expansion which requires only a constant number of automorphism keys. They observe that we can compute the automorphism for an arbitrary k' by successively applying the automorphism for a special k , that is, $p(X^{k'}) \equiv p(X) \rightarrow p(X^k) \rightarrow p(X^{k^2}) \rightarrow \dots$. This approach would only require the automorphism key for a special k . In practice, no generator can generate the set $\{1 + N/2^i\}_{i \in [\log_2 N]} \subset \mathbb{Z}_{2N}^*$. Hence, they choose two elements in $\mathbb{Z}_{2N}^*$ which can generate the entire set. For our chosen plaintext modulus degree N ($N = 4096$, as outlined below), de Castro et al. choose 3 and 1173 as the two generators. These two elements can be used to obliviously expand any ciphertext, with the least

Algorithm 19 RLWEPIR: Query function.

```
1: function RLWEPIR.Query( $i$ )  $\triangleright i \in \{1, \dots, n\}$ 
2:    $\text{sk}, \text{pk} \leftarrow \text{RLWE.KGen}(1^\lambda)$ 
3:   for  $j \in \{1, 2, \dots, \lceil n/N \rceil\}$  do
4:     if  $j * N \leq i < (j + 1) * N$  then
5:        $\text{ct}[j] \leftarrow \text{RLWE.Enc}(\text{sk}, X^{i \bmod N})$ 
6:     else
7:        $\text{ct}[j] \leftarrow \text{RLWE.Enc}(\text{sk}, 0)$ 
8:   return  $(\text{sk}, (\text{pk}, \text{ct}))$ 
```

number of automorphisms. Our RLWEPIR2 variant uses de Castro et al.’s approach for oblivious expansion, with two automorphism keys.

While de Castro et al.’s approach reduces the number of automorphism keys, it requires repeatedly applying the automorphism a large number of times, and is thus inefficient. To find a middle ground, we propose to use three keys instead of just two. We wish to obviously expand all possible ciphertexts, with fewer automorphisms. We conduct an efficient grid search of the space, for our chosen plaintext modulus degree, while minimizing the objective function stated by de Castro et al., that is, the total number of automorphisms. We arrive at three generators: namely 3, 5, and 1167. Our RLWEPIR3 variant uses three automorphism keys for oblivious expansion.

Security. In all variants of RLWEPIR, we use a polynomial modulus degree of $N = 4096$ and set the plaintext modulus to $p = 40961$, thus ensuring that $p = 1 \bmod 2N$. For our BGV scheme, we use two levels, and thus two ciphertext moduli. Our largest ciphertext modulus q is the product of 54-bit and 55-bit public primes, and thus $\log_2(q) = 109$. The aforementioned selection of parameters ($N = 4096$, $\log_2(q) = 109$) ensure correctness and provide 128-bit security for this distribution [ACC+18, Pg. 30]. The security of our protocol follows directly from that of prior work [ACLS18; CLS24]. We note that generating fewer automorphism keys in RLWEPIR2 and RLWEPIR3 does not affect the security of the protocol. We parameterize the number of bins for private provider routing and the PRIVATEWANTHAVE step, based on our value of N for RLWEPIR. Specifically, we choose to use 4096 bins, given that in RLWEPIR, the client can encode a query as large as 4096 within one ciphertext with no extra cost.

We prove the correctness and security of our PAILLIERPIR algorithm below. We then prove that our private provider advertisements algorithm serves as a symmetric PIR scheme in ??.

Algorithm 20 RLWEPIR: Response and Extract functions.

```

1: function RLWEPIR.Response((pk, ct), D)
2:    $C \leftarrow []$ 
3:   for  $j \in \{1, 2, \dots, \lceil n/N \rceil\}$  do
4:      $\triangleright$  Each  $ct[j]$  is expanded to  $N$  ciphertexts, which are appended to  $C$ .
5:      $C[(j-1)*N+1], C[(j-1)*N+2] \dots C[j*N] \leftarrow \text{ObliviousExpand}(\text{pk}, ct[j])$ 
6:      $\text{ans} \leftarrow \{0\}^\ell$   $\triangleright$  Initialize ans to a vector of  $\ell$  zeroes.
7:     for  $k \in \{1, \dots, \ell\}$  do
8:        $\text{ans}[k] \leftarrow \text{RLWE.ScalMult}(C[1], D[1][k])$ 
9:       for  $j \in \{2, \dots, n\}$  do
10:         $t \leftarrow \text{RLWE.ScalMult}(C[j], D[j][k])$ 
11:         $\text{ans}[k] \leftarrow \text{RLWE.Add}(\text{ans}[k], t)$ 
12:   return ans
13: function RLWEPIR.Extract(sk, ans)
14:   for  $k \in \{1, \dots, \ell\}$  do
15:      $D[k] = \text{RLWE.Dec}(\text{sk}, \text{ans}[k])$ 
16:   return D

```

4.8.3 Correctness and Security of PAILLIERPIR

We begin with preliminary definitions. We remind the reader of ??, which outlines the PIR scheme functions Query, Response and Extract. We now define the correctness and security guarantees for PIR schemes.

Definition 4.8.1 (Correctness of a PIR scheme.). We define a PIR scheme as being *correct* if, for any database $D = \{d_i\} \in \mathcal{D}^n$ and any queried index $i \in [n]$, the row returned by the protocol, d^* , is identical to the row in the database, d_i , with a high probability given by $1 - \delta$, for some negligible δ .

$$\Pr \left[d_i = d^* \mid \begin{array}{l} (\text{st}, \text{qu}) \leftarrow \text{Query}(i), \\ \text{ans} \leftarrow \text{Response}(\text{qu}, D), \\ d^* \leftarrow \text{Extract}(\text{st}, \text{ans}) \end{array} \right] \geq 1 - \delta,$$

Definition 4.8.2 (PIR security). For a security parameter λ , we define a PIR scheme as being *secure*, if, given a PIR query qu , the probability by which any polynomial-time adversary $\mathcal{A}$ can correctly determine whether the query was constructed from index i_0

$G_{\text{PIR}}^{\text{Security}}(\mathcal{A})$	$G_{\text{PKE}}^{\text{IND-CPA}}(\mathcal{B})$
1 : $i_0, i_1 \in [n] \leftarrow \$ \mathcal{A}(1^\lambda)$	1 : $(\text{pk}, \text{sk}) \leftarrow \$ \text{KGen}(1^\lambda)$
2 : $b \leftarrow \$ \{0, 1\}$	2 : $m_0, m_1 \leftarrow \$ \mathcal{B}(1^\lambda, \text{pk})$
3 : $\text{st}, \text{qu} \leftarrow \$ \text{Query}(i_b)$	3 : $b \leftarrow \$ \{0, 1\}$
4 : $b' \leftarrow \mathcal{A}(1^\lambda, \text{qu})$	4 : $c \leftarrow \text{Enc}(\text{pk}, m_b)$
5 : return $b = b'$	5 : $b' \leftarrow \mathcal{B}(1^\lambda, \text{pk}, c)$
	6 : return $b = b'$

FIGURE 4.1: The indistinguishability game for security of a PIR scheme (??) is shown on the left, whereas that for semantic security of a PKE scheme (??) is shown on the right.

or i_1 , where $i_0, i_1 \in [n]$, is bounded by a negligible function of λ .

$$\forall i_0, i_1 \in [n] \left| \Pr[\mathcal{A}(1^\lambda, \text{qu}) = 1 : (\text{st}, \text{qu}) \leftarrow \text{Query}(i_0)] - \Pr[\mathcal{A}(1^\lambda, \text{qu}) = 1 : (\text{st}, \text{qu}) \leftarrow \text{Query}(i_1)] \right| \leq \text{negl}(\lambda)$$

This property reflects the privacy of the client in the PIR scheme. We present an indistinguishability game $G_{\text{PIR}}^{\text{Security}}(\mathcal{A})$ in ?? to reflect this notion. The adversary samples two indices (??) and the challenger generates a PIR query for one of these indices (Lines ??–??). We note that the Query function internally generates a keypair (e.g., ?? in ??), and this key generation step depends on the security parameter λ . The adversary then receives the challenger’s query and guesses the index chosen by the challenger (Line ??). Implicitly, the adversary has access to the Response function and can choose arbitrary databases D , and can compute the response ans to the query qu that is supplied by the challenger.

Based on this indistinguishability game, a PIR scheme consisting of (Query, Response, Extract) functions is secure, if for any polynomial-time adversary $\mathcal{A}$ that is parameterized by λ , we have that

$$\text{Adv}_{\text{PIR}}^{\text{security}}(\mathcal{A}) = \left| \Pr[G_{\text{PIR}}^{\text{Security}}(\mathcal{A}) = 1] - 1/2 \right| \leq \text{negl}(\lambda)$$

We introduce the notion of semantic security of public-key encryption schemes.

Definition 4.8.3 (Semantic security of PKE schemes). For a security parameter λ , a public-key encryption scheme $\text{PKE} = (\text{KGen}, \text{Enc}, \text{Dec})$ is IND-CPA secure, if for any

polynomial-time adversary $\mathcal{B}$, we have that

$$\text{Adv}_{\text{PKE}}^{\text{IND-CPA}}(\mathcal{B}) = |\Pr[\text{G}_{\text{PKE}}^{\text{IND-CPA}}(\mathcal{B}) = 1] - 1/2| \leq \text{negl}(\lambda),$$

where $\text{G}_{\text{PKE}}^{\text{IND-CPA}}$ is defined above.

Proof of correctness for PAILLIERPIR.

Theorem 4.8.1. Let M denote the plaintext modulus of Paillier. For any database D of size $n \times \ell$, i.e., $D \in \mathbb{Z}_M^{n \times \ell}$, and any row index $i \in [n]$, if

$$(\text{sk}, (s, p, \text{pk})) \leftarrow \text{PAILLIERPIR.Query}(i) \quad (??, ??) \quad (4.1)$$

$$\text{ans} \leftarrow \text{PAILLIERPIR.Response}((s, p, \text{pk}), D) \quad (??, ??) \quad (4.2)$$

$$d^* \leftarrow \text{PAILLIERPIR.Extract}(\text{sk}, \text{ans}) \quad (??, ??) \quad (4.3)$$

then $d^* = D[i]$.

Proof. Since d^* and $D[i]$ are rows ($d^*, D[i] \in \mathbb{Z}_M^\ell$), we will prove that they are identical for each element, i.e., $d^*[k] = D[i][k]$ for every $k \in [\ell]$. We observe that the intermediate ciphertext ct' that is computed within ?? of our PAILLIER.Response function, simply encrypts the indicator vector b . In other words, using the additive properties of Paillier, we see that for every column element $j \in [n]$,

$$\text{PAILLIER.Dec}(\text{sk}, ct'[j]) \quad (4.4)$$

$$= \text{PAILLIER.Dec}(\text{sk}, \text{PAILLIER.Add}(\text{pk}, ct[j], p[j])) \quad (4.5)$$

$$= \text{PAILLIER.Dec}(\text{sk}, ct[j]) + p[j] \quad (4.6)$$

$$= r_j + (b_j - r_j) = b_j \quad (4.7)$$

where operations are modulo M in the above equations.

Then, considering the PAILLIER.Extract function, we can then reason that for every row element $k \in [\ell]$,

$$d^*[k] = \text{PAILLIER.Dec}(\text{sk}, \text{ans}[k]) \quad (4.8)$$

$$= \text{PAILLIER.Dec}(\text{sk}, \text{PAILLIER.CAdd}(\text{pk}, ct_1, ct_2, \dots, ct_n)) \quad (4.9)$$

$$\text{where } ct_j = \text{PAILLIER.ScalMult}(\text{pk}, ct'[j], D[j][k])$$

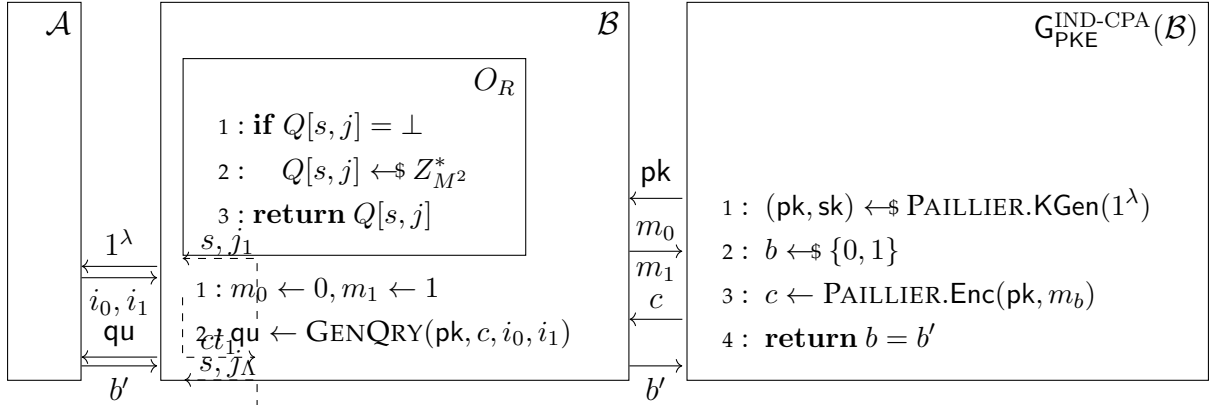


FIGURE 4.2: Black-box reduction from an adversary $\mathcal{A}$ against the PIR security of PAILLIERPIR (?? and $G_{\text{PIR}}^{\text{Security}}$ in ??) to an adversary $\mathcal{B}$ against IND-CPA of the Paillier PKE cryptosystem (?? and $G_{\text{PKE}}^{\text{IND-CPA}}$ in ??). The GENQRY function, which is used by $\mathcal{B}$ to generate a PAILLIERPIR query for $\mathcal{A}$, is described in ?. O_R denotes the random oracle implementing a CSPRNG. We depict queries made by the PIR adversary $\mathcal{A}$ to the random oracle through dashed lines; $\mathcal{A}$ can query this oracle before, during, or after the execution of GENQRY up to a polynomial number of times in the security parameter λ , that is, up to $\Lambda = \text{poly}(\lambda)$ times.

Then, through the scalar multiplicative property of Paillier ($\text{PAILLIER.Dec}(sk, \text{PAILLIER.ScalMult}(ct, \alpha)) = \alpha \cdot \text{PAILLIER.Dec}(sk, ct)$), we can conclude that:

$$d^*[k] = \sum_{j=1}^n D[j][k] \cdot \text{PAILLIER.Dec}(sk, ct'[j]) \quad (4.10)$$

$$= \sum_{j=1}^n D[j][k] \cdot b_j = D[i][k] \quad (4.11)$$

where the last equation is due to the definition of the indicator vector ($b_j = \mathbb{I}[j = i]$). Combining the result for all values of $k \in [\ell]$, we observe that $d^* = D[i]$. $\square$

Proof of security for PAILLIERPIR.

Theorem 4.8.2. Let Query be as defined in PAILLIERPIR.Query of ?. Modelling the CSPRNG used within that algorithm as a random oracle, PAILLIERPIR is secure (??) if PAILLIER is IND-CPA secure (??).

Algorithm 21 GENQRY function used by $\mathcal{B}$ to generate a PIR query for $\mathcal{A}$ in the black box reduction in ??.

```

1: function GENQRY(pk, c,  $i_0, i_1$ )
2:    $s_0 \leftarrow \$ \{0, 1\}^\lambda$   $\triangleright$  Sampling a seed for the random oracle.
3:   for  $i \in [n]$  do
4:     if  $i = i_0$  then
5:        $v \leftarrow \text{PAILLIER.CSub}(\text{PAILLIER.Enc}(\text{pk}, 1), c)$ 
6:     else if  $i = i_1$  then
7:        $v \leftarrow \text{PAILLIER.CAdd}(\text{PAILLIER.Enc}(\text{pk}, 0), c)$ 
8:     else
9:        $v \leftarrow \text{PAILLIER.Enc}(\text{pk}, 0)$ 
10:     $x \leftarrow \$ Z_M$   $\triangleright$  Sample a random plaintext.
11:     $ct \leftarrow \text{PAILLIER.Sub}(v, x)$   $\triangleright$  Mask  $v$  by the random plaintext to get  $ct$ .
12:     $p[i] \leftarrow x$ 
13:    if  $Q[s_0, i] \neq \perp$  then
14:      Abort  $\triangleright \mathcal{A}$  queried for  $Q[s_0, i]$  before  $\mathcal{B}$  chose  $s_0$ .
15:    else
16:       $Q[s_0, i] \leftarrow ct$ 
17:  return ( $s_0, p, \text{pk}$ )

```

Proof. We reduce the security of the PIR scheme to the IND-CPA security of the Paillier cryptosystem. In particular, for any adversary $\mathcal{A}$ attacking the security of PAILLIERPIR, we construct a new adversary $\mathcal{B}$, that provides an implementation of the random oracle to $\mathcal{A}$ and attacks the IND-CPA security of Paillier. We present a black-box reduction from $\mathcal{A}$ to $\mathcal{B}$ in ??. We depict the IND-CPA game at the right of ?? and its adversary $\mathcal{B}$ in the middle of the figure, while the adversary $\mathcal{A}$ for the PIR security game is shown at the left of the figure. We use a random oracle $\mathcal{O}_R$ to model the CSPRNG used in the PAILLIERPIR.Query function (Lines ??–?? in ??). In particular, $\mathcal{A}$ has access to $\mathcal{O}_R$, which is implemented by $\mathcal{B}$. When given a seed s of λ bits and an index, the random oracle checks if its internal table Q is set for that seed and index, and if not, it sets this value to a random Paillier ciphertext. When queried again with the same inputs, $\mathcal{O}_R$ always returns the same ciphertext. We emphasize that $\mathcal{A}$ can query the random oracle *before, during or after* interacting with $\mathcal{B}$.

$\mathcal{B}$ first requests $\mathcal{A}$ to return two indices $i_0, i_1 \in [n]$ for the PIR security game. $\mathcal{B}$ then engages in the IND-CPA game, and obtains the public key pk from the IND-CPA challenger. $\mathcal{B}$ creates two messages such that m_0 is zero and m_1 is one, and supplies them to the IND-CPA challenger, which returns with a Paillier encryption c of m_b for

a random bit b . $\mathcal{B}$ generates a PIR query qu through the GENQRY function, using the public key pk , the ciphertext c , as well as the indices i_0 and i_1 . (We note that $\mathcal{B}$ programs the random oracle within this function.) Finally, $\mathcal{B}$ provides this PIR query to $\mathcal{A}$, which returns back with a guess b' for the bit b . $\mathcal{B}$ simply forwards the bit guessed by $\mathcal{A}$ to the IND-CPA challenger.

We now describe the GENQRY function used by $\mathcal{B}$ to generate a PIR query for $\mathcal{A}$. We first sample a seed of λ bits for the random oracle (??). We then iterate over all row indices of the database (??). We construct a ciphertext v , whose value depends on the row index and the ciphertext c output by the IND-CPA challenger. If the row index is i_0 , then v is given by the difference between a Paillier encryption of 1 and the ciphertext c (??). If the row index is i_1 , then v is given by the sum of c and a Paillier encryption of 0 (??). For all other indices, v is given by the Paillier encryption of 0 (??). We then sample a plaintext x uniformly at random (??) and then add it to v , in order to get a ciphertext ct (??). We then set the i -th index of the plaintext vector to the random plaintext x (??). We query whether the random oracle is set for the seed s_0 and the given index i (??). If it is, then $\mathcal{A}$ queried for $Q[s_0, i]$ before $\mathcal{B}$ chose s_0 , and we abort the game (??). If the random oracle is *not* set for this seed and index, then we set it to the ciphertext ct . We repeat this process for all row indices.

$\mathcal{B}$ perfectly simulates the Security game to $\mathcal{A}$, as follows. $\mathcal{A}$ receives an input of 1^λ , which is trivially identical in both $G_{\text{PIR}}^{\text{Security}}(\mathcal{A})$ from ?? and in ?. $\mathcal{A}$ also obtains a query, which consists of the seed s_0 , the plaintext p and the public key pk . First, we note that $\mathcal{B}$ chooses the seed s_0 uniformly at random (Line ??, ??), and thus, $\mathcal{A}$ would observe identical distribution of the seed, whether it is in the game $G_{\text{PIR}}^{\text{Security}}(\mathcal{A})$ or with $\mathcal{B}$ as the challenger. Second, the Paillier keypair is generated in the IND-CPA game (?? in $G_{\text{PKE}}^{\text{IND-CPA}}(\mathcal{A})$ in ??) exactly as within the PAILLIERPIR.Query function in the game $G_{\text{PIR}}^{\text{Security}}(\mathcal{A})$ (??, ??). Thus, $\mathcal{A}$ would observe an identical distribution of the public key, irrespective of whether it is interacting with $G_{\text{PIR}}^{\text{Security}}(\mathcal{A})$ or with $\mathcal{B}$, who simply forwards the public key generated by $G_{\text{PKE}}^{\text{IND-CPA}}(\mathcal{A})$. Third, each plaintext entry in p is selected uniformly at random from Z_M , and is thus indistinguishable to $\mathcal{A}$ from that in Line ?? of ??, which is a plaintext selected uniformly at random, and then offset by a constant (zero or one).

We now show that ct is indistinguishable to $\mathcal{A}$ from a ciphertext chosen at random from $Z_{M^2}^*$. If the IND-CPA game chose bit b , then since c is set to the Paillier encryption of $m_b = b$, v will be set to the encryption of 1 for the chosen index i_b and to the encryption of bit 0 for all other indices, based on Lines ??-?? of ?. $v = \text{PAILLIER.Enc}(pk, b)$ will be of the form $g^{br^M} \bmod M^2$, and based on the definition of PAILLIER.Sub (Line ??,

Algorithm ??), ct will be of the form $g^{(M-x+b) \bmod M} r^M \bmod M^2$, and a ciphertext chosen at random is of the form $g^y r^M \bmod M^2$, and the two follow identical distributions. In other words, $\mathcal{A}$ observes identical distributions of the ciphertexts from the random oracle, irrespective of whether the random oracle samples the ciphertexts at random (Line ?? of $\mathcal{O}_R$ in ??) or whether $\mathcal{B}$ programs the oracle (Line ?? in ??). In summary, we can see that $\mathcal{B}$ creates a valid query, which includes the seed s_0 , a vector of plaintexts p and a vector of ciphertexts returned by the random oracle. We have shown that each of the three components of this query are indistinguishable from that generated by the `PAILLIERPIR.Query` function (?). While both ct and p are drawn from uniform distributions, when the PIR adversary $\mathcal{A}$ adds them together within the `PAILLIERPIR.Response` function (Line ??, Algorithm ??), the resulting ciphertext vector should be indistinguishable from an encryption of an indicator vector. For the chosen index i_b , ct will be equal to `PAILLIER.Enc(pk, 1 - x)`, and for all other indices to `PAILLIER.Enc(pk, -x)`. Thus, the sum of the vector of ciphertexts and the plaintexts, will simply be an indicator vector for the chosen index. Importantly, $\mathcal{B}$ creates a valid query for index i_b when the IND-CPA game chooses bit b . So, when $\mathcal{A}$ makes a guess b' for the challenge bit b , then $\mathcal{B}$ can simply pass this bit back to the IND-CPA challenger.

The probability that $\mathcal{B}$ wins is therefore at least the probability that $\mathcal{A}$ wins minus the probability that $\mathcal{B}$ aborts. As discussed above, $\mathcal{B}$ aborts when $\mathcal{A}$ queried the random oracle for $Q[s_0, i]$ before $\mathcal{B}$ chose s_0 (?). This only occurs when $\mathcal{A}$ guesses s_0 correctly, which is true with a probability $1/2^\lambda$. For a security parameter of λ , $\mathcal{A}$ can only query the random oracle at most $\text{poly}(\lambda)$ times, and thus $\mathcal{B}$ aborts with a probability at most $\text{poly}(\lambda)/2^\lambda$, which is negligible. Thus, the advantages of the two adversaries are related as follows:

$$\text{Adv}_{\text{PAILLIERPIR}}^{\text{Security}}(\mathcal{A}) \leq \text{Adv}_{\text{PAILLIER}}^{\text{IND-CPA}}(\mathcal{B}) + \text{poly}(\lambda)/2^\lambda$$

If `PAILLIER` is IND-CPA secure, then $\text{Adv}_{\text{PAILLIERPIR}}^{\text{Security}}(\mathcal{A}) \leq \text{negl}(\lambda)$, and thus, `PAILLIERPIR` is secure. $\square$

4.8.4 Security of Private Provider Advertisements

We formalize the security requirements and proof for the private provider advertisements protocol (?). The results here also apply to the protocol used for `PRIVATEWANTHAVE`, since the same algorithm is used. We require that this protocol satisfies a *sym-metric* property, such that the client only learns the value (provider peer IDs and multiaddresses) for a CID that they know, and does not learn the value for any CIDs that they do not know. We outline a security game in ?? that demonstrates that an adversary

Algorithm 22 Indistinguishability game G_1 for symmetric PIR for Private Provider Advertisements.

```

1: function  $G_1(\mathcal{A}, 1^\lambda)$ 
2:    $T \leftarrow \mathcal{A}(1^\lambda)$   $\triangleright \mathcal{A}$  runs PRIVATEPROVIDERADVERTISEMENTPREPROCESS.
3:    $\nabla$  Each list  $L_b$  consists of  $s$  provider peers and their multiaddresses;  $\mathcal{A}$  is also free
   to choose  $s = \text{poly}(\lambda)$  arbitrarily.
4:    $L_0, L_1 \leftarrow \mathcal{A}$ 
5:    $b \leftarrow \$ \{0, 1\}$   $\triangleright$  Challenger samples a bit.
6:    $\{m_1, m_2, \dots, m_s\} \leftarrow L_b$   $\triangleright$  Challenger picks one of the two lists.
7:    $v \leftarrow 0$   $\triangleright v$  represents the size of the largest bin in  $T$ .
8:   for  $i \in \{1, 2, \dots, s\}$  do  $\triangleright$  Challenger processes each  $m_i$ .
9:      $c_i \leftarrow \$ \{0, 1\}^L$   $\triangleright$  Sample a random CID of length  $L = 256$  bits.
10:     $\nabla$  Note that Lines ??–?? are identical to Lines ??–?? of the PRIVATEPROVIDER-
    ADVERTISEMENTPREPROCESS function in Algorithm ??.
11:     $j \leftarrow c_i[: \log_2 B]$   $\triangleright$  Compute bin index using the CID prefix.
12:     $k_i \leftarrow \text{KDF}(c_i[: L - \log_2 B])$   $\triangleright$  Derive a key using the suffix of the CID.
13:     $\Psi \leftarrow \text{SE.Enc}(k_i, m_i)$   $\triangleright$  Encrypt provider peers' multiaddresses under  $k_i$ .
14:     $T[j] \leftarrow T[j] \cup \Psi$   $\triangleright$  Add the ciphertext multiaddresses into the  $j$ -th bin of  $T$ .
15:    if  $|T[j]| > v$  then  $\triangleright T[j]$  has more ciphertexts than the biggest bin so far.
16:       $v \leftarrow |T[j]|$ 
17:   for  $\beta \in \{1, 2, \dots, B\}$  do  $\triangleright$  Iterate through each bin.
18:      $\text{PADBIN}(T[\beta], v)$   $\triangleright$  Pad  $T[\beta]$  to have exactly  $v$  SE ciphertexts.
19:    $b' \leftarrow \mathcal{A}(T)$ 
20:   return  $b = b'$ 

```

$\mathcal{A}$ that does not know a CID cannot retrieve the values for that CID. Importantly, even if the adversary were to *insert* values for a set of CIDs that it *does* know, and crafts malicious values for the set of CIDs that it does not know and supplies these values to the challenger, we show that knowledge of the CIDs for the first set does not help the adversary in learning the values for the second set. We also highlight that our game does not depend on the PIR scheme at all; in fact, we use a trivial PIR scheme to demonstrate that any leakage of the payload through the PIR scheme is irrelevant and eliminated through the symmetric encryption scheme.

We present our security game G_1 in ??. The adversary supplies the challenger with a preliminary table T (Line ??). It may do so by creating a provider store P and preprocessing it through the PRIVATEPROVIDERADVERTISEMENTPREPROCESS function (Lines ??–?? of ??) or by any other means. The adversary thus has full control over

the CIDs, their provider peers' IDs and these peers' multiaddresses, and in particular, can choose how the (joined) provider advertisements are symmetrically encrypted and binned in T . The adversary creates two lists: L_0 and L_1 that consist of up to s provider peer IDs and their multiaddresses, namely $\{m_1, \dots, m_s\}$, while excluding the CIDs. The adversary may create these lists by going through Lines ??–?? of Algorithm ?? or by any other means; the respective CIDs for each provider peer ID are irrelevant. The adversary then supplies these lists to the challenger (Line ??). The challenger picks a bit b at random (Line ??) and the corresponding list (Line ??). We denote the maximum size of any bin in T , including any padding elements, by v . The challenger generates $s = |L_b|$ CIDs at random from the CID space (Line ??). The challenger then determines the bin index that each CID c_i should be placed in (Line ??), derives a key from a suffix of the CID, of length $L - \log_2(B)$ bits, through a KDF (Line ??), and symmetrically encrypts each joined provider advertisement m_i , which was supplied by the adversary, using this key (Line ??). The challenger then places the ciphertext into the appropriate bin of the table T (Line ??), which was also supplied by the adversary. The challenger also pads each bin, ensuring that each bin has a total of v elements (Lines ??–??). The challenger then simply returns the resulting table to the adversary; in this step, we effectively model a trivial PIR scheme, which supplies at least as much information to $\mathcal{A}$ as any other PIR scheme. The adversary must then guess the bit b ; that is, the adversary must determine whether the resulting table was constructed using L_0 or L_1 (Lines ??–??).

Theorem 4.8.3. Let the `PRIVATEPROVIDERADVERTISEMENTPREPROCESS` function (PPAP) be defined as in ?. For an adversary $\mathcal{A}$ against this function, as defined in the game G_1 in ?, we give algorithms $\mathcal{B}_1, \mathcal{B}_2$ such that:

$$\text{Adv}_{\text{PPAP}, G_1}^{\text{spirsecurity}}(\mathcal{A}) \leq \text{Adv}_{\text{KDF}}^{\text{PRF}}(\mathcal{B}_1) + \text{poly}(\lambda) \cdot \text{Adv}_{\text{SE}}^{\text{IND-CPA}}(\mathcal{B}_2)$$

where KDF refers to the key derivation function (Line ??), and SE refers to the symmetric encryption scheme (Line ??), used in PPAP.

Proof. We proceed with a sequence of games. In G_2 , the challenger uses a truly random function in place of the KDF (Line ??, ??) to encrypt the CIDs that it randomly generates. This replaces the key k_i that was derived from each CID c_i ($k_i = \text{KDF}(c_i[1 : L - \log_2(B)])$), with a random value k_i^* of the same length. We require the CIDs to be sufficiently long in comparison to the security parameter ($L - \log_2(B) \geq \lambda$); this condition is satisfied when $L \geq 2\lambda$. Consider an adversary $\mathcal{B}_1$ who follows a game to attack the security of the pseudorandom function used in the KDF [Kra10, Definition 7], that is, $\mathcal{B}_1$ attempts to distinguish the output of a KDF from a truly random string. We denote the advantage

of this adversary by $\text{Adv}_{\text{KDF}}^{\text{PRF}}(\mathcal{B}_1)$. We can then bound the difference in the advantage of $\mathcal{A}$ in G_1 and in G_2 , by the aforementioned advantage term, as follows:

$$|\text{Adv}_{\text{PPAP}, G_1}^{\text{spirsecurity}}(\mathcal{A}) - \text{Adv}_{\text{PPAP}, G_2}^{\text{spirsecurity}}(\mathcal{A})| \leq \text{Adv}_{\text{KDF}}^{\text{PRF}}(\mathcal{B}_1)$$

Next, in G_3 , the challenger replaces the input of SE.Enc (Line ??, ??), which was the value m_i provided by the adversary, with a random value m_i^* of the same length. We need $s = \text{poly}(\lambda)$ game hop steps in order to change all m_i to m_i^* . Expecting the adversary to distinguish between the ciphertexts generated under the randomly sampled key k_i^* for the messages m_i and m_i^* is equivalent to the IND-CPA game for the symmetric encryption scheme. Consider an adversary $\mathcal{B}_2$ in the IND-CPA security game for the symmetric encryption scheme. Then, we can bound this hop by the advantage of $\mathcal{B}_2$ against the IND-CPA security of SE.Enc , which is denoted by $\text{Adv}_{\text{SE}}^{\text{IND-CPA}}(\mathcal{B}_2)$:

$$|\text{Adv}_{\text{PPAP}, G_2}^{\text{spirsecurity}}(\mathcal{A}) - \text{Adv}_{\text{PPAP}, G_3}^{\text{spirsecurity}}(\mathcal{A})| \leq \text{poly}(\lambda) \cdot \text{Adv}_{\text{SE}}^{\text{IND-CPA}}(\mathcal{B}_2)$$

Now, the response T' returned to $\mathcal{A}$ does not depend on L_b at all. Hence, $\mathcal{A}$ has no greater advantage than by guessing the challenge bit b and $\text{Adv}_{\text{PPAP}, G_3}^{\text{spirsecurity}}(\mathcal{A}) = 0$. This concludes the proof. We note that the key commitment property, which was discussed in ??, is not required for our proof to hold; that property is simply required for the client to be able to correctly determine whether a decryption of a ciphertext is valid. $\square$

4.9 Implementation and Privacy Engineering

We now describe our implementation from the lens of integrating a PET, namely Peer2PIR, which realizes the MINIMIZE and HIDE privacy design strategies, into a back-end web architecture, namely the IPFS distributed file system. We first outline the IPFS architecture and then describe our implementation. IPFS follows a modular architecture and consists of libraries for different functionalities. Each library is often implemented in multiple languages; we focus on implementations in Golang. IPFS includes a core library, namely LibP2P, that implements the Kademlia DHT and routing table. This library provides an interface over the RT; peers can add other peers to the RT, as well as retrieve the peers closest to a given peer ID from the RT. IPFS provides its peer routing and provider routing algorithms in its Zikade library. This library enables peers to communicate with each other using Protobufs [Ste20]. It maintains data structures for both functionalities, including an address book that maps peer IDs to multiaddresses (??) and a provider store that maps CIDs to their providers (??). In particular, when a

client wishes to find peer IDs of providers for a CID, the Zikade library constructs a provider routing query with that CID as input, along with a peer routing query with the hash of the CID. Similarly, once a client has found a provider peer ID, and wishes to find the provider’s multiaddress, this library constructs a peer routing query with that provider peer ID as the input. This library also implements server-side logic to handle each query. It uses the LibP2P library to interface with the RT for peer routing.

First, we implement our normalization algorithm (??) as an extension of the LibP2P Golang library. Second, we implement PAILLIERPIR (Algorithms ??, ??) and RLWEPIR (Algorithms ??, ??) for peer and provider routing within the Zikade library. Third, we develop a benchmark to evaluate existing PIR protocols for content retrieval. In our extension of the LibP2P and Zikade libraries, we used existing tests and tooling to determine where and how to integrate our PET within an existing codebase, as we describe below. In this process, we also identified two challenges, namely, anticipating different types of preprocessing required for PETs, and dealing with randomization introduced by PETs. Finally, we seek two properties for our implementation, namely, backwards compatibility with non-private versions and interoperability across PETs implementations. We contextualize these two properties within the IPFS setting, and describe how we achieve them below. Our discussion may also apply to other problems where the goal is to integrate PETs, such as differential privacy or multi-party computation, into an existing non-private codebase.

Start with tests and observability tools. The LibP2P and Zikade Golang libraries included unit and end-to-end tests. We executed these tests and simultaneously captured data flows by using observability tools such as distributed tracing. Distributed tracing (DT) enables the flow of data through different functions and consequently, identify misbehaving functions [PSMI20]. Popular distributed tracing implementations include OpenTelemetry [Ope22c], Jaeger [Jae23a] and Zipkin [Ope23d]. Similar to flame graphs used to profile performance [Gre20], distributed tracing outputs traces for each top-level function invocation. Each trace forms a tree of spans [Ope23c], with a span describing a function call.

Traces generated by tests for a given top-level function will quickly reveal the set of functions that are used to implement that feature. In particular, we will be able to track how private information is used by these functions. Typically, each function that uses the private information in such a trace will need to be re-implemented privately.

Non-private and private implementations of the same interface may need to coexist in a codebase, potentially for backwards compatibility reasons as we discuss below.

Thus, after implementing these functions privately, distributed tracing can help ensure that the correct (private or non-private) implementation is being called. For instance, when a client peer begins a peer routing process to find a target peer ID, it first consults its own routing table to find other peers to contact. This first lookup should continue with the full target peer ID. However, whenever a server peer receives a query from a client peer, we arrange for our private implementation to be called, and we confirm that this is the case through distributed tracing.

Moreover, if any of the newly implemented functions are also present in traces of tests for other features, then these features could also be impacted by integrating the private implementation. So, performing end-to-end tests and analyzing their traces allows us to identify cross-feature interactions that would be relevant when re-implementing the above set of functions. As an example of an *expected* cross-feature interaction, we find that our private peer routing algorithm is also invoked during provider routing, with the hash of the CID whose provider we want to find as input; this is expected, as discussed in ??.

Preprocessing data structures. PETs researchers often focus on cryptographic preprocessing operations that the PET itself requires. For instance, when a new row is added to a table, PIR schemes require re-encoding that row before PIR queries can be answered. In practice, other non-cryptographic operations may *also* be required when integrating PETs into real-world codebases, since the data structures in the non-private codebase often differ from the model used in PETs. Apart from analyzing traces, we drew a data flow diagram for the non-private codebase. Consequently, we determined that when answering a peer routing query non-privately, the server first looks up a routing table with peer IDs and then looks up an address store for multiaddresses, as discussed in ?. We note that PIR papers assume that queries are conducted over one table. Our data flow diagram revealed that we needed to gather data from multiple tables into a single table for PIR. Specifically, we join the routing table with the address book for private peer routing (?), and we join the provider store with the address book for private provider routing (?). We thus highlight that both cryptographic and non-cryptographic preprocessing operations are required when integrating PETs. Developing a data flow diagram of the non-private codebase may help privacy engineers in identifying non-cryptographic preprocessing operations.

Dealing with randomization when sending queries or receiving responses. Current IPFS implementations expect the client to send the same message to each server in its

path to the target, for peer and provider routing. However, the private versions of these functionalities require that the client send a different message to each server peer. In particular, these messages are homomorphically encrypted ciphertexts, corresponding to the PIR query. In both private peer and provider routing, different ciphertexts are sent to each server peer. In private peer routing, even the plaintext for each peer is different, since it is the target peer’s CPL with each server peer. In private provider routing, the prefix of the CID serves as the plaintext, and thus the plaintext is identical for different server peers. Nevertheless, since homomorphic encryption schemes are randomized and not deterministic, different ciphertexts are sent to each peer within private provider routing.

Engineering this change in IPFS required appropriately abstracting all client-side logic for sending and receiving messages, and executing it through callbacks. The callbacks would derive a ciphertext from the non-private message (target peer ID or target CID) for each server peer; they would be executed just before, and instead of, transmitting the non-private message.

Many PETs inherently involve randomization; for instance, through encryption or differential privacy. For example, in the case of differential privacy, query responses may be randomized every time. Consequently, in the private implementation of a given top-level feature, the sender might send different, randomized messages to different recipients. As such, this discrepancy may arise in both centralized (client-server) or distributed (peer-to-peer) architectures. It may be challenging to send different queries (or responses) to different recipients, when the non-private implementation does not expect such behaviour, as exemplified above in the case of IPFS. In general, addressing such problems may require significant redesign to avoid code smell [LPPG20, Table 2].

Backwards compatibility with non-private versions. In general, backwards compatibility enables gradual deployment of private implementations as peers gradually update their code. While a peer using an older version and one using a newer version would only be able to communicate using a non-private implementation, pairs of peers who use the newer versions should communicate using the private implementation by default. Consequently, they will be able to obtain the full privacy guarantees of our work for those messages.

We achieved backward compatibility with non-private versions through careful integration with interface definition languages such as Protobufs [Ste20]. Specifically, we did not alter the non-private versions of peer or provider routing, instead, we used a different Protobuf message to signal that the client uses the private version. This allowed

us to test both private and non-private versions simultaneously for correctness.

We highlight that PETs for peer-to-peer networks should not only be backwards compatible at a protocol level, but also from the perspective of being able to handle data that is already stored in the network. In particular, backwards compatible PETs in a distributed setting should not require existing content to be republished. Within our threat model and our design, we do not modify how peers *insert* existing content, including provider advertisements and content blocks, into the distributed file system. As an example of a violation of this property, we observe that Michel’s double hashing approach [Mic23a], which we discussed in ??, requires republishing existing content on IPFS.

Interoperability across PET implementations. During our prototyping stage, we want to be able to integrate different implementations of a PET into our system. We design our private interfaces to be sufficiently generic so that we can instantiate different PIR schemes, such as PAILLIERPIR and RLWEPIR, under them. While benchmarking existing PIR protocols for content retrieval, we find that some PIR implementations, out of the ones we analyzed in ??, were difficult to benchmark due to two reasons. First, they were difficult to build due to outdated dependencies. Second, they did not allow easily changing parameters such as the payload sizes. We consider these issues in the broader context of a surge of interest in Artifact Evaluation (AE), within the security and privacy field in particular [Vah25], as well as computer science in general [WTH+22]. The artifact evaluation process proceeds as follows. Artifacts of scientific papers are evaluated for one or more *badges*, depending on whether the source code has a license and can be accessed publicly (“Availability”), whether this source code can be built and run without errors (“Functionality”), and whether the output of the execution, including figures and tables, can be used to reproduce the results from the paper (“Reproducibility”) [Pri25]. Open-source repositories containing artifacts that have been evaluated by AE committees have been found to be under active development and garner interest *after* the artifact evaluation [WTH+22]. This leads us to reason that it is possible to address the aforementioned issues effectively through AE, such as by directing authors to set up tooling that automatically updates package dependency versions, and ensuring that the artifact is functional for a range of inputs. Apart from a well-funded, systemic focus on research artifact quality through artifact evaluation, we concur with Fischer et al.’s user study [FTG+24] in that standardization may initiate a drive to build robust PETs in academia, which can then be easily integrated by the industry.

Relation to research statement. In summary, we describe our process for, and challenges in, integrating our PET, namely Peer2PIR, within an existing codebase, namely IPFS. We outline how tests and distributed tracing can be used to identify what modules and functions of the codebase need to be modified to integrate the PET. We also discuss the needs for non-private and private implementations to coexist. We illustrate that integrating PETs may require handling both cryptographic and non-cryptographic pre-processing operations. We demonstrate that while a non-private implementation may only involve deterministic messages between entities, PETs often involve randomization, such as in the form of randomized encryption, and dealing with this randomization is non-trivial. We also emphasize that backwards compatibility within distributed file systems is not limited to the messages sent by older, outdated peers, but also entails prior content published by peers. Finally, we find that PIR implementations were not interoperable, and discuss how this can be addressed through the process of artifact evaluation.

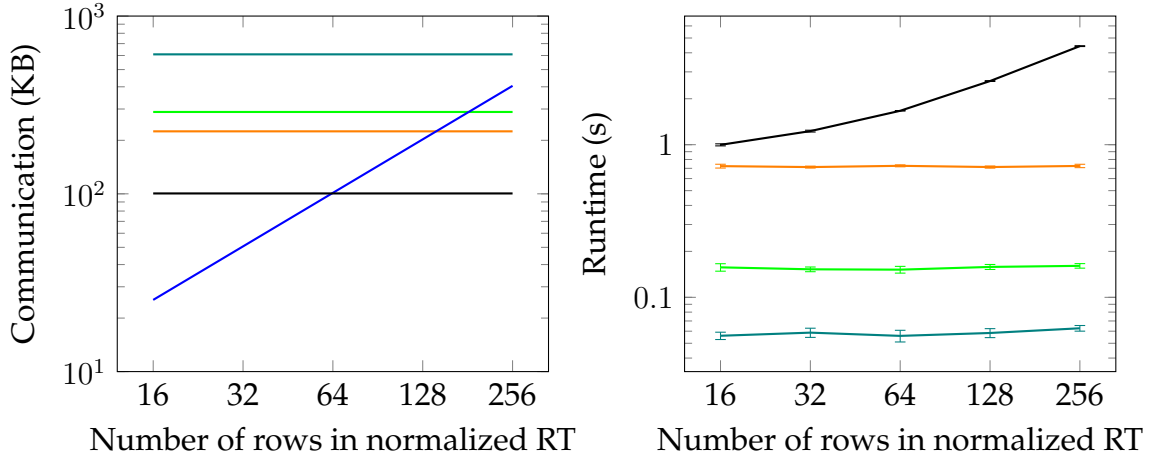
4.10 Evaluation

Experimental setup. We evaluate the total communication cost and the server-side runtime for our three use cases in ?????? respectively. We perform our peer and provider routing experiments on an AMD Ryzen 9 7900X with 12 cores and 32 GB RAM, whereas our content retrieval experiment was run on an AMD EPYC 7302 with 16 cores and 6 TB RAM. We report averages and standard deviations over $N = 10$ runs.

The *Runtime* metric denotes the server-side runtime to process a single private query in each use case. We selectively parallelized our bottlenecks for peer and provider routing to minimize runtimes. For our RLWEPIR schemes, it also includes the time to perform oblivious expansion. Client-side runtime is typically small in comparison to the server-side runtime. The *Communication* metric denotes the total round-trip network cost of one private query. For peer routing, we evaluate overheads only for a single “hop” in the iterative routing process; in a complete lookup, the client would ultimately contact a given target peer ID by querying closer server peers in each hop.

We hypothesize the following and examine the validity of these hypotheses in all three of our use cases:

- H1 In order of increasing communication cost, we expect PAILLIERPIR followed by RLWEPIR2, RLWEPIR3, and RLWEPIR. As the number of automorphism keys sent increases, the communication cost of our RLWEPIR schemes should increase.
- H2 In order of increasing server runtimes, we expect RLWEPIR followed by RL-



??

FIGURE 4.3: Communication and runtime overheads for private peer routing as a function of the number of rows in the normalized RT. Error bars are shown for the runtime plot. Note the logarithmic scale on the y-axes for both plots.

WEPIR3, RLWEPIR2 and PAILLIERPIR. We expect RLWEPIR2 to be slower than RLWEPIR3 since the server would have one fewer automorphism key and must perform additional automorphisms.

Private peer routing (????). As discussed in ??, in this use case we query a routing table of up to 256 rows with 1.5 KB entries. We recall from ?? that existing PIR schemes outlined in ?? are designed for large payloads. We thus only evaluate our PAILLIERPIR as well as our RLWEPIR, RLWEPIR2 and RLWEPIR3 schemes for the peer routing and provider advertisements use cases.

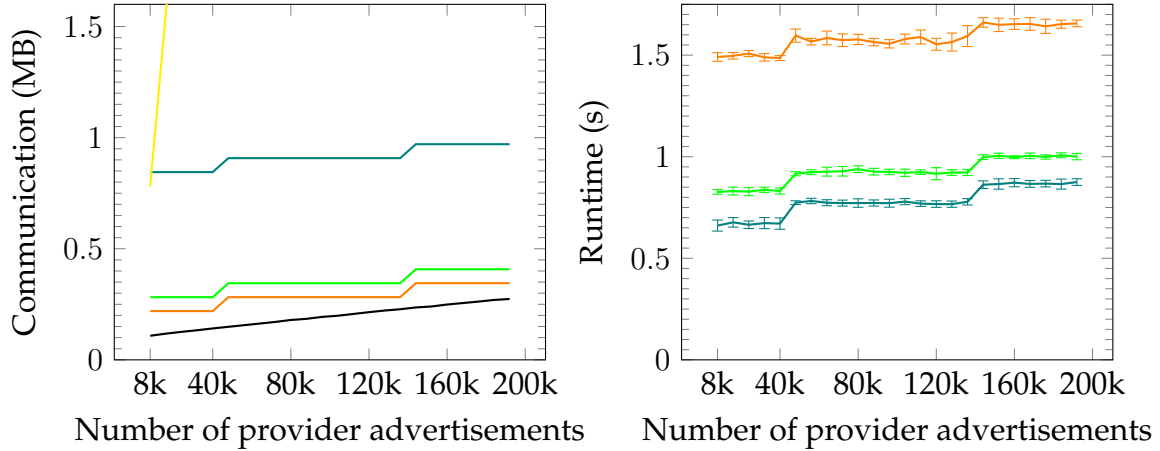
All hypotheses hold across the communication and computation cost plots for peer routing. In order of increasing communication cost, we find PAILLIERPIR followed by RLWEPIR2, RLWEPIR3, and RLWEPIR; thus ?? holds. Increasing the number of automorphism keys sent increases the communication cost across the latter three schemes, as expected. Second, each of our new PIR algorithms has a constant communication overhead, even as the number of rows increases. This is because the client issues its query assuming the worst case scenario: that the server’s RT has 256 buckets. Third, we analyze and plot the communication overhead when the server sends the entire normalized RT, in the TRIVIALPIR plot. PAILLIERPIR sends less data than TRIVIALPIR as long as the normalized RT has more than approximately 64 rows. We recall that normalizing the RT involves filling buckets with IDs and multiaddresses of peers from other

buckets, and thus the original RT is even smaller than the normalized RT. Nevertheless, sending the entire RT, in either its original or normalized forms, increases the risk of eclipse attacks, and is thus unsuitable for our threat model.

Finally, we find that RLWEPIR has the smallest server runtime, followed by RLWEPIR3, RLWEPIR2, and PAILLIERPIR, validating ???. Our RLWEPIR3 and RLWEPIR2 schemes are bottlenecked by the oblivious expansion step; *decreasing* the number of automorphism keys requires the server to perform additional automorphisms, which increases the server runtime. Our PAILLIERPIR scheme scales poorly with increasing number of rows, as expected. We recommend RLWEPIR3 for this use case, as it provides the best communication–computation tradeoff for our goal.

Private Provider Advertisements (????). We evaluate PIR protocols for the private provider advertisement use case (??). We randomly generate a table with the desired number of bins; this table corresponds to the output of the binning and symmetric encryption steps. As such, we expect the binning and symmetric encryption steps to incur minimal latency, as we discuss in Section ??. We use 4096 bins for our RLWE-based protocols, since the client can encode a query as large as the polynomial modulus degree $N = 4096$ within one RLWE ciphertext with no extra cost. We use 256 bins for PAILLIERPIR. These bins store increasing numbers of provider advertisements, varying from 8 k–192 k. We conservatively overestimate the size of each provider advertisement to 100 B. Thus, our provider advertisement store is between 800 kB–18.75 GB.

We again observe that hypotheses ?? and ?? hold from ?? and ?? respectively. First, we note that the communication costs and runtimes of our RLWE schemes are in the form of a step function over the number of provider advertisements; these overheads increase sharply at approximately 40k and 140k advertisements. As bins fill with more provider advertisements, the server needs to compute additional ciphertexts to encode a bin in a PIR response, resulting in the sharp increases in both overheads. Second, we note that the communication cost of PAILLIERPIR increases linearly with the number of provider advertisements. The *number* of rows (bins) for PAILLIERPIR, and thus the size of the PIR query, is identical to the peer routing case. As we increase the *size* of each row, the size of the PIR *query* remains constant, whereas the size of the PIR *response* increases linearly, and thus the total communication cost increases linearly. PAILLIERPIR is an order of magnitude slower than RLWEPIR, taking between 0.5–6 minutes, which is consistent with existing results [ALP+21]. Thus, we do not plot its runtime in ??, and do not recommend it for this use case. Finally, we consider a trivial symmetric PIR protocol, namely TRIVIALSPIR, which is discussed in Section ??); this protocol quickly exceeds



??

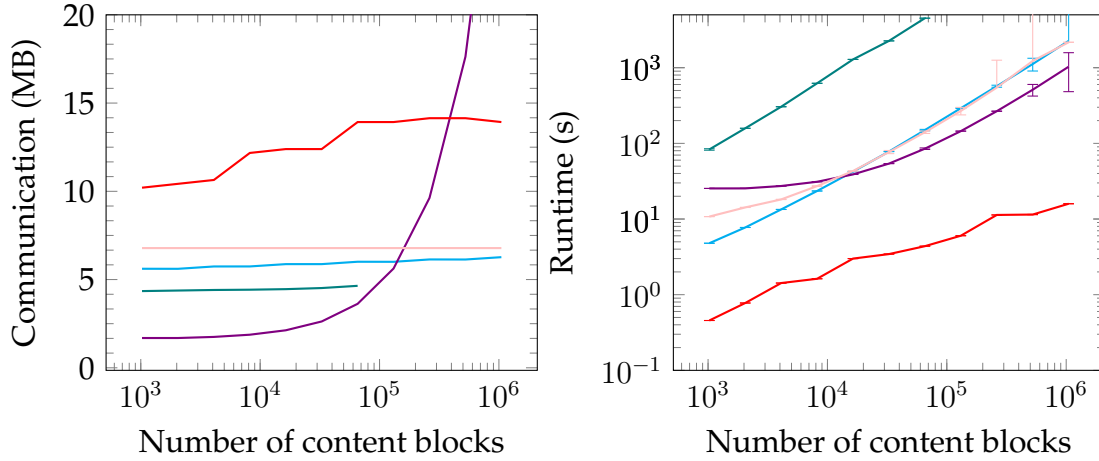
FIGURE 4.4: Communication and runtime overheads for private provider advertisements as a function of the number of provider advertisements. Error bars are shown for the runtime plot.

the communication costs of our RLWEPIR and PAILLIERPIR protocols. In conclusion, comparing the variants of RLWEPIR, we again recommend using RLWEPIR3.

Our recommendation also extends to the PRIVATEWANTHAVE step (??). We recall that the PRIVATEWANTHAVE step also uses ??, but involves marginally smaller payloads, namely list indices instead of peers' multiaddresses, as discussed in ??. Thus, this step would incur marginally smaller communication and computation overheads as compared to those in ??? respectively.

Private Content Retrieval (????). Finally, we examine the case for large content blocks sent during the PRIVATEBLOCK step. In the PRIVATEWANTHAVE step, the server will output the protocol to use for the PRIVATEBLOCK step. So, we recommend two different protocols for this use case: one for a lightweight server with few content blocks and one for a heavyweight server with more content blocks. PAILLIERPIR and RLWEPIR have been designed for the peer routing and provider advertisement use cases and provide a poor communication-computation tradeoff for this case. Thus, we exclude these options and compare the remaining options from ?? for this step.

Since implementations of some of these schemes do not support a 256 KB payload, we extrapolate communication and computation overheads by multiplying the overheads for the largest supported payload size by the number of plaintexts required to



??

FIGURE 4.5: Communication and computation overheads for private content retrieval as a function of the number of content blocks. Error bars are shown for the runtime plot. Note the logarithmic scale on the x-axis for both plots, as well as the logarithmic scale for the y-axis for the runtime plot.

send a 256 KB payload. We run these implementations as-is, and do not optimize or parallelize them. We use the SpiralStreamPack variant of Spiral [MW22], as it is optimized for large payloads.

We observe the following from ?????. Spiral enables reasonably low runtimes (between 0.4–20 s) for the ranges of database sizes that we consider, while incurring a high communication overhead. SealPIR has almost half as much communication overhead as Spiral, and is at least an order of magnitude slower. At the other extreme of the communication–computation tradeoff, FastPIR has the lowest communication overhead until 100,000 blocks, and this overhead then scales linearly for larger databases. FastPIR is noticeably slower than other protocols until 10,000 blocks.

We thus recommend Spiral for the PRIVATEBLOCK step in Bitswap, due to its significantly lower computation overhead. Alternately, to reduce communication costs, lightweight peers that hold less than 10,000 blocks may opt for FastPIR, whereas heavyweight peers can opt for SealPIR. Since heavyweight peers are typically backed by large organizations [Pro24c], it is reasonable that they may procure servers to reduce runtimes as required. As discussed in ??, the client can include multiple content identifiers within one PIR query and the server can include multiple content blocks within one PIR response. This reduces multiple round trips between a client and a server, when the latter stores multiple content blocks of the same file. In particular, the client only

needs to send their public key once, and this public key can be reused by the server to process multiple PIR queries. Thus, the communication cost of sending a large public key (13 MB for Spiral) can be amortized across multiple PIR queries.

4.11 End-to-end latency and churn analysis

In ??, we evaluated the efficiency of our private algorithms for our use cases, considering only a single “hop” or round trip between a client and a server peer. In this section, we analyze the end-to-end latency for each use case, extrapolating from our evaluation. We also evaluate the impact of churn on our private algorithms.

4.11.1 End-to-end latency analysis

We summarize the four steps that a client engages in, in order to retrieve a content block with a given CID. In the first step, the client opportunistically searches for the content block, by running the WANTHAVE step *in parallel* among its neighbours. The total computational latency for our private counterpart, namely the PRIVATEWANTHAVE step, would be similar to that in ?. The client proceeds only if the first step fails, which is common in practice. In the second step, the client determines the peer IDs of providers of that block by simultaneously running peer and provider routing queries. As we describe below, these queries are crafted such that the peer routing query ultimately leads the client to a peer who will be able to answer the provider routing query successfully. In the third step, the client finds the provider peer’s multiaddress, by running another peer routing query for peers closest to the provider peer ID. In the final step, the client runs both WANTHAVE and BLOCK steps of the Bitswap protocol with the provider peer. We represent the network round trip time (RTT) for hop i by τ_i . The end-to-end latency of the third step is thus given by the end-to-end latency of the peer routing query, whereas that of the second step is given by the end-to-end latency of peer and provider routing queries run simultaneously. We first analyze the simpler case, namely the end-to-end latency for a peer routing query, and then compute the end-to-end latencies of simultaneous peer and provider routing queries, relating these latencies to that of the second and third steps. Finally, we analyze the end-to-end latency of the final step.

For peer routing, each *hop*, which takes the client closer to the target peer ID, is sequential, whereas the client contacts up to k peers in parallel within one hop. Thus, the total server-side computational latency for private peer routing would be the path length multiplied by the latency for one hop (≈ 60 ms from ?). The client’s overheads to compute the PIR query and process the PIR response for each hop would also similarly

scale with the path length. Our path length does not increase over standard IPFS in practice (??). For an IPFS network of size $n = 200,000$ [TRT+22], the average path length in Kademlia is $\log_2(n) = 17.6$ [MM02], and so, the total server-side computational latency of a private peer routing query would be ≈ 1 s. The client would experience this latency for the third step, namely when running a peer routing query to learn the multiaddress of a provider peer. The client would *also* incur this latency for the second step, that is, in order to reach a peer that has a provider advertisement; however, as we outline below, the end-to-end latency for the second step is upper bounded by that of the provider routing query, rather than the peer routing query.

In the second step, the client determines the peer IDs of providers of a block with CID c as follows. The client runs a peer routing query for peers closest to $\text{Hash}(c)$ and simultaneously, runs a provider routing query for providers for c . Here, the peer routing query reveals whether the queried peer knows (the IDs and multiaddresses of) peers who may host a provider advertisement for that CID. The provider routing query reveals whether the queried peer knows a provider for that CID. From [????], we observe that provider routing takes longer ($t_2 \approx 600\text{--}900$ ms) than peer routing ($t_1 \approx 60$ ms). At the end of a single hop, through the peer routing query response, which is obtained at $t = t_1 + \tau_i$, the client would know the IDs and multiaddresses of peers who are more likely to have the provider advertisement for c than the server. Nevertheless, the client will not know whether the server itself stores the provider advertisement for c until it obtains the *provider* routing response, which is obtained at $t = t_2 + \tau_i$. Instead of waiting ($\approx 540\text{--}840$ ms) for a private provider routing response for each hop, the client can proceed with the next hop of the private peer and provider routing queries. This design decision avoids bottlenecking the computational latency for each hop to the latency of the slower query, namely the private provider routing query, at the cost of redundantly querying other peers in case the client would have found a provider in a previous hop. The end-to-end latency includes both the total computational latency and the network round trip time (RTT) for each hop. The end-to-end latency T to find a provider peer ID would then be given by:

$$T = \underbrace{\log_2(n) \cdot t_1}_{\text{Peer routing}} + \underbrace{\sum_{i=1}^{\log_2(n)} \tau_i}_{\text{Network RTT}} + \underbrace{\max(\tau_i) + t_2}_{\text{Provider routing}}$$

We note that waiting for the provider peer response for each hop would result in a larger end-to-end latency T' :

$$T' = \underbrace{\log_2(n) \cdot \max(t_1, t_2)}_{\text{Peer \& provider routing}} + \underbrace{\sum_{i=1}^{\log_2(n)} \tau_i}_{\text{Network RTT}}$$

For content retrieval, each file is decomposed into a Merkle DAG of content blocks (??, [Gre23; Pro24a]). The height of this DAG is logarithmic in the file size, and it bounds the number of sequential retrievals for which the latency of our private content retrieval algorithm may accumulate. All blocks within the same level of the tree can be fetched in parallel from the same or different provider servers.

4.11.2 Churn analysis

We consider two types of churn in the network: node churn, namely when a peer enters or leaves the network, and content churn, namely when a provider adds or removes a content block from the network. Trautwein et al. [TRT+22] empirically evaluate node and content churn in IPFS. They find that most peers are online for less than 8 hours; few peers are long-lived.⁵ In a separate study, based on the rate at which provider advertisements are updated, Trautwein observes that up to half of the advertised content blocks are advertised for at most one day, which stems from the default provider advertisement expiry period of a day [Tra22c]. Less than half of the remaining content blocks are advertised for at most two days and so on, leading to a heavy-tail distribution of the content churn rate for provider advertisements and content blocks.

We outline which of these churn rates are applicable to each of our algorithms, and describe how they are impacted. In particular, we show that each algorithm succeeds under the applicable churn rate, and we analyze whether it would incur a higher latency under churn. At a high level, we argue that the latencies of each of our three private algorithms are negligible in comparison to the rate of the applicable churns. Consequently, the likelihood that our algorithms will fail a client peer in reaching a target peer or in retrieving a target CID, due to churn, would be similar to that in the current IPFS deployment.

Private peer routing. Peer routing algorithms can be impacted by node churn, but not by content churn. There can be three potential types of consequences of node churn. First, as peers join the network, the routing table at each server peer may update; we discuss below why our private peer routing algorithm would *not* experience higher latency, at each such peer, than without node churn. Second, as peers leave the network, some of them may have been in path to the target peer; this does not impact the convergence of our private peer routing algorithm, as we describe below.

⁵Trautwein et al.’s CDF of node uptimes [TRT+22, Fig. 8] approximates a Pareto distribution, which is heavy-tailed. This corroborates Maymounkov and Mazières’ finding in their original Kademlia paper that peers who are online for x hours are more likely to stay online for $x + 1$ hours [MM02, Section 2.1].

Regarding the first point, despite the fact that node churn is high, we observe that routing tables are not updated as frequently. This is due to the fact that Kademlia favours retaining long-lived nodes in its routing table [MM02, Section 2.1]. Nevertheless, even if peers are added to the routing table, we note that re-normalizing it involves no cryptographic operations and thus has negligible impact on the latency incurred by one server peer. In other words, we expect that the latency of our private peer routing algorithm, under churn, would be close to that without churn, which is depicted in ??.

For the second point, we observe that the original peer routing algorithm has specifically been designed to handle high churn. In particular, Maymounkov and Mazières' original proof of convergence relies on choosing a sufficiently high k such that the likelihood of k peers all being offline at the same time is negligible [MM02, Section 3]. Similarly, Trautwein et al. argue that their choice of a high value for k , namely $k = 20$, accounts for short-lived peers, such that storing k peers in each bucket in the routing table leads to reasonably low number of hops to discover peers [TRT+22, Section 5.3]. As long as the original peer routing algorithm converges, we show through our proof of convergence (??) that our private peer routing algorithm converges with at most a constant number of additional hops. We estimate above that our peer routing algorithm would add an overhead of approximately 1 s to the end-to-end latency; this is negligible in comparison to the node churn rate, so the likelihood of a node leaving the network by the time a lookup is done, would be similar in both private and non-private cases.

Private provider routing. Our provider routing algorithm can be impacted by node churn, as the peers storing the advertisement leave the network. It can also be impacted by content churn, as the advertisements themselves expire in a day, by default, and new advertisements are inserted. We argued above that our private peer routing algorithm is resilient to node churn; the same argument applies for finding provider peers privately. Regarding content churn, a new provider advertisement is processed through four main steps: it is joined with the address book, then the result is symmetrically encrypted, and then binned through cryptographic hashing, and the bin is then re-encoded for PIR. The first step is a non-cryptographic operation while the next two steps involve commonly used, efficient cryptographic operations; thus, we expect the first three steps to incur negligible latency.

The runtime to encode one bin scales with the number of plaintexts that need to be formed, which is proportional to the size of the largest bin. So, when a provider advertisement is added to the largest bin, other bins will also need to be re-encoded.

We observe that different bins can be encoded in parallel, that is, each plaintext for a bin can be encoded in parallel. However, in most cases, we will only need to re-encode the bin into which the provider advertisement is placed. In this case, since we append the provider advertisement to the bin, adding one provider advertisement can result in at most a constant number of new plaintexts, and thus, incur a constant overhead. So, we expect that the latency of private provider routing under node churn can be made as close as possible to $\Theta(1)$, potentially with sufficient parallelization of the encoding step.

Private content retrieval. Our content retrieval algorithm incurs a non-negligibly high latency (order of minutes) over the baseline non-private content retrieval algorithm. Nevertheless, we note that this latency is still very small in comparison to the content churn rate. Thus, we can safely conclude that there would be sufficient time between successive updates to the content store to re-encode the table for PIR schemes. Again, as was the case with the provider advertisement store for provider routing, this re-encoding step can be parallelized.

4.12 Conclusion

We address the problem of a peer privately retrieving content and routing information from other peers within IPFS. We provide three scalable algorithms to enable a peer to privately retrieve routes to another peer, to discover peers providing desired content, and to fetch this content from these peers. We integrate our algorithms within the IPFS design, ensuring that they handle constraints unique to it. We minimize the amount of information exposed to peers while they act as a server and answer other peers' queries, and thus we realize Hoepman's MINIMIZE strategy. Specifically, the aforementioned algorithms cumulatively hide who a peer is trying to contact, and which content a peer is trying to retrieve. Our algorithms rely on private information retrieval protocols, and so we systematically analyze how recent protocols can handle the three aforementioned problems. We propose two CPIR protocols that combine techniques from cryptographic folklore and address a gap in the design space. In doing so, we operationalize the HIDE strategy for distributed hash tables. Additionally, we evaluate the correctness and runtime overheads of our algorithms. We also highlight privacy engineering takeaways that can be applied to the integration of PETs into distributed architectures.

Chapter 5

Prose

Modern back-end software stacks are often implemented in the form of a distributed architecture consisting of microservices. Although recent work [DPI+22; LCL+21] addresses security challenges such as authorization and auditing for such architectures, privacy challenges such as data minimization are yet to be addressed. To demonstrate compliance with privacy legislation, privacy engineers need to integrate privacy design strategies into their software development lifecycle. To this end, we build Prose, a Privacy Observability and Enforcement framework. We provide a privacy policy workflow for privacy engineers to effectively realize Hoepman’s MINIMIZE strategy (??) across a large architecture of microservices. Our framework extends existing observability tools to derive dynamic profiles of each service, and efficiently composes them to arrive at a system-wide privacy profile. Thus, Prose informs privacy engineers of the data processing practices of their back-end system that consists of microservices. Privacy engineers can use our privacy profiles to populate conventional privacy policies, in order to ensure that the policy text reflects the back-end code. Thus, Prose can be used to realize the INFORM strategy. Additionally, Prose enables privacy engineers to iteratively prototype target privacy policies of varying strictness, with an eye towards data minimization. Our framework detects whenever a service profile violates a target privacy policy, and Prose also demonstrates compliance with this policy, effectively realizing the DEMONSTRATE strategy. Prose also enforces the target privacy policy in a production environment, thereby operationalizing the ENFORCE strategy. We motivate privacy engineers’, as well as software engineers’, requirements for Prose. We present the design of our system, which fulfills these requirements. We evaluate the effectiveness and efficiency of our system against use cases derived from reported privacy legislation violations.

5.1 Introduction

Modern back-end software stacks have evolved from monolithic architectures to distributed architectures consisting of microservices [Lou21]. Each microservice focuses on a certain business functionality or purpose of use and is independently developed by a team of software engineers. For example, an online e-commerce shop [Goo23] might have CHECKOUT, SHIPPING, AUTHENTICATION and ADVERTISING services. Cloud computing platforms [Goo24a; Ama24b] facilitate the deployment of microservices using technologies such as Kubernetes [Kub23] and Istio [Ist23b]. Microservices are ubiquitous; big tech companies such as Google [McL16], Amazon [Ama19], Netflix [Var21], and Spotify [Gol15] drive the development of microservices-based architectures, and contribute to the aforementioned cloud computing technologies. Additionally, public and private sector enterprises around the world increasingly use microservices and adopt these technologies [IBM21; Clo23].

Within this architecture, users' personally identifiable information (PII) is collected by multiple microservices, for different purposes. PII collected by one microservice for one purpose, can be reused by other microservices for *secondary* purposes; that is, purposes other than the purpose for which it was collected. For example, Twitter obtained its users' phone numbers for the purpose of improving security, such as two-factor authentication, and then used them for targeted advertising [Ala22]. Alternately, a microservice may share users' PII with third parties for secondary purposes. For instance, BetterHelp, an online counselling service, shared their users' PII such as email addresses, their mental health history, and their usage of the service, with Facebook and Snapchat to target advertisements [Fai23].

Processing and sharing users' data for secondary purposes violates the privacy design principles of *data minimization* and *purpose limitation*, which were discussed in ???. Recent academic literature on distributed web back-end architectures only focuses on security problems such as authorization [LCL+21] and auditing [DPI+22], rather than data minimization. Privacy engineers are left with ad-hoc tools to monitor and mediate data flows in a complex architecture of thousands of microservices. We address the question: *Can we design a framework for privacy engineers and software engineers to use to systematically and proactively realize data minimization and purpose limitation for a modern microservice architecture and software development lifecycle?*

Since multiple services process users' PII in a microservice architecture, for privacy engineers to provide meaningful privacy guarantees over users' data, they require knowledge of the entire distributed back-end system; that is, how each microservice consumes and shares users' PII. The privacy engineer can then prototype various strict

target privacy policies. They need to efficiently determine which services violate the policy, and how. Armed with this knowledge, privacy engineers can work with software engineers who maintain the violating services, to fix them. In each release of the system to production, the privacy engineer should be able to iteratively improve the entire system towards complying with a target privacy policy. Privacy engineers should be able to prioritize fixing certain violating services before the next release. They should be informed of the privacy policy that the entire system would effectively comply with, if only some violating services were fixed.

To summarize, we build a PPrivacy ObServability and Enforcement framework, which we refer to as *Prose*. We provide the following contributions:

- We discuss background on the distributed microservices architecture (??), orchestration frameworks (??) and the software development lifecycle (??). We then formulate general requirements for the system to operate within a microservice architecture and for ease of use by software engineers and DevOps engineers (??).
- We present a privacy policy workflow that is integrated into the software development lifecycle (??). Privacy engineers who use *Prose* would follow this workflow. Moreover, our workflow may be of independent interest as it can be generalized to other distributed back-end architectures. We draw functional requirements catered towards privacy engineers (??). We contextualize two types of privacy violations, namely secondary purpose-of-use and data sharing violations, within the microservice architecture (??).
- We describe our threat model, which outlines our trusted computing base, as well as how the aforementioned users are expected to behave in the context of the aforementioned privacy policy workflow (??).
- For our design, we reuse components that are commonly used when deploying microservices, namely service proxies, authorization agents and distributed tracing (??). Our design can be divided into a data plane and a control plane.
- We then present the design of our data plane, which carefully combines service proxies and authorization agents to form a system of distributed runtime monitors, in order to detect and prevent the aforementioned violations of a target privacy policy (??).
- We then describe our control plane design (??). We use distributed tracing to convey relevant information about policy-compliant and violating data flows, from the data plane to the control plane. The control plane uses this information to maintain data structures that we define in the privacy privacy workflow. Ultimately the control plane helps privacy engineers get a bird's-eye view of the data processing practices of the entire system, and manage violations.

- We evaluate the effectiveness of our framework in detecting privacy policy violations, as well as the latency overheads it incurs on end-to-end requests (??).

We contrast our system with related work in ?? and conclude in ??.

5.2 Background

We describe the architecture of distributed microservice applications, the orchestration frameworks used to deploy them, and their software development lifecycle.

5.2.1 Distributed microservices architectures

Deployments of microservices could vary in scale, from tens [Goo23] to thousands of microservices [Var21]. These services are written in different languages; that is, a distributed microservices architecture is polyglot. Containerization platforms, such as Docker, may be used to run one or more services in a reproducible manner inside a container. Each microservice focuses on a certain business functionality, and may store associated state in a database; for example, an online shop [Goo23] might have CHECKOUT and SHIPPING services with their associated databases. This architecture reduces coupling between microservices, and thereby supports independent evolution, building, testing and deployment of microservices [New20, Chapter 1], in contrast with monolithic architectures, which require all modules to be rebuilt and redeployed with a change in any module.

Microservices communicate with each other using application layer protocols, such as HTTP. Each microservice provides an Application Programming Interface (API) that other microservices can access. A microservice can provide multiple API endpoints, wherein each API endpoint has a unique URL. Each API endpoint URL can be accessed through the application layer protocol(s) supported by the microservice [Clo24d]. An HTTP API endpoint can be accessed using HTTP requests with methods such as GET, POST and others. The microservice container exposes ports and accepts application layer traffic, such as HTTP requests on that port. The website’s front-end application running in a user’s browser requests content from a gateway API service, which routes requests to appropriate back-end microservices.

5.2.2 Orchestration frameworks

Orchestration frameworks can be used to scalably deploy microservices. Popular orchestration frameworks [Clo24c] include Kubernetes [Kub23], Docker Swarm [Doc25a],

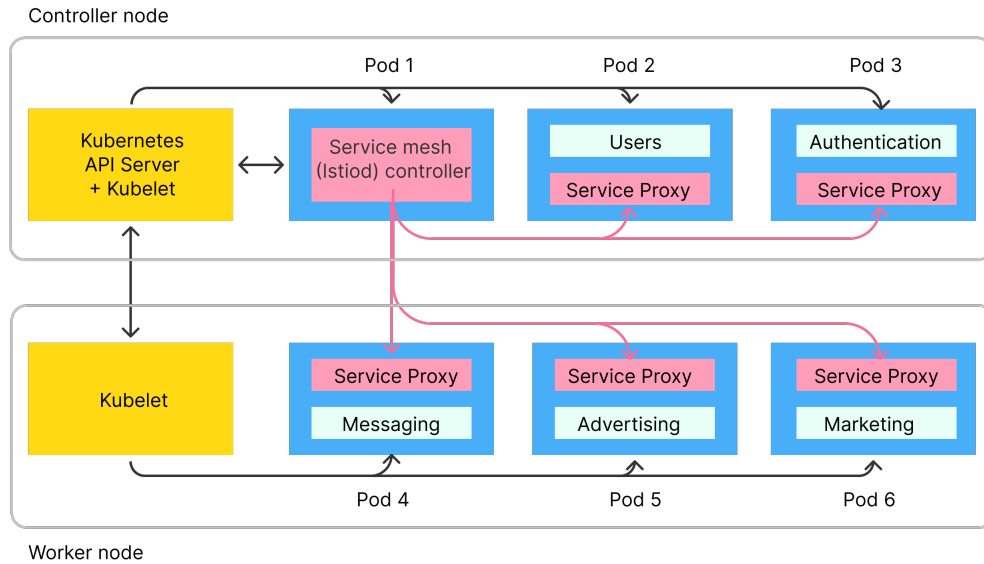


FIGURE 5.1: Depicts five microservices named Users, Authentication, Messaging, Advertising and Marketing (in light blue), which are managed by the Kubernetes API Server and Kubelet (both in yellow). A service proxy (in pink) is alongside each service container; the service proxies are managed by the Istio service mesh controller (named Istiod and also shown in pink).

Apache Mesos [Apa24] and Nomad [Has25]. We focus on the Kubernetes architecture, which we outline below, though our design can be generalized to other orchestrators.

A cluster consists of physical or virtual machines, which are known as *nodes*. A node can be classified as a controller or a worker. Controller nodes manage worker nodes. A *pod* is used to run one or more containers. All containers in a pod share the same virtual network namespace [Kub24]. A node runs multiple pods. Since pods are ephemeral and their IP addresses can change, Kubernetes *services* (not to be confused with the microservice itself) enable communication between pods in the cluster and from clients outside a cluster. All pods that are involved in resolving an end user’s request are referred to as being *in path* of the request.

A *Kubelet agent* on each node monitors the state of resources, such as pods or Kubernetes services, on that node. The agent regularly propagates the status of these resources to one of the *Kubernetes API servers*, which run on each controller node. Internally, the Kubernetes API server runs a controller for each resource, which determines how to reach the desired state of the resource from its current state. For example, the REPLICASET controller ensures that the desired number of replicas of each pod exist across all worker nodes in the cluster. The Kubernetes orchestrator can be extended by defining

custom resources and controllers for those resources. For example, service meshes such as Istio [Ist23b; Ist23a], which we describe in ??, extend the Kubernetes orchestrator.

We collectively refer to pods that manage Kubernetes native and custom resources, such as the Kubernetes API server, as the *control plane*. The control plane manages the *data plane*, which consists of pods in path of an end user's request. At a high level, Kubernetes separates the data and control planes, by assigning pods in these planes to the worker and controller nodes respectively. In other words, controller nodes typically only run the control plane pods and the worker nodes run the data plane pods. When a worker node fails, the Kubernetes API server directs another node to create pods that were running on the failed node. Additionally, high availability of the Kubernetes API server is achieved through multiple controller nodes. We illustrate an example of five microservices that constitute the back-end of an e-commerce website, when deployed within Kubernetes, in ??. (We present the communication between these microservices, and between these microservices and external hosts later in ??, and discuss this further in ??.)

5.2.3 Software Development Lifecycle (SDLC)

Our system involves the following parties: service developers, DevOps engineers and privacy engineers. Service developers write and test individual microservices. They regularly publish a microservice container image to an image registry. DevOps engineers are responsible for deploying these microservice images. Privacy engineers are tasked with ensuring compliance of all microservices with a target privacy policy, as we discuss in ??. We describe below the processes and tools that software engineers and DevOps engineers regularly engage with, in the context of a microservice architecture.

Continuous Integration and Continuous Delivery (CI/CD) pipelines are widely used today [GDM22], including within modern microservices architectures [Azu25]. Popular source code repository hosts for the Git version control system, including Github and Gitlab, provide CI/CD pipelines, named Github Actions [Git24d] and Gitlab Pipelines [Git25b] respectively. A Continuous Integration (CI) pipeline ensures that before a feature is merged into the main version, the feature's source code is automatically linted and the microservice is rebuilt. The CI pipeline commonly consists of language dependent components for code linting, static analysis, compiling, security vulnerability scanning and testing [Git24a; Git25c; Jac25].

Service developers include different tests within the CI/CD pipeline [WLS+21]. For example, unit tests for a new feature are often included in the CI pipeline, to ensure that code merged into the main branch has been tested at least once [Git24a]. Microservices

architectures also benefit from integration and end-to-end tests, which may be geared towards functionality or performance [CF14]. End-to-end testing frameworks, such as Cypress [Cyp24] or Playwright [Mic25], can be integrated into the CI pipeline. (An obvious benefit of testing is to detect bugs before the product is deployed to a customer; an extensive suite of tests helps meet quality assurance requirements. Additionally, the presence of different types of automated tests is also considered a measure of the maintainability of the code — a lack of testing has been known to contribute to technical debt [TAV13].) Once the new feature successfully passes all tests in the CI pipeline, the software developers merge the feature into the main branch. The CI pipeline may then publish the microservice image onto an image registry, such as DockerHub [Doc25b] or Github Container Registry [Git25a]; the CI pipeline may sign the service image automatically, using a private signing key of the service developer who merged the feature.

Cloud computing platforms, such as Google Kubernetes Engine (GKE) [Goo24a; Git24c], Amazon Elastic Kubernetes Service (EKS) [Ama24b], or Azure Kubernetes Service (AKS) [Git24b] can be used to deploy these container images. Once an image is built through the CI pipeline, DevOps engineers address the question: *how do we deploy this image into a testing and/or production environment within a cloud computing platform (e.g., GKE), that runs an orchestrator (e.g., Kubernetes)?* Configuration files specify how various resources, such as Kubernetes pods, need to be configured, in order to deploy the microservice. In the context of deploying microservices to a cloud computing platform, the Continuous Delivery (CD) pipeline instructs the orchestrator to configure its resources using these files, along with the new container image.

As an example of a configuration that is relevant for Prose, the CD pipeline may automatically label each pod with the purpose of the microservice container that it will execute. The purpose of the service can be hard-coded into the pod configuration file. Alternately, it can also be inferred dynamically, based on the role of the service developer or team who publishes and signs the container image. The purpose of use of the service can also be automatically derived from the role of the team publishing the container image; both of these techniques have been used in prior work [SGD+14].

The aforementioned CI/CD pipelines enable deploying these microservice container images to cloud computing platforms. Additionally, these configuration files, which describe the infrastructure, can themselves be treated as code, and managed through version control systems, leading to the *Infrastructure as Code* or *GitOps* paradigm [Git25d; Red24]. Instead of the CD pipeline instructing the orchestrator to use the configuration files to redeploy the new microservice image after each merge, GitOps-based tools, including Flux [Flu25] and Argo [Arg24], monitor these files for changes, and relay these changes to the orchestrator, to redeploy the new microservice image.

A large architecture with many microservices may fail, resulting in user-facing errors, or slow responses. Observability tools are commonly used by DevOps engineers [Clo21; LPX+22] to measure whether service level agreements are met and identify such failure incidents [BJPM16, Chapter 4]. Alert management and incident response tools, such as PagerDuty [Pag25], alert DevOps engineers and service developers of such incidents. Observability tools also assist DevOps engineers in efficiently identifying the root causes of these incidents [MF21, Chapter 19]. Distributed tracing is an observability tool that helps DevOps engineers trace the chronological order in which API endpoints of different services are called, in responding to a user’s request [PSMI20]. Consequently, they can identify any API endpoints that cause errors or incur high latency.

Prose is designed such that DevOps engineers can easily integrate it into their existing CI/CD pipeline for their microservice architecture with their preferred orchestrator. In turn, Prose relies upon an existing end-to-end test suite, in that it is intended to run within a CD pipeline, during end-to-end testing. Prose also expects the source code to support observability tooling, and in particular, distributed tracing, as we outline in ?? and detail in ??.

5.2.4 General requirements

We outline architectural requirements that Prose should follow. First, DevOps engineers should be able to scalably deploy our framework using an orchestrator. So, Prose should be built in the form of containerized components. In turn, DevOps engineers can continue to use their existing CI/CD pipelines for orchestrators for cloud computing platforms, as discussed above.

GR1 *Ease of scalable deployment.* The framework should be built with containerized components that can be scalably deployed using CI/CD pipelines for cloud computing platforms.

We use the notion of a data plane and a control plane, which were described in Section ??, to structure our design. In other words, Prose consists of a data plane and a control plane, both of which are managed by the orchestrator.

Second, different microservices are developed by different teams of software developers, and these microservices could be written in different languages. Prose should support a polyglot architecture.

GR2 *Polyglot:* The framework must handle polyglot microservices architectures.

To satisfy this requirement, Prose effectively conducts dynamic analysis rather than static analysis.

As we describe in Section ??, service developers and DevOps engineers need only include two manual source code modifications for our framework to operate: purpose-of-use labelling for microservice container images and distributed tracing instrumentation for microservice source code. Importantly, we do not require any other changes to the microservices' source code for our framework to operate.

GR3 *Modifications to microservice source code*: The microservice source code may need to be modified in order to satisfy a target privacy policy. However, for the framework itself to work, changes to the microservices' code should be limited to distributed tracing instrumentation and purpose-of-use labelling.

Fulfilling this requirement enables our system to be integrated and adopted easily.

While meeting ??, Prose should follow a scalable design, and so it should incur minimal impact on the efficiency and availability of the data plane. We measure efficiency in terms of the latency overhead experienced by end users. Since users' requests go through the data plane as discussed in Section ??, our system should reduce the latency overhead in the data plane. (The control plane, which will be used by the privacy engineers, will run asynchronously with respect to users' requests. Therefore, as a design decision, our system should emphasize data plane, rather than control plane, latency reduction.)

GR4 *Efficiency*. Prose should incur, at most, a small overhead for the end-to-end latency when the data plane handles moderate loads.

5.3 Privacy policy workflow and violations

We first describe our privacy policy workflow (??), which is integrated within the software development lifecycle (??). Our workflow enables privacy engineers to realize data minimization iteratively, while keeping in mind the fact that service developers may have limited time to improve their services towards data minimization. We outline privacy engineering requirements for our system, which are necessitated by our workflow (??). We then delve into a running example, describing the type of privacy policy violations in scope for our system (??).

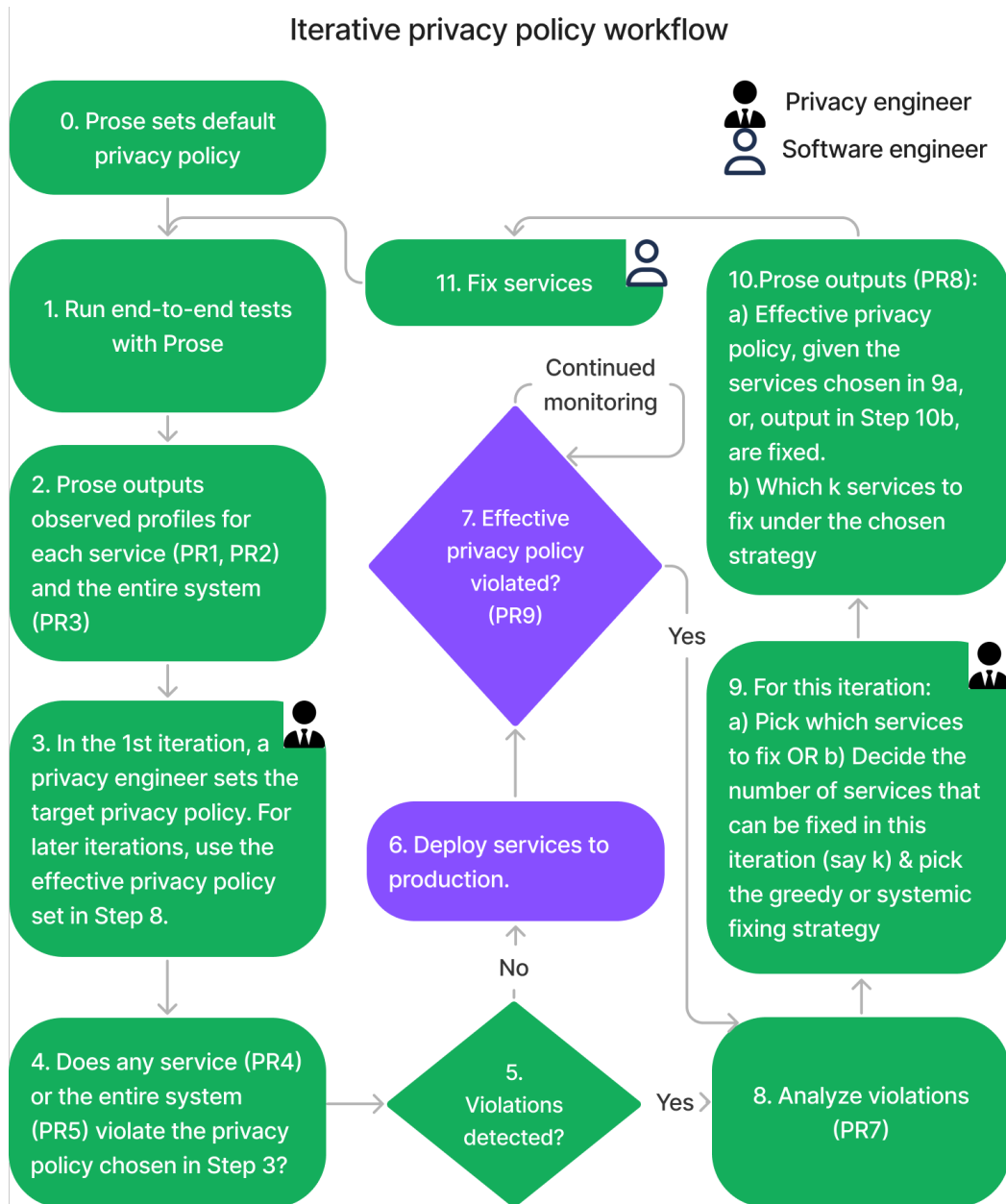


FIGURE 5.2: Iterative privacy policy workflow: Green and purple nodes represent the testing and production environments respectively. Steps are marked with relevant privacy engineering requirements (discussed in ??). We note that ?? is not represented directly by a step; instead, this requirement corresponds to rerunning Steps 3–5, under a different target privacy policy, without rerunning Steps 1–2.

5.3.1 Iterative privacy policy workflow

We present our workflow in ???. While describing this workflow below, we refer to its steps, as well as the associated privacy engineering requirements (described in ?? and shown by the label PR). As we outline in ??, we expect privacy engineers to define a set of all possible purposes of use across all microservices, as well as to define a set of PII types, such as credit card numbers, addresses, names, etc., that they seek to detect.

For privacy engineers to provide meaningful privacy guarantees over users' data, they need to know how each microservice handles that data. We define an *observed profile* of each microservice as an association of each of its API endpoints with the PII types that the endpoint processes. In particular, this profile specifies which PII types the endpoint consumes, as well as any external domains that this endpoint shares PII types with. After each microservice is updated, rebuilt and retested (Step 1 in ??), Prose should efficiently (re)compute this profile (Step 2; ??, ??).

Privacy engineers also need to determine how the entire system as a whole handles users' data. We extend the definition of an observed profile to apply to the entire system, rather than just to individual microservices. Observed profiles of multiple microservices can be composed together, to form a *system-wide observed profile*. The system-wide observed profile encapsulates the purposes for which each PII type is processed throughout the system, and any third parties with which it is shared. Prose should efficiently compute this system-wide observed profile (Step 2; ??).

While a system-wide observed *profile* reflects the current state of the system as far as privacy is concerned, a target privacy *policy* reflects the desired state of the system. Each *clause* of a privacy policy allows a PII type to be processed, or shared with a specified external domain, for a given purpose. A privacy *violation* arises when any microservice does not comply with the target privacy policy. We focus on detecting the following types of violations, and as we show later, we structure our observed profile in order to do this.

- *Secondary purposes of use* [Com18; Fed22]. A PII type is collected and processed for a purpose that is disallowed in the target policy.
- *Extensive data sharing* [Int18b; Dil21; Fed23]. A PII type is shared with an external domain for a given purpose and the target policy forbids sharing the PII type for that purpose or with that domain.

The privacy engineer chooses a target privacy policy (Step 3). Prose then determines all violations of microservices' observed profiles, or the system-wide observed profile,

with this policy (Step 4; ??, ??). If there are no violations with the target privacy policy (Step 5), then all microservices can be directly deployed into production (Step 6).

If one or more microservices violate the target privacy policy (Step 5), the privacy engineer should be able to analyze these violations (Step 8, ??). They should be able to retrieve violations by type (secondary purpose of use, or extensive data sharing), as well as violations that involve a given microservice, purpose of use, set of PII types, or a given external domain. Privacy engineers are intended to collaborate with service developers to fix these violations.

However, the privacy engineer and service developers may be limited in that they may not be able to fix each violating microservice before it needs to be deployed to production. They may only be able to fix a fraction of all violations in a single iteration or deployment of the website. Then, they may want to prioritize fixing some microservices, over fixing others. They can either pick which microservices to fix in the current iteration (Step 9a). Alternately, they can decide the number of microservices, or the number of clauses of the target privacy policy, that can be fixed in this iteration (Step 9b).

We conceptualize two iterative fixing strategies that the privacy engineer can adopt. For example, they may focus on fixing the k worst microservices first; that is, they would fix all violations across k microservices that violate the most number of clauses of the target policy. We refer to this as the *greedy fix* strategy. Alternately, in a *systemic fix* strategy, they may focus on fixing k systemic violations that span many microservices. Prose should output which microservices to fix, under the strategy chosen by the privacy engineer (Step 8b; ??). Assuming that a given set of violations are fixed, the system can be seen as complying with an *effective privacy policy*, which is less restrictive than the target privacy policy. In general, the effective privacy policy will be at least as strict as the one given by the system-wide observed profile, and at most as strict as the target privacy policy. Prose also outputs the effective privacy policy (Step 10a; ??), assuming that the microservices chosen by the privacy engineer (in Step 9a) or output through either of the two aforementioned strategies (in Step 10b), are fixed. We observe that if *all* violations are fixed, then the effective privacy policy will be the same as the target privacy policy.

After the microservices are fixed and retested, we iterate over the workflow for the second time. The microservices' new observed profiles and the system-wide profile are derived in Step 2. If any microservice violates the *effective* privacy policy (Step 5), then the privacy engineer and software developers continue with the next iteration of the loop, in the testing environment. Otherwise, these microservices may be ready to be

deployed into the production environment (Step 6).

In production, the privacy engineer may want to *enforce* the *effective* privacy policy (Step 7; ??). That is, they may want to ensure that *only* data flows, in and out of a microservice, that are compliant with this policy, are allowed in production, and any violating traffic should be dropped. This serves as a defense-in-depth measure, since there may be data flows in the production environment that were not observed during deployments to the testing environment, and that violate this policy. (If, during the first iteration of the loop, the microservices are compliant with the target privacy policy (Step 5), then the effective privacy policy is the same as the target privacy policy.)

Privacy engineers may trigger incident response or alert management tools, which we outline in ??, in order to alert software developers of a privacy violation incident in production (Step 7). Since any data flows that violate the effective privacy policy will be dropped, the impact of our framework in production will be the same as a network failure to the back-end policy-violating microservice. Potentially, the user may not be able to experience the front-end functionality properly. Privacy engineers and software developers can then engage in the same process as they did in the testing environment, to fix these violations (Steps 8–11). If they are unable to fix all violating microservices in a limited time available to respond to the incident, then they may set a weaker effective privacy policy (than the one currently in place), to prevent violating data flows from being dropped. They may then fix any remaining microservices that cause the violations shortly thereafter, and then redeploy with the same effective privacy policy as before the incident.

We note that the privacy engineer can prototype multiple target privacy policies, by repeating steps 3–5 in ??, without needing to rerun Steps 1–2; that is, without retesting all services. They would do this until they determine that they can fix all violations within multiple deployments of the system to production. Thus, Prose should support privacy engineers in efficiently prototyping target privacy policies (?). (We discuss how we handle the privacy engineer not setting a target privacy policy in ??.)

We define two modes of operation, namely *observe* and *enforce* modes, which dictate whether the applicable privacy policy is enforced or not. When microservices are deployed into the production environment (Step 6), as we discussed earlier, Prose enforces the effective privacy policy, and to do so, it runs under the enforce mode. When microservices are deployed into the testing environment (Step 1), Prose *generally* runs in the observe mode.

One exception to this rule arises when privacy engineers want to simulate the production environment, when Prose is set to the enforce mode in the testing environment.

So, once no violations are detected (against the target or effective privacy policy), the microservices can optionally be retested with Prose set to the *enforce* mode in the testing environment. (In other words, in ??, after Step 5, but before Step 6, we can optionally repeat Step 1, under the enforce mode, and continue with Steps 2–5. Microservices can then be deployed into production in Step 6 only after no violations were detected in the *enforce* mode in Step 5.) Privacy engineers can then be assured that when the given versions of microservices are deployed into production, Prose will not drop any traffic (other than unexpected policy-violating data flows that were not seen by Prose in the testing environment, as discussed above).

We note that when the per-service and the system-wide observed profiles are formed for the first time (Step 2), the privacy engineer is not expected to have yet set the target privacy policy (Step 3). When running Prose on the tested microservices for the first time (Step 1), our system does require a privacy policy to be set. We can use a default privacy policy of “allow all”, or, “reject all” (Step 0). We highlight that the choice of the default does not matter, since the privacy engineer sets a target privacy policy later on. As long as Prose is run in the *observe* mode the first time that the microservices are tested (Step 1), each microservice’s observed profile will reflect PII types consumed or shared in all traffic that is triggered by the end-to-end test.

To summarize, our privacy policy workflow conceptualizes an observed profile for each unit in our architecture, namely a microservice, as well as a system-wide observed profile. We define and operationalize two types of privacy policies: a target privacy policy, that the system will ultimately comply with (at the end of multiple deployments) after fixing all violations, and an effective privacy policy, that the system complies with after fixing some, but not all violations. This workflow can be extended to other architectures, such as distributed data processing pipelines with jobs instead of microservices. While we specify the format of our observed profile and privacy policy, and structure them towards the violations that we are interested in detecting and preventing, this workflow is agnostic of the type of violations. Indeed, this workflow can be applied to detect and prevent violations relevant to the underlying architecture, such as data linking violations for distributed data processing pipelines. We contrast our privacy policy workflow with related work in ??.

5.3.2 Privacy engineering requirements

Based on our privacy policy workflow, we formulate functional requirements for our system; these requirements are catered towards privacy engineers. We annotate each requirement with the corresponding plane. Separating requirements by planes enables

modular design; a better design to address a given requirement need only address one of the planes, while reusing the design for the other plane components.

- PR1 *Data plane*: Automatically monitor data flows between microservices in the service mesh, while these microservices are tested.
- PR2 *Control plane*: Derive an *observed profile* for each microservice, using the data flows observed in the data plane (??).
- PR3 *Control plane*: Compose a set of observed profiles, which correspond to a set of microservices, to derive a *system-wide observed profile*.
- PR4 *Data plane*: Determine violations of *a given microservice* under a target privacy policy, while minimizing false negatives and false positives, and if necessary, trading off a higher false positive rate for a lower false negative rate.
- PR5 *Control plane*: Determine violations of *the system-wide observed profile* under the target privacy policy, subject to the same constraints as outlined in ?? on the false negative and positive rates.
- PR6 *Control plane*: Determine the violations of a given microservice or the entire system under a target privacy policy T_2 , given violations under policy T_1 . We wish to do this efficiently, without rerunning the data plane and retesting all services in light of ??.
- PR7 *Control plane*: Analyze violations based on the microservices involved, their purposes of use, set of PII types, or the external domain.
- PR8 *Control plane*: Given violations of the system-wide profile with the target privacy policy (??), compute the set of microservices to fix and the effective privacy policy, for the greedy and systemic iterative fix strategies.
- PR9 *Data plane*: Prevent any flow of user data that violates the effective privacy policy in production. (In light of requirement ??, the implementation to mediate data flows should be decoupled from individual microservices' source code.)

5.3.3 Privacy policy violations

We remind the reader of the secondary purpose-of-use and data sharing violations introduced in ??. We discuss each type of violation, through examples, and outline how it can be fixed.

We present a simplified example based on past FTC rulings [Fed22; Fed23], to illustrate these two types of violations. We focus on the following microservices within an online social network: USERS, AUTHENTICATION, ADVERTISING, and MARKETING microservices. We present communication between these microservices, and between these microservices and external domains, in ??.

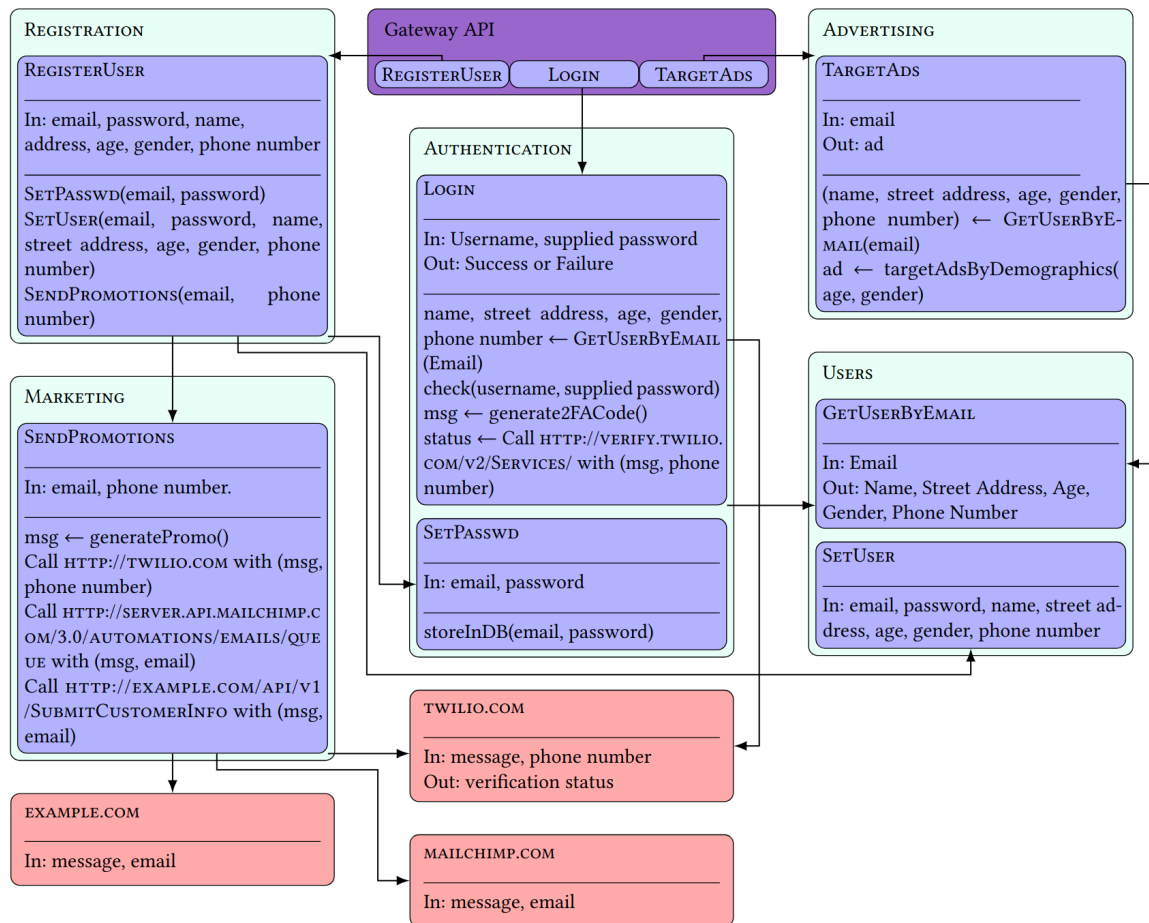


FIGURE 5.3: Depicts five services (e.g., **MARKETING**) and their endpoints (e.g., **SENDPROMOTIONS**). Each endpoint text includes inputs, and optionally, outputs as well as a pseudocode implementation. In this pseudocode, calls to internal functions in the same service are in regular font and start with a lower case letter (e.g., generatePromo()). Calls to other services' endpoints or to external domains are in SMALL CAPS font, and start with an upper case letter. Any external domain contacted by any endpoint is shown as: **TWILIO.COM**.

The privacy engineer sets the first version of a target privacy policy, which is represented informally in ?? below. Each clause of the policy has a purpose of use, the PII type, and a list of external domains that the PII type can be shared with. If no PII types are allowed to be shared with any external domains for the given purpose, as in ??, then this list is empty.

Listing 1: First version of a target privacy policy for ??.

- Advertising: Username, street address.
- Authentication: User’s email address, phone number and password.
- Marketing: User’s email address.

Secondary purpose-of-use violation. Within a microservice architecture, a secondary purpose-of-use violation can occur when a microservice consumes data items that are disallowed for its purpose of use. A microservice can either directly consume disallowed PII types through an endpoint that it offers, or within that endpoint’s implementation, it could call another microservice’s endpoint, and receive the disallowed PII types in their response. We thus distinguish between these two forms of secondary purpose-of-use violations.

Suppose that microservice A makes a request to one of microservice B’s APIs, and receives a response; here microservice A is the caller and microservice B is the callee. In a *direct* purpose-of-use violation, the callee consumes as an input a PII type that is disallowed for its purpose. Going back to the running example, under the given target privacy policy, the MARKETING microservice’s SENDPROMOTIONS endpoint is permitted to consume the user’s email, but not the phone number.

On the other hand, instead of considering the inputs to the callee, we can focus on its output to the caller. In an *indirect* purpose-of-use violation, the caller microservice cannot use the PII types returned by the callee microservice. For instance, the ADVERTISING microservice’s TARGETADS endpoint calls the USER microservice’s GETUSERBYUSERNAME endpoint. The latter endpoint responds with the user’s details, including the user’s phone number. However, the ADVERTISING microservice is not permitted to consume the user’s phone number under the given target policy. Similarly, the AUTHENTICATION microservice’s LOGIN endpoint, which also calls the GETUSERBYUSERNAME endpoint, also has an indirect purpose-of-use violation, since it is not allowed to consume the user’s street address under the target policy.

Data sharing violation. An endpoint can share a PII type with an external domain. The target privacy policy should explicitly allow that PII type to be shared with that domain

for the purpose of use associated with that endpoint’s microservice. So, sharing any PII type with any domain, when the target privacy policy does not allow it for that purpose of use, results in a data sharing violation. As an example, the SENDPROMOTIONS and LOGIN endpoints share data with different external domains, which is disallowed in the privacy policy in ?? (for all purposes). Similarly, sharing a PII type with a disallowed domain, or sharing a disallowed PII type with an allowed domain, both violate the privacy policy.

Fixing violations. While automatically fixing violations is outside of our scope (see ??), we describe how service developers can fix each violation type manually, and how the privacy engineer can revise the target privacy policy if they deem it to be too restrictive. For instance, out of the data sharing violations to EXAMPLE.COM, TWILIO.COM and MAILCHIMP.COM, the privacy engineer decides to allow the latter two, by adding extra clauses to the privacy policy as in ??. However, the SENDPROMOTIONS endpoint implementation should be fixed to stop sharing the users’ email addresses with the EXAMPLE.COM domain.

Listing 2: Second version of a target privacy policy, which extends the first version in ?? with the following clauses.

- Authentication: Share user’s phone number with TWILIO.COM.
- Marketing: Share user’s email address with MAILCHIMP.COM.

To fix a direct violation, developers of the callee microservice are responsible for removing the disallowed PII type as an input parameter, or replacing it with one that is allowed, and reimplementing the violating endpoint. For instance, in the second version of the MARKETING microservice, the developers may reimplement the SENDPROMOTIONS endpoint to only consume the email address as input (sending promotions only through email via MAILCHIMP.COM as usual), and consequently, to drop calls to the TWILIO.COM domain.

Both the caller and the callee may be responsible for fixing an indirect purpose-of-use violation. When multiple callers of an endpoint incur an indirect purpose of use violation, then the callee maybe overexposing PII types, and should refactor the endpoint into multiple granular endpoints. In our example, the developers of the USER microservice are responsible for breaking up the GETUSERBYUSERNAME to GETPHONENUMBERBYUSERNAME, which will be called by the LOGIN endpoint, and the GETADDRESSBYUSERNAME endpoint, which will be used by the TARGETADS endpoint. After these updates, the LOGIN endpoint will no longer violate the policy, but the TARGETADS endpoint will continue to violate the target policy. So, this particular indirect purpose-of-use violation needs additional fixes at the caller microservice (ADVERTISING). Its

developers need to reimplement their TARGETADS endpoint, and in particular, the `targetAdsByDemographics` internal function, to instead target ads by the zip code.¹

As an example of an effective privacy policy, suppose that the developers of the MARKETING microservice do not fix their PROMOTIONS endpoint (i.e., the call to `EXAMPLE.COM` remains) before it is to be deployed into production. Then the privacy engineer can temporarily allow this violation by adding a corresponding data sharing clause, for the marketing purpose, to `??`, and using the resulting policy as the effective privacy policy for production. Note that compliance with the effective privacy policy would still require fixing the ADVERTISING and USERS services.

Testing and detecting violations. We remind the reader of unit and integration tests described in `??`. We outline which types of tests are required, in the context of Step 1 of our privacy policy workflow in `??` to detect each type of violation. While direct purpose-of-use and data sharing violations can be triggered through unit tests, indirect purpose-of-use violations require end-to-end tests to evaluate how the output of each tested microservice's endpoints are used by other microservices. As such, whenever a microservice is redeployed, we need to rerun all end-to-end tests that depend upon at least one API endpoint of that microservice.

5.4 Threat model

We begin with outlining how Prose trusts service developers, privacy engineers and DevOps engineers. We then outline the Trusted Computing Base (TCB) for our system.

Privacy engineers. First, we expect privacy engineers to set a strict target privacy policy. If the privacy engineer does not set any target privacy policy in Step 3 in `??`, then it is set to one that only allows processing data as given by the system-wide profile. (We discuss how Prose can set a target privacy policy, using the system-wide observed profile, in `??`.) We highlight the fact that the system *always* complies with this privacy policy, and in fact, this is the most strict privacy policy that the system is known to always adhere to. Independently of whether and how the privacy engineer engages with the privacy policy workflow towards data minimization, we emphasize that at the end of Step 2, they will learn the system-wide observed profile and use it to formulate the privacy policy text, which is displayed to end users. In doing so, Prose realizes

¹Alternately, as discussed above, the privacy engineer may revise the target privacy policy to allow the use of age and gender for advertising purposes; however, for the following discussion, the clause for the ADVERTISING purpose of use remains as it was set initially in `??`.

Hoepman’s INFORM strategy. We also expect privacy engineers to set effective privacy policies, that are successively stricter, in order to ultimately comply with their chosen target privacy policy. If no target privacy policy was set, then since the effective privacy policy is at least as strict as the system-wide observed profile, our system would enforce the system-wide observed profile in Step 10.

Second, Prose does not automatically fix violations of the target privacy policy; privacy engineers are responsible for overseeing the fixes in co-ordination with service developers, as discussed in ???. Privacy engineers would be incentivized to do these tasks, in order to demonstrate compliance of their back-end code with data privacy regulations such as the GDPR. Broadly, organizations are obligated to follow such regulations in order to avoid large fines [STUD22], as well as to avoid losing privacy-conscious users [OFI21]. Third, privacy engineers will need to configure our PII identification unit, which is described in ??, in order to recognize PII types for their business domain, such as fine-grained health or financial data. Finally, we note that the clauses of our privacy policies do not take into account the users’ consent. The problem of storing users’ consent and selectively allowing data flows, based on the associated user’s consent, is outside of our scope. We discuss this problem further in ??.

Service developers. First, we expect service developers to write high-quality integration tests for their services. Since service developers are already required to thoroughly test their services in order to meet functional and performance requirements (??), it is reasonable for Prose, similar to security or quality assurance operations, to rely on and benefit from such tests (??). Second, we expect service developers to instrument their service for distributed tracing, which requires minimal effort as we detail in ?? and is independently useful for DevOps engineers, as outlined in ??. If this is not done, then Prose will not be able to derive accurate observed privacy profiles.

Third, we expect service developers maintaining each microservice to collaborate with privacy engineers to fix violations within the service. We argue that since service developers will benefit from Prose, they are incentivized to operate honestly. For example, Prose outputs an observed privacy profile for each service. Using this observed privacy profile, a new service developer joining a team can easily discover data flows within the implementation of each API endpoint.

Purpose-of-use labelling. Privacy engineers define the set of purposes that any service can be labelled with. We note that this task is relatively infrequent; the privacy engineer will only need to add a purpose of use for an added business functionality,

such as advertising and recommendations. (We outline how DevOps engineers can ensure that the CD pipeline labels each pod with the purpose of use of each microservice container that it will execute in ??.) Each microservice is labelled with one purpose of use. In labelling a microservice’s purpose of use, we emphasize that we do not need to determine whether it is essential for providing the business’ functionality or not. Each service only needs to be assigned a purpose from a set of purposes pre-defined by the privacy engineer or roles derived from the organization hierarchy.

A DevOps engineer or service developer can misrepresent the purpose of use of a service, such as by labelling it as “shipping” instead of “advertising”. Their goal would be to prevent Prose from detecting their service as violating the target privacy policy. They may claim a purpose of use that allows their service’s API endpoints to consume or share a superset of PII types that are disallowed for their actual purpose of use in the target privacy policy. We note that they would need to identify a fake purpose-of-use for each target privacy policy that the privacy engineer sets. We trust service developers and DevOps engineers to label the purpose of use of each service accurately, as is common in literature [KNB24; HBK24].

TCB. We trust the underlying orchestrator. Consequently, the orchestrator is trusted to handle threats to availability. In other words, machine failures will not impact the guarantees made by Prose, since the orchestrator will regenerate Prose components, such as pods, to ensure high availability. Our TCB also includes all of our design components, which are built on top of the orchestrator. We describe these components in the next section. Importantly, the microservices’ source code is not part of our trusted computing base; we do not need to assume that the microservices’ source code is itself free of vulnerabilities.

The soundness and completeness of Prose, with respect to requirement ??, depends on that of our design components. We summarize our assumptions on their soundness and completeness, as follows. We assume that our implementation of our policy-checking logic in our authorization agent is sound, in that it will not flag a violation when there is none, and complete, in that it will always detect a violation. As we discuss in ??, this logic only includes checking for inclusion of elements in finite sets, and is easily decidable. We also assume that our service proxy mediates all traffic; if our service proxy is misconfigured and does not mediate all traffic, this can lead to false negatives. We evaluate the soundness and completeness of our PII identification unit in ??.

5.5 Design components

We present our design in terms of two logical planes: the data and control planes. The data plane monitors and mediates inter-service communication, effectively serving as a distributed system of runtime monitors for the microservices. The control plane manages the data plane and, in turn, receives information from it.

Our data plane design consists of four main components: a service proxy, a PII identification unit, an authorization agent and distributed tracing. First, a service proxy is attached to each microservice container in the data plane. The service proxy monitors and mediates data flows, that is, API calls, to and from that microservice (Section ??). Second, a PII identification tool is used to determine whether and which PII types are present in a given API call (Section ??). Third, authorization agents are used to detect and prevent violations of the target privacy policy (Section ??). Fourth, distributed tracing is used to propagate information between the data and control planes of our frameworks (Section ??). The service proxy is responsible for calling a PII identification unit and an authorization agent, and is configured to support distributed tracing.

Prose’s control plane plays two roles; it configures the data plane, such as by supplying the target or effective privacy policy. The control plane also processes the traces output from Prose’s data plane, and uses them through custom components to meet its own requirements such as an observed profile updater, privacy operators, and an analysis engine, as we discuss in ??.

5.5.1 Service proxies

Different back-end microservices may need to perform a common set of peripheral tasks, relating to observability (distributed tracing), traffic management (load balancing), and security (authorization). While each microservice can be modified individually to perform these tasks, this approach couples the source code for the tasks with the business logic implemented in the microservice, as well as with the microservice source code language. The *sidecar architectural pattern* is used to minimize these two forms of coupling. Here, a sidecar container that performs these common tasks runs alongside a microservice container [Mic22b]. Service proxies (SPs) such as Envoy [Env23] can be used to implement the sidecar architectural pattern. They can monitor and mediate L3–L7 layer traffic to and from a microservice and perform the aforementioned tasks. Service meshes, such as Istio [Ist23a], configure and attach a service proxy to each pod. They do so by integrating with an orchestration platform such as Kubernetes, through custom controllers as shown in Figure ??. Thereby, they form a control plane over these

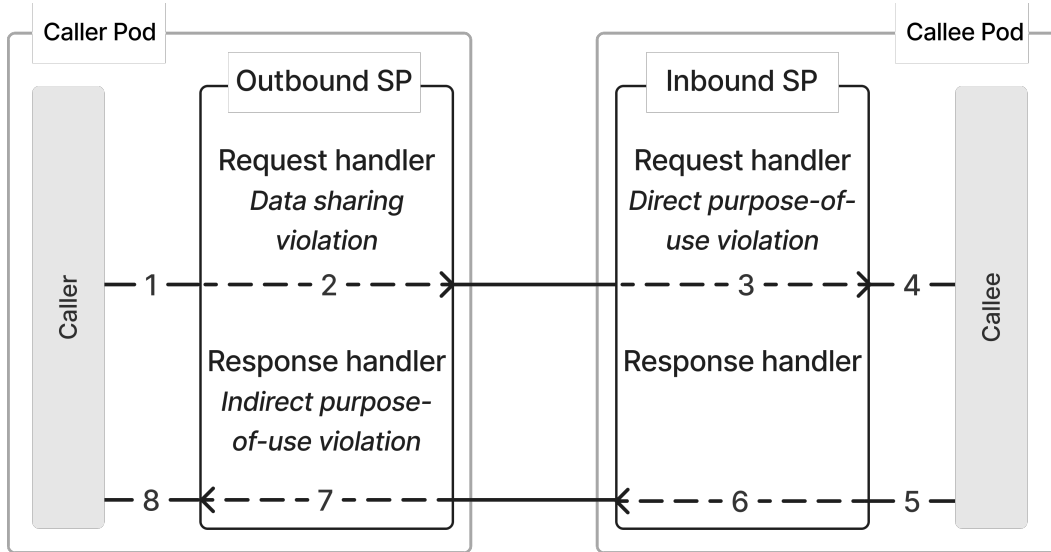


FIGURE 5.4: In an API call between a caller and a callee microservice, this figure depicts the chronological order in which request and response handlers on the attached service proxies intervene. The caller microservice’s outbound SP is triggered, and the callee microservice’s inbound SP is triggered. As discussed in ??, to detect a data sharing violation, we instrument the request handler on the outbound SP (step 2). To detect a direct purpose-of-use violation, we instrument the request handler on the inbound SP (step 3). To detect an indirect purpose-of-use violation, we instrument the response handler on the outbound SP (step 7).

proxies [Kle17].

Service proxies fit our requirement for Prose to be decoupled from the microservice source code (?). When Prose is run in *observe* mode, service proxies are configured to *monitor* microservices’ API calls and requests to external domains, identify any PII types in these data flows, and detect any violations with the target or effective privacy policy. Whereas, when Prose is run in the *enforce* mode, service proxies are configured to *mediate* the aforementioned communication, by dropping policy-violating data flows.

Inbound and outbound traffic mediation. We remind the reader that all containers in the same pod share a networking interface. The service mesh controller (Istiod in ??) automatically injects an *init* container into each pod. As its name suggests, the *init* container is run before the application container and modifies IP tables rules to redirect inbound and outbound traffic to ports where the service proxy is listening [Son24; Chu25]. Instead of microservices communicating directly with each other, their associ-

ated SPs receive traffic inbound to, and outbound from, the microservice, and support mediating this traffic. In inbound mode, the SP mediates requests *to* the microservice and the associated responses, through handlers. In outbound mode, the SP mediates requests *from* the microservice and their responses. (We conceptualize each mode of the service proxy as a separate service proxy; in practice, both modes are bundled into one executable.) Consider an API call from a caller to a callee microservice. Figure ?? depicts the chronological order in which request and response handlers on each microservice’s service proxy intervene. The caller microservice’s *outbound* SP will be triggered, on both the request and the response, whereas for the callee microservice, its *inbound* SP will be triggered. Specifically, when the caller microservice makes an API call (step 1), its *outbound* SP is triggered on the request (step 2). Consecutively, the request handler of the *inbound* SP of the callee microservice is also triggered on this request (step 3). Once the callee microservice receives the request (step 4) and returns with a response (step 5), the response handler of its inbound SP processes the response (step 6). Similarly, the outbound SP of the caller microservice is also triggered on the response (step 7), before passing it to the caller microservice (step 8).

We use a combination of request and response handlers on the inbound and outbound proxies, in order to detect purpose-of-use and data sharing violations, as we explain in ?. To detect a data sharing violation, we instrument the request handler on the outbound SP (step 2). To detect a direct purpose-of-use violation, we instrument the request handler on the inbound SP (step 3). To detect an indirect purpose-of-use violation, we instrument the response handler on the outbound SP (step 7).

5.5.2 PII identification

A central challenge in analyzing information flows for privacy violations is automatically identifying data items in these flows as personally identifiable information (PII). Structured PII types such as credit cards or email addresses can be detected through regular expressions (regexes). Sen et al. [SGD+14] use regexes to annotate datasets with PII types from their privacy policy language taxonomy, while minimizing manual annotation to increase confidence in the inferred PII types. In general, custom PII types can be detected using Natural Language Processing (NLP) techniques. Existing NLP libraries such as SpaCy [Exp25] tokenize and lemmatize the input text, and perform Named Entity Recognition (NER). NER classifies entities named within the text, including proper and common nouns, into an entity class, such as a name or location. NER uses a machine learning model that is trained to detect a set of entity classes, and outputs a confidence interval along with the detected entity. PII identification libraries, such as Presidio [Mic23b], offer both regex and NLP recognizers; the latter recognizers

use NER to detect various PII types. (Presidio was used by Statistics Canada to sanitize PII in survey forms, in order to comply with Canadian privacy regulations [Sap24].)

5.5.3 Authorization agents

Authorization agents, such as the Open Policy Agent (OPA) [Ope22a], Falco [Fal25], and Kyverno [Kyy24] are used to offload the task of authorizing requests to or from a microservice [Ope22b]. Each microservice need not include any authorization source code or policy. Instead, an authorization agent is instantiated with an authorization policy. For instance, OPA supports policies written in a declarative query language, Rego [Ope23a]. The microservice can then query the authorization agent for each request, and it returns an authorization decision based on the authorization policy. To further separate concerns, instead of modifying microservices to integrate calls to authorization agents, a service proxy can make these authorization calls. Although authorization agents are typically expected to authorize *incoming requests* to a service, we use them to prevent the *service* from consuming or sharing user data in violation of the target privacy policy. Authorization agents enable detecting violations with a target privacy policy in a testing environment, towards *??*. They also prevent data flows that violate the target policy in production, satisfying *??*.

5.5.4 Distributed tracing

Distributed tracing (DT) is an observability tool that allows DevOps engineers to monitor the flow of requests through different microservices and consequently, identify misbehaving microservices [PSMI20]. Popular distributed tracing implementations include OpenTelemetry [Ope22c], Jaeger [Jae23a] and Zipkin [Ope23d]. Source code needs to be configured or *instrumented* for distributed tracing [Ope23b]. It can be instrumented at different granularities. For instance, each API endpoint or function implementation can be instrumented. A *span* represents a unit of work. As an example, a span for an API endpoint corresponds to that endpoint processing a request. It includes start and end timestamps that indicate the duration, in wall clock time, taken by the endpoint implementation. This implementation can set the span *status* to indicate errors and add tags to the span.

A *trace* consists of spans for each instrumented API endpoint that the request passes through. In other words, a trace represents the cumulative work done across multiple microservices to process an end user's request. Each span and trace are indexed by unique identifiers (IDs). Each trace forms a tree of spans [Ope23c], similar to flame graphs used to profile performance of a monolith application [Gre20]. In other words,

spans for all endpoints that the request goes through are arranged into a tree, which is simply the trace. Suppose a given API endpoint implementation calls an endpoint of another microservice. Their spans will form a parent-child relationship on this tree. The caller endpoint's span will be the parent, whereas the callee endpoint's span will be the child. This parent-child relationship is propagated by the caller to the callee through the *span context*. The span context consists of the trace ID and the parent span ID, which is the caller endpoint's span ID. It is sent as a part of the request, for instance, through request headers. Whenever an instrumented callee receives a request, it checks the headers for these identifiers. If they exist, then it uses them to set its own span context. If they do not exist, then it initializes a new trace, by generating a trace ID. In turn, when that endpoint makes a call, it propagates the span context, with its own span ID as the parent span ID. Spans can be annotated with span tags; we use span tags to convey information between the data and the control planes.

Spans from multiple deployed microservices are collected by a trace collector microservice, which outputs service maps. Nodes in a service map represent microservices or third parties, and edges represent communication between microservices or between a microservice and a third party. Many DT implementations, such as Jaeger, include built-in tools to collect, visualize and query traces. Using these querying units, we filter traces by the span names and values of span tags within our control plane.

Integration with service proxies. Service proxies can reduce the amount of distributed tracing instrumentation that service developers need to do. Importantly, a service proxy can start a span for each incoming request to an API endpoint, and end the span when the implementation returns a response. It can also batch spans and send them periodically to the trace collector microservice.

We observe that the service implementation is responsible for propagating the span context. This is because if a service endpoint's implementation made an outgoing request, then as a black box, the service proxy will not be able to reason that the outgoing request was caused by that incoming request. In practice, the span context is propagated through a DT library for the language that the microservice is written in. The library usually instruments the relevant application layer protocol to propagate the span context, such as through headers for HTTP.

5.6 Data plane design

The Prose data plane meets the following requirements: it automatically mediates data flows between microservices in order to detect different types of violations (??), detects whether each microservice violates a policy (??) and prevents these violations in the enforce mode (??). The data plane mediates each microservice by using the SP, PII identification unit, and distributed tracing in tandem. The SP mediates the relevant traffic for each violation type (Section ??). The PII identification unit detects PII types within this traffic (Section ??). The authorization agent detects and prevents target policy violations (Section ??). Distributed tracing is used to propagate the detected PII and violation types to the control plane, such that the control plane can form the observed profile of the microservice (??).

5.6.1 Mediating data flows for each violation

The SP can be attached to mediate inbound or outbound traffic, as discussed in ?? and depicted in ?. We remind the reader of the three types of privacy violations that were exemplified in ?. Our two types of secondary purpose-of-use violations, namely direct and indirect purpose-of-use violations, require mediating different points in the request flows. For a direct purpose-of-use violation, the request needs to be mediated either after it is sent by the caller microservice (Step 2 in ??) or before it is received by the callee microservice (Step 3). To detect this violation type, we need the purpose of use of the *callee* microservice, which is unknown to the service proxy of the *caller* microservice. So, we detect this violation type using an inbound request handler at the callee microservice (Step 3). Similarly, for an indirect purpose-of-use violation, we could either mediate the response when it is sent by the callee microservice (Step 6) or when it is received by the caller microservice (Step 7). Since we need the purpose of use of the *caller* microservice, we detect this violation at that microservice, using an outbound response handler. In summary, we detect purpose of use violations by intervening at the microservice that *consumes* the PII in violation of the target policy.

Finally, we detect data sharing violations by intervening at the microservice that *shares* the PII in violation of the target policy. Since the caller microservice will contact an external domain that is outside the service mesh, this request will not pass through an inbound SP. So, we must use the request handler at the outbound SP (step 2), and mediate requests to external domains. We do not mediate outgoing requests to *other microservices in the service mesh* in the outbound request handler, since direct purpose-of-use violations are detected at the callee microservice, as described above.

To summarize, we map the violations to the service proxy handlers as follows:

- Direct purpose-of-use violation: Inbound request handler
- Indirect purpose-of-use violation: Outbound response handler
- Data-sharing violation: Outbound request handler

5.6.2 Detecting PII types

We remind the reader of PII identification libraries and in particular, Named Entity Recognition (NER), both of which were discussed in ???. Prose requires a PII identification library to detect PII exchanged by services. In particular, we detect PII sent within JSON objects in HTTP requests and responses through regex and NER-based scanners for predefined PII types. In ??, we describe how NER-based scanners can be configured for this task. The SP propagates the set of PII types in a given request or response through a span tag to the control plane, as we outline in ??.

5.6.3 Detecting and preventing violations

Considering our privacy policy workflow (??), when Prose is run for the first time (Step 1), the privacy engineer may not yet have set the target privacy policy. In this case, the default privacy policy, such as a fully permissible policy, is distributed to the authorization agents in the data plane, and consequently, no violations occur. Once the privacy engineer sets a target privacy policy (Step 3), the control plane first validates it using an authorization agent, and then distributes the policy to the authorization agents in the data plane. The authorization agent in the control plane validates the target privacy policy as follows. First, the authorization agent ensures that the purposes of use within the target policy fall within a set of purposes defined by the privacy engineer. Second, it checks that the PII types in the policy are a subset of the set of PII types that could be output by the PII identification unit. If an effective privacy policy is set (Step 9), the control plane uses it over the target privacy policy.

In the data plane, the agent uses the given privacy policy to authorize data flows that are observed by the service proxy. It is always invoked with the following inputs, which are necessary to detect violations: the purpose of use of the data flow, and a set of PII types within it. It is optionally supplied with an external domain, in order to detect data sharing violations. The agent first validates the purpose of use and set of PII types for the data flow. First, the agent ensures that the input purpose of use is allowed by the privacy policy. Second, the agent checks that the PII types are a subset of the set of PII types allowed by the target policy for this purpose of use. If the authorization agent

encodedata

Service: **golang-filter** | Duration: **90.49ms** | Start Time: **40.38ms**

▼ Tags

DATA_FLOW	ENCODE_DATA
internal.span.format	otlp
prose_opa_decision	deny
prose_opa_enforce	false
prose_pii_types	US_DRIVER_LICENSE,PHONE_NUMBER,US_BANK_NUMBER,PERSON
prose_sidecar_direction	Outbound
prose_violation_type	PURPOSE_OF_USE_INDIRECT
SIDECAR_DIRECTION	Outbound

> Process:

> Warnings (1)

SpanID: 6b42ed21b43e2965

FIGURE 5.5: Example of selected tags within a Prose span generated by our service proxy, which is attached to one of the microservices of the Bookinfo application used in our evaluation.

is invoked from an outgoing request handler, then the agent also receives the external domain to which the PII types would be sent. In this case, the agent also checks that the privacy policy allows each PII type to be shared with that external domain for the given purpose of use.² The agent responds with an allow or deny decision, and a string outlining the violated clauses, in case of a denial.

Importantly, the SP enforces the authorization agent’s decision by dropping violating requests or responses, *only* when Prose is in the enforce mode. In other words, the SP and thus the data plane does not drop any traffic in the observe mode. In both observe and enforce modes, the SP propagates the decision from the data plane to the control plane through a span tag, as we detail below.

5.6.4 Communicating from data to control plane

The service proxy uses distributed tracing to send relevant information about the data flow from the data plane to the control plane. It does three main tasks in the data plane towards this goal. As in a usual application of distributed tracing, the SP inserts a new span describing its operations, which we refer to as a Prose span. It adds tags to this span and finally, exports the span to the control plane.

First, a Prose span is created for each handler (request or response) in each traffic direction (inbound or outbound), which are shown in Figure ???. The SP names each span appropriately, in order to distinguish it from other spans inserted by the services as well as to enable the control plane to retrieve all spans inserted by Prose. Second, the SP adds the following information as tags to each Prose span; we depict some of the tags within a Prose span in Figure ??.

- Cryptographic hash of the target policy used by the authorization agent,
- Cryptographic hash of the service image,
- Traffic direction (inbound or outbound),
- Handler type (request or response),
- Purpose of use of the service,
- Set of detected PII types,
- External domain in case of outbound request handlers (for data sharing violations),
- Callee service and endpoint in case of outbound response handlers (for indirect purpose-of-use violations), and
- Authorization agent decision, including the reason in case of a violation.

Using the exported span and its tags, the control plane can form observed profiles of services. We highlight that all of the information that is used by the authorization agent to make a decision on a data flow, namely the purpose of use, detected PII types, external domain (if any) and a hash of the privacy policy, is always sent to the control plane. This property enables the control plane to exactly determine the authorization agent's decision, for a new privacy policy, without retesting the service in the data plane, in light of ???. Distributed tracing functions identically in both observe and enforce modes, in that all aforementioned tags are included in a span.

²The external domain is not applicable for our direct or indirect purpose-of-use violations, and thus, this check is not performed when the agent is invoked from the inbound request handler or outbound response handler, which may simply pass an empty list of external domains. We trust the service proxy to properly mediate all traffic as outlined in ??, and in this context, we trust the outbound request handler to always pass the external domain to the authorization agent.

5.7 Control plane design

The control plane meets the following requirements. In ??, we discuss how the control plane automatically processes traces to form the observed profile of each microservice, and updates this profile efficiently with changes in the microservice (?). The control plane also composes per-service observed profiles to form a system-wide observed profile (?), determines whether and how the system, as a whole, violates a target privacy policy (?), and given how a service violates one policy, efficiently determines its violations of a different policy (?). In ??, we provide two operators to handle these requirements. Finally, privacy engineers can analyze violations along different criteria (?), and prioritize which services to fix, using our iterative fix strategies (?) and our analysis engine, as we describe in ?.

5.7.1 Forming observed profiles

We form per-service observed profiles by processing the traces received from the data plane. We first outline the observed profile structure. We then describe how we process each trace to enqueue a change to the observed profile associated with each span in the trace. We also discuss how we update the respective observed profile with the enqueued changes. We conclude with consistency criteria that the per-service observed profiles follow.

Observed profile structure. The observed profile is structured as a nested key-value store. As an example, in ??, we show the observed profile of the AUTHENTICATION service from ?. We remind the reader of the first version of the privacy policy in ?. The observed profile contains a hash of the target privacy policy (Line 1) as well as a hash of the service image (Line 2). At the top level, it includes the purpose of use of the service (Line 3) and a list of API endpoints that the service exposes (Lines 4–36). It maps each API endpoint (Lines 5, 30) to a list of endpoint profile objects, which are indexed by their hashes (Lines 6, 31) as we explain below.

We draw the reader’s attention to the endpoint profile key (Line 9), which describes the PII types that a given endpoint consumes (directly or indirectly) or shares when it is called. We can see that the LOGIN endpoint directly consumes the email address (Lines 10–11), which complies with the target privacy policy, as per ?. Similarly, the SET-PASSWD endpoint also directly consumes the email address (Lines 35–36). (Note that while this endpoint also consumes the password, the PII identification unit by default does not detect passwords, and so the password is not detected as a PII type and is thus

```

1 {"target_policy_hash": "0xd6ed55b44a1458c5",
2  "service_hash": "0x3843a2d07f79794a", // image hash prefix
3  "purpose_of_use": "authentication",
4  "endpoints": {
5    "Login": {
6      "d2f536a59847bc52": { // enclosed JSON hash
7        "traceID": "0x3e8a025cb0c1a97e",
8        "spanID_of_call": "0xa46ce6bf916d6191",
9        "endpoint_profile": {
10         "direct": {
11           "pii_compliant": ["EMAIL_ADDRESS"]},
12         "outgoing": [
13           { "type": "indirect",
14             "spanID": "0x00000000004789bb",
15             "callee_host": "users.default.
16               svc.cluster.local",
17             "callee_path": "/GetUserByUsername/",
18             "pii_compliant": ["EMAIL_ADDRESS",
19               "PHONE_NUMBER"],
20             "pii_violation": ["PERSON", "LOCATION",
21               "DATE", "GENDER"],
22             "violation_reason": "Detected PII type
23               PERSON, LOCATION, DATE, GENDER cannot be
24               processed for purpose of use
25               AUTHENTICATION" },
26           { "type": "shared",
27             "spanID": "0x0000000000621def",
28             "pii_violation": ["PHONE_NUMBER"],
29             "external_domain": "twilio.com" } ] ] } },
30   "SetPasswd": {
31     "0xff9ad63344df255c": {
32       "traceID": "0x00000000831bbcce",
33       "spanID_of_call": "0x0000000093aaddee",
34       "endpoint_profile": {
35         "direct": {
36           "pii_compliant": ["EMAIL_ADDRESS"] } } } } } }

```

Listing 3: Observed profile for the AUTHENTICATION service.

not propagated through spans.) The LOGIN endpoint also consumes PII types indirectly (Lines 13–25), through the response to its call to the GETUSERBYEMAIL endpoint of the USER service (Lines 15–17). While consuming the email address and phone number is compliant with the privacy policy (Lines 18–19), consuming additional PII types, such as the location, violates the privacy policy, since it does not explicitly allow processing them for the purpose of authentication. Hence, the authorization agent correctly reports a violation (Lines 20–21) with the appropriate reason (Lines 22–25). We can also see that the LOGIN endpoint shares the phone number with an external domain, namely TWILIO.COM (Lines 26–29). Since the target privacy policy (??) forbids sharing any PII types, the authorization agent reports a violation.

An endpoint may be called multiple times in responding to an end user’s request. Potentially, the endpoint may behave differently in different calls to it. It may consume or share different PII types in calls, and consequently, it may satisfy the target policy in one call and violate it in another call. It can also, in turn, make a different number or ordering of calls to other microservices or external hosts, or call different external hosts. The endpoint profile captures each call during which an endpoint behaves uniquely, as far as processing and sharing PII types is concerned, in comparison to previous calls. We use a cryptographic hash of the endpoint profile, *excluding the span ID field*, to capture differences in its behavior across different calls to it.³ As evidence of a given data flow for service developers and privacy engineers, we include the trace ID of the end-to-end request (Line 7, 32), as well as the span ID of the call to the endpoint (Line 8, 33), within our endpoint profile. The service developer can use the trace ID to disambiguate calls to the same endpoint. They can use the span ID of the call to disambiguate multiple calls to the same endpoint within the same trace.

Processing spans. With this structure of the observed profile in mind, we can reason about how traces sent from the data plane (??) can be used to form per-service observed profiles in the control plane. We first retrieve all spans that are inserted by the service proxy in the data plane, by querying for Prose spans. We then extract the values of span tags, and use them to fill in an endpoint profile object. There are two relevant questions here: First, how should this endpoint profile object be filled, using the span tags? Second, since a span corresponds to a call from a caller microservice to a callee microservice (??), should we modify the *caller’s* or the *callee’s* observed profile with this endpoint profile object?

³If the privacy engineer finds that multiple unnecessary profiles are created for an endpoint, then they can exclude certain key-value pairs in the computation of this hash, thereby effectively merging these profiles into one.

We answer these questions based on whether the span was produced by an inbound request handler, an outbound response handler or an outbound request handler (??, ??). First, since an *inbound request* span describes a request to a *callee* endpoint, the observed profile of the callee endpoint's microservice should be modified. Specifically, since this span describes PII types that the endpoint received in a request, we construct a new endpoint profile object, with the value of the `"direct"` key filled with the compliant and/or violating PII types (Lines 10–11 in ??). Second, since an *outbound response* span describes a response received by a *caller* endpoint, we modify the observed profile of the caller endpoint's microservice. Since this span details PII types received in a response (from the callee microservice), our new endpoint profile object for the caller will include information from this span alongside the `"type": "indirect"` key–value pair (Lines 13–25). Third, we remind the reader that an *outbound request* handler span is only generated when a microservice calls an external domain, rather than when a microservice calls another microservice in the service mesh. So, given an outbound request handler span, the observed profile of the *caller* microservice is modified. In particular, the new endpoint profile object will include an object with the `"type": "shared"` key–value pair (Lines 26–29).

Updating the observed profile. We now know which microservice's observed profile should be updated and the endpoint profile object that it should be updated with. Whenever the profile for one call to an endpoint differs from another call to it, in at least one of the aforementioned ways, the hash of the new endpoint profile would differ from the hash of the existing profile. When this hash differs, we append that endpoint profile, along with its trace ID and span ID, under the new hash (Line 6). If this hash is identical, meaning that the microservice behaves identically in both calls, then we prefer to retain the more recent trace as evidence. Specifically, we update the trace ID and the span ID (Lines 7–8), with the newer counterparts if the start timestamp of the newer trace is later than that of the recorded trace.

Consistency criteria. We outline criteria that govern the consistency of the observed profile of a microservice. We observe that in the data plane, each handler (??) in a given service proxy evaluates each data flow, to or from an endpoint, under the same purpose of use of the monitored microservice and against the same privacy policy. So, we expect that a sound authorization agent will *not* mark a PII type as being in compliance with the policy in one data flow, while marking it as a violation when in another data flow, in the following scenarios:

- CC1 when an endpoint receives two requests, each containing that PII type.
- CC2 when an endpoint receives responses to two requests made to other microservices' endpoint(s), and each response contains that PII type. The requests may have been sent to the same or different callee endpoints; the values for the `"callee_host"` and `"callee_path"` fields are not used by the authorization agent and are thus irrelevant for the current discussion. The requests may have been made while handling the same or different initial request to that endpoint, and so these PII types may be recorded within the same or different endpoint profiles.
- CC3 when an endpoint sends a PII type in two requests to the *same* external host. We note that the privacy policy may allow a given PII type to be shared with one external host, but not with a different external host; so, this consistency criterion only applies when the `"external_host"` key has the same value.

Each of the above criteria also hold for two data flows that involve different endpoints of the same microservice, as well as for data flows to endpoints across microservices that have the same purpose of use. The control plane checks that these consistency criteria hold while updating observed profiles with span tags from a trace. If they do not hold, then the associated authorization agents may be misconfigured or misbehaving and thus the privacy engineer is informed. Our composition operator, which we discuss below, relies on these criteria to hold, in order to output a consistent system-wide observed profile. For example, an inconsistent profile may convey that for a given purpose one PII type is processed both in compliance with, and in violation of, the target privacy policy. If these criteria do not hold, then as we discuss below, the composition operator can be redefined to nonetheless output a consistent system-wide observed profile.

5.7.2 Privacy operators

We describe two operators below: a composition operator to combine multiple microservices' observed profiles into a single system-wide observed profile (??, ??) and a comparison operator to compare two privacy policies (??).

Composition operator. In terms of our privacy policy workflow (??), this composition operator (Step 2) may be run after the effective privacy policy is set, that is, while repeating Steps 1 and 2 during the second iteration of the workflow. This operator may also be run when the privacy engineer repeats Steps 2–5 while setting and prototyping multiple target privacy policies (in Step 3). Thus, the term privacy policy in the text below can refer to either the initial, target, or effective privacy policy.

The system-wide observed profile has the same format as the privacy policy, in that it maps each purpose of use to all PII types consumed for that purpose, whether directly or indirectly, as well as any PII types shared with any external domains. If either the target or effective privacy policy have been set by the time this composition operator is run, then the system-wide observed profile also marks these data flows as in compliance with, or in violation of, this policy.

We define how the `"pii_compliant"`, `"pii_violation"` as well as the `"external_host"` fields should be composed, beginning at the fine-grained level of endpoint profiles, up to the coarse-grained level of microservices. By defining how the `"pii_violation"` field is composed, we ensure that the system-wide profile aggregates violations of the target or effective privacy policy, which may be present in the per-service observed profiles, in light of ???. We rely on the consistency criteria outlined in ???. We first summarize each endpoint profile and we then compose two endpoint profiles for the same endpoint. We then compose the resulting endpoint profiles of different endpoints, to obtain a summarized observed profile for each microservice. Finally, we compose summarized observed profiles across microservices to arrive at a system-wide observed profile.

In summarizing each endpoint profile, we note that the privacy policy format (??) does not distinguish between an endpoint directly or indirectly consuming a PII type. Thus, we first compute the union of the respective values for the `"pii_compliant"` field, across direct and indirect processing keys, within an endpoint profile. We similarly compute the union across the `"pii_violation"` fields, across direct and indirect processing keys. Due to the consistency criteria ??? and ??? outlined in ??, we expect that this union will result in two disjoint sets for the `"pii_compliant"` and `"pii_violation"` fields.⁴ Based on the consistency criterion ??, if two profiles for one endpoint describe data sharing with the same external host, then we can compose them into one entry, by simply computing the union of their PII types. Otherwise, if they involve different external hosts, then we store a list of tuples of the set of PII types and the external host with which they are shared.

In our second step, we compose direct and indirect processing and data sharing, across two endpoint profiles, just as we composed them within an endpoint profile. In our third step, we compose the resulting (single) summarized profile for each endpoint, across endpoints. We can do this since the privacy policy format does not distinguish

⁴If the consistency criteria do *not* hold, then the union will not be disjoint. If a PII type is in violation once, then we can conservatively consider it to be in violation, and drop it from the `"pii_compliant"` field. This also holds when these fields refer to PII types with the same external host.

between endpoints, and the consistency criteria hold even if the two data flows are to different endpoints in the same microservice. In our fourth step, we compose microservices' summarized observed profiles based on their purposes of use. If two microservices have different purposes of use then we simply concatenate their summarized observed profiles. Otherwise, we compose them just as we handled different endpoints in the same microservice.

Comparison operator. The privacy engineer may want to efficiently prototype various target policies (??). A naive approach is to rerun all tests against all microservices, that is, rerun the data plane in the observe mode under each target policy, in order to obtain a new system-wide profile with violations for that policy. However, for a large architecture of thousands of microservices, this approach would be inefficient.

To satisfy requirement ??, once the privacy engineer runs the data plane once in the observe mode under a policy T_1 , Prose must determine whether this set of tested microservices would violate a policy T_2 , while avoiding rerunning the data plane against T_2 as much as possible. Our observation is that just as the authorization agent was used to evaluate that a data flow complies with a policy T_1 in the data plane, it can similarly be used to evaluate an observed profile, or another target policy T_2 , against T_1 in the control plane. In the data plane, we check whether the PII types and external domains in the data flow are allowed by T_1 for the purpose of the flow, whereas when comparing policy T_2 to T_1 , instead of iterating over all data flows, we would iterate over *all purposes* listed in T_2 .

This comparison only outputs whether T_2 is at most as strict as T_1 . For T_2 to be less strict than T_1 , T_2 should be at most as strict as T_1 and T_1 should *not* be at most as strict as T_2 . However, when the respective sets of purposes, PII types and external domains of two policies are disjoint, then T_2 would not be at most as strict as T_1 and T_1 would not be at most as strict as T_2 .⁵ As an example of two such policies, consider a T_2 that is identical to T_1 except that while T_1 allows the phone number to be sent to TWILIO.COM for the purpose of authentication, T_2 allows it to be sent to GOOGLE.COM. In contrast, Listing ?? and Listing ?? serve as an example of comparable and non-equal policies; the policy in Listing ?? is stricter than that in Listing ??.

When one of the policies is stricter than the other, if a microservice satisfies the stricter policy, then it will continue to satisfy the looser policy. Conversely, if a microservice violates the looser policy, then it will continue to violate the stricter policy. Consequently, in both cases, we do not need to reevaluate its observed profile against

⁵This occurs since the sets of purposes of use, PII types and external domains are not totally ordered.

the new target policy. However, if a microservice satisfies the looser policy, then we cannot infer whether it will satisfy the stricter policy. Similarly, if a microservice violates the stricter policy, then we cannot infer whether it will violate the looser policy. Moreover, when neither policy is stricter than the other, as we saw in the aforementioned example, we need to evaluate *each* microservice’s observed profile against the new target policy T_2 . Importantly, we still do not need to rerun and retest all of the services in the entire data plane against T_2 . We use an authorization agent in the control plane to compare each service’s observed profile against the target privacy policy, to reevaluate violations. This approach can also be used to determine whether the entire system would comply with a new target policy, by using the system-wide observed profile instead of a service’s observed profile.

5.7.3 Analysis Engine

The size of the system-wide observed profile is proportional to the number of microservices, and specifically, the number of microservices with unique purposes of use. In practice, for architectures with thousands of microservices, the analysis engine may store per-service observed profiles using a DBMS. The system-wide observed profile can be dynamically computed through a stored procedure. To assist the privacy engineer in processing the system-wide observed profile, the analysis engine provides queries that analyzes this profile for the top n most common PII types, purposes of use and external hosts. The engine also assists the privacy engineer in categorizing violations of the target or effective privacy policy (??). To this end, it includes the following queries outlined in ?? . The query results are tagged with a hash consisting of the target policy version and the service versions. While we do not go into the implementation of the DBMS schema and SQL query definitions, we highlight that we can store aggregates that help Prose answer these queries during a single pass over all per-service observed profiles to form the system-wide observed profile. As an example, we discuss below how Prose stores aggregates to meet ?? regarding iterative fixing strategies.

Iterative fixing strategies. Fixing each microservice that violates the target privacy policy would typically require a privacy engineer to coordinate with the team of developers maintaining that microservice. Given that microservices may violate different sets of clauses of the target privacy policy, addressing multiple clauses across different teams in a single iteration may be challenging for the privacy engineer.

Prose can help the privacy engineer prioritize which microservices to fix, through two iterative fixing strategies, which are based on bounding either the number of teams

TABLE 5.1: Queries used to analyze violations of the system-wide profile with a given privacy policy.

Query name	Query
Violations by type	All services with a given type of violation (direct or indirect purpose of use or data sharing)
PII type	All services with a violation for a PII type
Purpose of use	All services with a direct or indirect purpose of use violation for a given purpose of use
Data sharing	All services with a data sharing violation for a given external domain
Violations by clause	All services that violate a given clause of the target policy
Top k violated clauses	k clauses of the target policy that are violated by the maximum number of services, and the number of services that violate each clause
Top k violating services	k services that violate the maximum number of clauses of the target policy

that they coordinate with, or the number of clauses of the target policy that they address, in each iteration. Each strategy produces two outputs. First, it outputs the violating microservices that should be fixed in the current iteration. Second, it also outputs the effective privacy policy that would be followed by the entire system once these violations are fixed.

A privacy engineer following the greedy fix strategy addresses all violated clauses across k microservices. A relevant question is: which k microservices should be fixed such that the new system-wide observed profile would comply with the maximum number of clauses of the target privacy policy? The *top k violating services* query from ?? answers this question. This strategy enables the privacy engineer to bound the number of teams that they coordinate with, in a single iteration, to k . In theory, this strategy could output an effective privacy policy that is identical to the system-wide profile. This happens when the clauses addressed by fixing the top- k violating microservices continue to be violated by other microservices that have fewer violations. For example, microservice 1 violates clauses a and b , whereas microservice 2 violates clause b and microservice 3 violates clause a . Fixing the top violating microservice, namely microservice 1, will result in an effective privacy policy equal to the system-wide observed profile.

A privacy engineer following the systemic fix strategy seeks to address clauses of the target policy that are a systemic problem, since they are violated by multiple microservices. So, they should focus on a clause that is violated by the maximum number of microservices; the *top k violated clauses* query from ?? outputs k such clauses. (We note that k here is upper bounded by the number of clauses in the privacy policy.) This strategy then fetches the microservices that violate each clause by running the *violations by clause* query on that clause.⁶ The privacy engineer may simultaneously work with the teams maintaining each of these microservices on a common fix. They would repeat this process for all violated clauses in a single iteration. In contrast with the greedy fix strategy, the systemic fix strategy will *always* output an effective privacy policy that is *stricter* than the system-wide observed profile.

We reiterate that each clause of a privacy policy maps a purpose of use to a set of PII types that can be processed for that purpose, or shared with an external domain, as shown in ?. We observe that the authorization agent is configured to output the clause of the policy that was violated, in the `"violation_reason"` field in the per-service observed profile (Line 22 in ?).

We compute the set of microservices to fix, when following each strategy, as follows. While aggregating observed profiles to form the system-wide observed profile, we maintain a list of the clauses violated by each microservice. Specifically, whenever a `"pii_violation"` field is present within an endpoint profile, we fetch the violated clause, using the `"violation_reason"` field. For the greedy fix strategy, we then sort microservices by the number of clauses violated, and pick the top- k microservices. For the systemic fix strategy, we invert this mapping, to determine the set of microservices violating each clause. We then sort the clauses by the number of microservices that violate it, and pick the top- k clauses.

We compute the effective privacy policy, when following each strategy, as follows. We remind the reader that the effective privacy policy is at least as strict as the one given by the system-wide observed profile, and at most as strict as the target privacy policy (?). We note that the system-wide observed profile will list violations with the target privacy policy, as discussed in ?. If no microservices are fixed, then the effective privacy policy is given by treating the system-wide observed profile as an allowlist, and adding a clause for each violation, effectively marking it as compliant behaviour. If one microservice is fixed, then we compute the effective privacy policy as follows: we

⁶Given that each clause of the Prose policy refers to one purpose, each clause may be violated by multiple microservices only if multiple microservices have the same purpose. If each microservice has a unique purpose, then the systemic fix strategy would output k clauses, in a predefined order, and their associated violating microservices.

TABLE 5.2: Components of the Prose data plane that we developed, along with the programming language that they are written in, and their number of Lines of Code (LoC). Third-party components that we use through libraries or executables, including Kubernetes, Istio, Envoy, Presidio, OPA and Jaeger, are not included in this table.

Component	Implementation	Language	LoC
Service proxy (SP)	Filter loaded by Envoy SP	Golang	1893
PII Identification unit	Server running Microsoft Presidio	Python	451
Authorization agent	Data flow validation logic and tests	Rego	135
Deployment configuration	Configuration to deploy Prose on top of an Istio service mesh	YAML Dockerfile	385 74
Evaluation configuration	Helm charts [Hel24] to deploy the test application and all variants	YAML	2696

drop all of its violations from its observed profile, compute the system-wide observed profile, and then add clauses for any *remaining* violations, which would correspond to non-fixed microservices. For the greedy fix strategy, we drop all violations from the selected k microservices' observed profiles, and follow the aforementioned approach to compute the effective privacy policy. For the systemic fix strategy, we follow the same approach; however, instead of dropping all violations from a microservice's observed profile, we drop only those violations that involve one of the top- k clauses.

The privacy engineer can further customize each strategy, to address only a given violation type, or only those violations that involve a given set of PII types or purposes of use or external domains. They can use the first four queries in Table ?? to this end. Importantly, the privacy engineer can interleave these two strategies. For example, the privacy engineer may choose to adopt the greedy fix strategy for early iterations of the target privacy policy, and then switch to the systemic fix strategy.

5.8 Evaluation

We outline our implementation of the data plane of Prose. We then discuss our experiment for the efficiency of the data plane of Prose, followed by our experiment for its effectiveness.

5.8.1 Implementation

We use Kubernetes [Kub23] as our orchestrator, and Istio [Ist23a] as our service mesh. Istio manages and deploys Envoy service proxies (SPs) [Env23] as sidecar containers. We implement the Prose data plane in the form of a Golang filter that is executed within the Envoy SP sidecar container [Env25]. Our control plane consists of a rudimentary trace collector service. In order to mediate data flows to detect all violation types, our Golang filter includes the three handlers as outlined in ?? and depicted in ??, namely an inbound request handler, an outbound request handler and an outbound response handler. Each handler first detects PII types in the data flow, checks for violations, sends a span to the trace collector service, and in enforce mode, drops violating requests or responses. In Table ??, we outline the components of the Prose data plane, along with the programming language that they are written in, and their number of Lines of Code (LoC).

Detecting PII. We use Presidio [Mic23b] as our PII detection library. We detect the default set of supported PII types, such as names, credit card numbers, locations, dates and so on [Mic22c]. We apply named entity recognition (NER) to detect PII sent within JSON objects in HTTP requests and responses. The library could analyze either the keys or the values of the JSON object, or both. We chose to recognize PII types within the *values* of the JSON object, rather than the *keys*. Consequently, non-descriptive keys, such as a random string instead of “username”, cannot be used to bypass the PII detection logic. However, a key named “username” is more likely to contain a value with the PII type “name”, than if it was “location” or a random string. Thus, the key is used as context to increase the confidence interval for NER.

Presidio is written in Python and it does not have Golang integrations, so it is executed as a separate service, which is then called by the Golang filter on each request or response. We deploy Presidio as a separate pod. We scale up replicas of this pod within our experiment setup.

Detecting violations. We use Open Policy Agent [Ope22a] to detect violations. We provide a small and thoroughly tested implementation of our violation detection logic (??) in Rego (??). We observe that the privacy engineer does not need to write the privacy policy itself within Rego. They can simply use a well-known structured document format such as JSON to specify the policy, by listing permissible purposes of use, PII types and external domains. In this way, the privacy engineer can prototype different target privacy policies without familiarity with Rego. Our Golang filter gathers the detected PII types, the purpose of use of the container, and optionally the external domain in case of an outbound request. Our filter then uses the OPA Golang SDK [Ope24] to

evaluate our violation detection logic against the target privacy policy with these inputs. For our experiments, we set a target privacy policy that allows all existing data flows in the application. This is not identical to a fully permissive policy; OPA still needs to evaluate each data flow against this policy.

Distributed Tracing. We use Zipkin for distributed tracing. Our choice of Zipkin is dictated by the set of providers compatible with the Envoy SP [Aut22]. We integrate the Zipkin Golang library within our Golang filter, and we use Jaeger [Jae23b] for our trace collector service.

5.8.2 Evaluating efficiency

Experiment setup. We deploy a microservices-based application, Bookinfo [Ist22], in multiple variants as we describe below. We stress-test the application at various request rates for a fixed time interval. We measure the end-to-end latency experienced by clients, and thus, this experiment evaluates whether Prose can meet ??.

We consider the following baselines or variants: orchestrator alone (Kubernetes, abbreviated as K8S), orchestrator with service mesh (K8S + ISTIO) and orchestrator with service mesh with Prose (K8S + ISTIO + PROSE). We also have a K8S + ISTIO + PROSE - PRESIDIO variant, which does not call Presidio on the mediated requests and responses but instead uses hard-coded PII types. The K8S + ISTIO + PROSE (PRESIDIO WITHOUT CACHE) variant does not include a cache within Presidio, as we detail later. For all but the K8S variant, each application pod has a service proxy as a sidecar.

We run our experiment on an AMD EPYC 7313P 16-Core Processor with 1 TB RAM and a 200 GB NVMe SSD. We create three virtual nodes on this server, and Minikube [Kub25] deploys a Kubernetes cluster across these three virtual nodes. We use Minikube backed by Docker, meaning that each node is represented by a Docker container. (Each Docker container runs Kubelet, which was described in ??, and all pods that are handled by Kubelet run as nested containers.) We execute $N = 10$ runs of each variant for each tested request rate r . Each run proceeds independently of other runs, as follows. We configure 10 replicas of each pod of the application and also scale the Presidio deployment to have 10 replicas. We first wait to ensure that all pods are ready to receive traffic. As recommended for stress testing [Gra24], before running the stress test, we first simulate an average load for $t = 10$ s, that is set to the minimum of half the request rate of the test and a constant threshold; thus, the request rate for the average load is: $\min(r/2, 100)$. For the stress test itself, we then send r req/s for t seconds, with a timeout of 30 s for the response. We use Vegeta [Sen24] to send both the average and stress test loads. (By default, Vegeta paces the requests constantly across

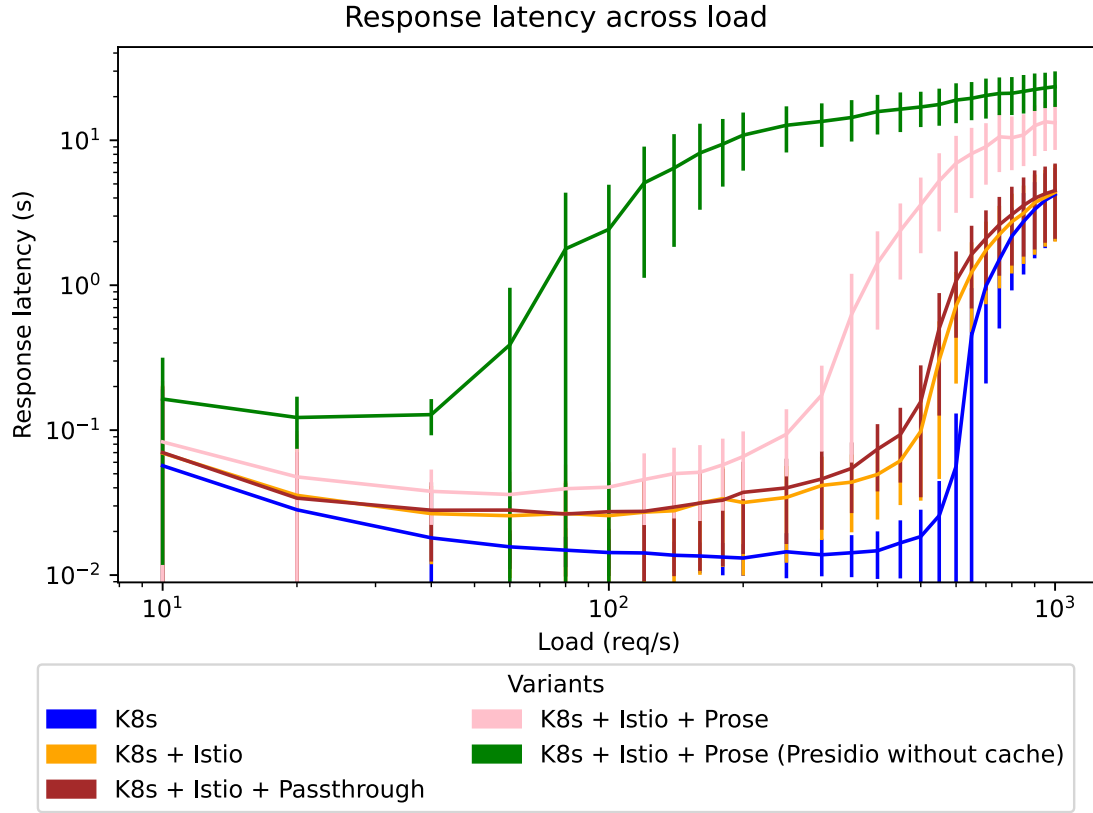


FIGURE 5.6: Average and standard deviation of the response latency for each variant, averaged across all $100 \cdot r$ responses for each request rate r on the x-axis. Note the log scale on the y-axis.

the test duration, at the granularity of nanoseconds.)

We compute the average and standard deviation of the end-to-end latency across all responses ($N \cdot r \cdot t$) for all N runs for each request rate r , and we plot them in ???. From that figure, we observe that for each variant, the response latency increases slowly with the request rate, since the underlying system is undersaturated, and as the system begins to saturate, the latency then shows a steep increase over medium to high request rates. We note that our client times out requests at 30 s, meaning that if the client does not receive a response by 30 s, then the client deems that request as having failed. Consequently, the end-to-end response latency is capped around 30 s for high request rates, and we thus obtain a convex plot. We are interested in the behaviour of our variants while the underlying system remains undersaturated. We hypothesize the following relations between the end-to-end latency over our variants:

- H1 In increasing order of latency, we expect: K8S, K8S + ISTIO, K8S + ISTIO + PROSE - PRESIDIO, K8S + ISTIO + PROSE and K8S + ISTIO + PROSE (PRESIDIO WITHOUT CACHE).
- H2 To meet ??, our proposed variant for Prose, namely K8S + ISTIO + PROSE, should incur a reasonable overhead over the K8S + ISTIO variant for low to moderate request rates, while retaining latencies of less than one second.

We also observe from ?? that both hypotheses hold true. Each successive variant in the list in Hypothesis ?? adds more functionality that can contribute to the end-to-end latency, as follows. The overhead of the K8S + ISTIO variant over the K8S variant is due to the service proxies themselves. We note that the same number of service proxies are in path to answering each request, and thus the overhead introduced by the service proxies in the K8S + ISTIO variant is the sum of the overheads introduced due to each service proxy in path of the request. The overhead incurred by the K8S + ISTIO + PROSE - PRESIDIO variant over the K8S + ISTIO variant is due to the authorization agent and distributed tracing logic, which is executed for each request that passes through each service proxy. Adding the PII identification unit, we observe that K8S + ISTIO + PROSE (PRESIDIO WITHOUT CACHE) incurs a large overhead over K8S + ISTIO variant. We find that the cause of this overhead is the time taken by Presidio to detect PII in the requests and/or responses. This is substantiated by the fact that the K8S + ISTIO + PROSE - PRESIDIO variant, which replaces all calls to Presidio with hard-coded PII types, incurs a much smaller overhead over the K8S + ISTIO variant, as compared to the overhead of the K8S + ISTIO + PROSE (PRESIDIO WITHOUT CACHE) variant. We could increase the number of replicas of Presidio in order to load each replica less. However, if we were to do this, then the K8S + ISTIO + PROSE (PRESIDIO WITHOUT CACHE) variant would maintain a smaller overhead over K8S + ISTIO until a higher load rate, thereby simply delaying the problem. We observe that deploying Presidio as a sidecar, instead of as separate pods, would only shave off the network latency from the sidecar service proxy to the service, which amounts to about 1–2 ms per service in path of the request, and a total of about 6 ms; we are CPU-bound, not network-bound.

To mitigate this problem, we include a naïve cache within Presidio in order to run NER less frequently for each client-side request. In our naïve cache, each JSON object, which could be within an HTTP request or response, serves as the key, and the PII types detected by Presidio serve as the value. Our cache prevents repeatedly rerunning NER for the same JSON object processed by each service proxy handler; our K8S + ISTIO + PROSE variant includes this cache. We note that Hypothesis ?? holds true for this variant, since its end-to-end latencies are higher than that for the K8S + ISTIO + PROSE - PRESIDIO variant, but lower than that of the K8S + ISTIO + PROSE (PRESIDIO

WITHOUT CACHE) variant. We also note that the K8S + ISTIO + PROSE variant adds at most an overhead by a factor of $\cong 0.2/0.02 = 10$ over the K8S + ISTIO variant up to 300 requests/second, while retaining latencies of 0.2 s. While arguably this is a large overhead in light of Hypothesis ??, we leave further optimization of the Prose data plane to future work.

While in our evaluated application, the JSON objects do not differ across end users' requests, we acknowledge that in general, this may not be the case. Specifically, the values within the JSON object body may differ across users, whereas in order to conform to the API of the endpoint, the keys (such as "name" or "email") are likely to be identical. Since Presidio executes NER on the *values* rather than the *keys*, the output of Presidio may differ for different users; consequently, our naïve cache would not suffice and Presidio would rerun NER on each user's data. Instead of setting the entire JSON object as the cache key, the cache can derive a key using the JSON keys within the JSON object. Consequently, the cache would be reused whenever the JSON objects have similar keys; that is, as long as HTTP requests (or responses) to (or from) the same API endpoint for different users' data are shaped similarly. Nevertheless, our evaluation demonstrates that with a reusable cache, Prose incurs a reasonable end-to-end latency overhead on top of the orchestrator and service mesh, in light of ??.

5.8.3 Evaluating effectiveness

We define the effectiveness of the Prose data plane in terms of its soundness and completeness in detecting privacy violations, in light of ??. Under our threat model, the soundness and completeness of Prose reduces to that of our PII identification unit. Our PII identification unit (Presidio) is not sound, in that it can mark a data item as PII when it is not (resulting in false positives), and not complete, in that it can miss marking some PII types (leading to false negatives). We begin with discussing the impact of false positives and negatives of Presidio on Prose. We then describe Presidio classifiers, outline our experiment setup and then present our hypotheses as well as our results.

Impact of false positives and negatives. A false negative contributes to the lack of completeness of Prose. Specifically, when in observe mode, a false negative means that the observed profile of a microservice misses a PII type being processed or shared. If it were detected, this PII type could have led to a privacy violation *only if* the given target or effective privacy policy disallows that particular PII type to be processed or shared for the purpose of use of the associated microservice. Thus, when Prose is run in enforce mode in production, false negatives could, but do not always, imply a privacy violation

that is missed. A missed privacy violation means that users’ PII has been handled by the back end while violating the privacy policy, and as a system designed to ensure that this does not happen, Prose would fail in its goal.

On the other hand, a false positive reported by Presidio means that a text token that contained no PII has been marked as such. Again, depending on the target or effective privacy policy, the PII type in a false positive could, but does not always, lead to Prose erroneously reporting a privacy violation. An erroneous privacy violation report impacts the microservice in enforce mode in production, as the associated data flow would then be dropped by the authorization agent. Thus an erroneous privacy violation may temporarily cause a lack of availability of the relevant API until the privacy engineer fixes this issue, as follows. Considering our privacy policy workflow (??), beginning with Step 10 in production, the privacy engineer would then go through Steps 5–7; in Step 7, they may set an effective privacy policy to allow the falsely flagged data flow until Prose or its PII identification unit can be modified to stop wrongly flagging the text in that data flow as a PII type. As with any classification task, when detecting PII, we navigate a tradeoff when deciding whether to reduce false positives or false negatives. In light of ??, this experiment examines how Presidio can navigate this tradeoff.

Presidio classifiers. We evaluate classifiers that are built in to Presidio for a total of 12 PII types. The classifiers use a combination of mechanisms used for NER, including pattern matching, checksums and custom logic [Mic24], as we summarize below and present in Table ?? . Classifiers can either be context-aware or not; context-aware classifiers use the context of the neighbouring words as well as any additional context supplied to Presidio. We divide the classifiers for these 12 PII types into four categories. First, we consider classifiers that use the context, pattern matching and a checksum; the International Bank Account Number or IBAN classifier is the only classifier in this category. Second, we then group classifiers that use only the context and pattern matching; the DATE_TIME, EMAIL_ADDRESS, IP_ADDRESS, URL, US_DRIVER_LICENSE and US_SSN are grouped into this category. Third, we consider classifiers that use custom logic; that is, the PERSON classifier searches for a person’s name, the NRP classifier searches for a person’s nationality, religious or political group, the LOCATION classifier searches for a geographic or politically defined location and the PHONE_NUMBER classifier searches for phone numbers. Finally, we have only a single classifier that does not use the context, namely the CREDIT_CARD classifier; this classifier uses only pattern matching and the checksum.

Experiment setup. We measure the effectiveness of Presidio, and to do so, we generate a ground truth of JSON key-value pairs as follows. Our sentence generator starts with a set of 210 template texts consisting of one or more sentences; out of these, 15 template

TABLE 5.3: Frequency of PII types in the ground truth and components of the classifier for each PII type.

PII type	Frequency	Classifier components			
		Context	Pattern Matching	Checksum	Custom logic
Credit card	166	○	●	●	○
Date / time	276	●	●	○	○
Email address	58	●	●	○	○
IBAN code	18	●	●	●	○
IP address	25	●	●	○	○
Location	4476	●	○	○	●
NRP	90	●	○	○	●
Person	1910	●	○	○	●
Phone number	518	●	●	○	●
URL	37	●	●	○	○
US Driver License	4	●	●	○	○
US SSN	105	●	●	○	○

texts have no PII types, whereas the rest have one or more PII types. It then substitutes a PII type in each string, with a generated value, such as an address or a credit card number. The generated sentences serve as the value within the JSON key-value pairs; we generate a total of 2000 pairs. We show the frequency of each PII type in the JSON value in Table ???. As described in ???, the JSON key serves as the context for increasing the confidence interval with which any word, in the value, is marked with a given PII type. We set the JSON key to one of the PII types present in the value.

In our experiment, we vary the threshold confidence score that is required to mark a word as a PII type, from a minimum of 0.3 to 0.9 at constant intervals of 0.05; the default confidence score for the classifiers is set to 0.4. We measure the false positives and false negatives produced by Presidio, across both datasets as input. The Receiver Operating Characteristic (ROC) of a classifier measures the true positive rate (TPR) against the false positive rate (FPR); a binary classifier that classifies an input word as a given PII type with any fixed probability, which we refer to as a random classifier, has an ROC of $y = x$. While varying the threshold confidence score, we plot the ROC for the classifier for each evaluated PII type in ???. The top left corner of the axes represents an ideal

classifier. A true positive here requires the classifier to mark one or more words as the correct PII type, at the correct start and end character indices in the text; detecting either the wrong PII type, or mismatching indices, lead to a false negative.

We hypothesize the following:

- H8 For a given classifier, we hypothesize that *increasing* the threshold should result in fewer true positives and/or fewer false positives.
- H9 We expect the classifier for each PII type to outperform the random classifier.
- H10 In light of ??, there exists at least one threshold value, such that all classifiers provide a reasonably high TPR, potentially at the cost of a moderate FPR. Satisfying this hypothesis would enable privacy engineers to set a threshold out-of-box, without the need to examine and tweak individual classifiers.

Through ??, we confirm that Hypothesis ?? holds, since within the ROC line plot for each classifier, with reference to a coordinate for a given threshold, the coordinate for the immediately larger threshold is below and/or to its left. Hypothesis ?? holds since all of our ROC line plots appear above the ROC for the random classifier.

We now discuss the performance of our classifiers with respect to Hypothesis ??, when considering a TPR value of at least 0.75 and an FPR value of at most 0.02 to meet this hypothesis. First, we note that all classifiers have a FPR of less than 0.02 for all thresholds, and thus Hypothesis ?? is not violated *due to high FPRs* for any classifier. Second, we consider classifiers that satisfy Hypothesis ?? for all thresholds, namely the IBAN, EMAIL_ADDRESS, IP_ADDRESS and URL classifiers. For all thresholds, these classifiers have a very high TPR and a very low FPR. For example, the lowest TPR shown by the URL classifier is 0.91, which appears for thresholds greater than 0.6; this TPR is still reasonably high and satisfies Hypothesis ??.

Third, we consider classifiers that satisfy Hypothesis ?? for a range of thresholds. The remaining classifiers that use only the context and pattern matching, namely the DATE_TIME, US_SSN and US_DRIVER_LICENSE classifiers, have a large enough TPR to satisfy Hypothesis ?? for a range of thresholds, rather than for *all* thresholds as was the case for the aforementioned classifiers. The US_SSN classifier has a perfect TPR for all thresholds below 0.9, whereas the US_DRIVER_LICENSE and DATE_TIME classifiers have a perfect TPR for thresholds of 0.3 and 0.4 only. For larger thresholds ($\{0.45, 0.5, 0.55, 0.6\}$), the US_DRIVER_LICENSE classifier has a TPR of 0.75 whereas the DATE_TIME classifier has a TPR of 0.6; thus, these three classifiers satisfy Hypothesis ?? for threshold values of 0.3 and 0.4. Importantly, for the largest threshold value, both the US_SSN and US_DRIVER_LICENSE classifiers show a TPR of zero; this is undesirable since *all* text tokens containing these PII types were missed and marked as false nega-

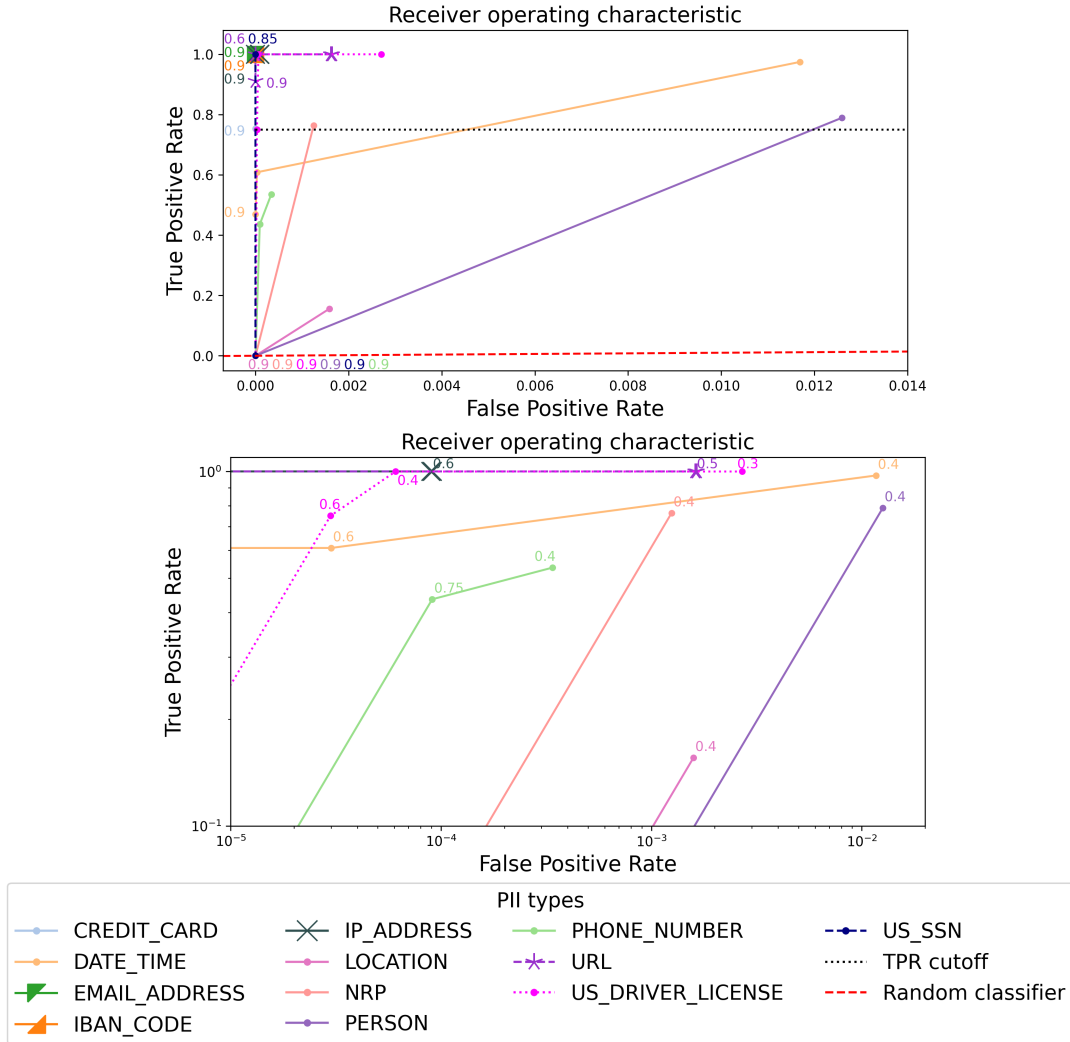


FIGURE 5.7: ROC of Presidio for classifying a word as a given PII type, while varying the threshold for the confidence score required for a successful classification. Note that the x-axis on the upper plot only goes up to 0.014 since all observed false positive rates are upper bounded by this. The lower plot has a log scale on both axes, and zooms in on the upper plot. For each PII type, we only plot each unique pair of true positive and false positive rates, across the thresholds in $\{0.3, 0.35, \dots, 0.9\}$; points on the ROC line plot for a given classifier are annotated in the same color as the line plot for that classifier, and are annotated with the largest threshold value for that pair of true and false positive rates. We show the TPR cutoff for Hypothesis ?? by a dotted line.

tives. This threshold value is thus not useful at all for these classifiers, when applied for PII detection. In contrast, the DATE_TIME classifier shows a non-zero, albeit small TPR value for thresholds greater than 0.6.

Considering classifiers in the third category, each of the LOCATION, NRP and PERSON classifiers have a non-zero TPR value, that is, 0.16, 0.77 and 0.79 respectively, for thresholds of 0.3 and 0.4, and a zero TPR value for higher thresholds. Similarly, the PHONE_NUMBER classifier has a TPR of only 0.54 for thresholds of 0.3 and 0.4, a TPR of 0.44 for thresholds in the range $\{0.45, 0.5, \dots, 0.75\}$, and again, a zero TPR value for higher thresholds. Thus, in light of Hypothesis ??, while the NRP and the PERSON classifiers may arguably satisfy the TPR cutoff of 0.75 for low thresholds, the PHONE_NUMBER and LOCATION classifiers have too low of a TPR for even these low thresholds. Finally, the CREDIT_CARD classifier, which is the only classifier in the final category, has an identical TPR and FPR value for all thresholds; the TPR value is 0.75 and thus this classifier satisfies the Hypothesis ?? for all thresholds.

In conclusion, Presidio classifiers perform differently at a given threshold, and there is no single value of the threshold to satisfy Hypothesis ?. Privacy engineers may expect that in order to have a non-zero TPR for all classifiers, they may be left with no choice but to set a low threshold value. However, we observe that Presidio allows configuring specific classifiers with a higher default confidence score value. Thus, for example, custom-logic based classifiers may be configured with a higher default confidence value than the pattern-based ones, in order to achieve a reasonably high TPR, at a reasonably high threshold, such as 0.8, for all PII types.

In general, privacy engineers may select different values of the threshold to navigate the tradeoff between false negatives and positives differently in the observe mode versus in the enforce mode. For example, privacy engineers may be willing to withstand a higher false positive rate in non-production environments, and so they may set a lower threshold in observe mode. They may wish to minimize the number of false positives reported in production environments, as software engineers may be required to be on call to respond to these incidents. So, they may set a high threshold in the enforce mode, thereby reducing the false positive rate at the cost of also reducing the true positive rate. We remind the reader that our privacy policy workflow handles privacy engineers' need to simulate the production environment in the testing environment, by retesting the microservices, in the testing environment, under the enforce mode. Thus, if privacy engineers were to choose different thresholds for the observe and enforce modes, they would still, nevertheless, be able to evaluate the microservices under both thresholds before deploying to production. Finally, we leave the evaluation of different PII identification units against PII within traffic encountered by real microservices

TABLE 5.4: Comparison of related work for Prose. Considering the architectural requirements outlined in ?? and the privacy engineering requirements outlined in ??, while excluding the efficiency requirement (??) in order to focus on the design, we compare each system against all requirements that are supported by at least one comparator system. We distinguish systems that provide ?? statically (S) versus dynamically (D). For ??, we compare support for purpose-of-use violations (P) and sharing violations (S). The comparator systems do not provide ??, ?? and ??.

System	??	??	??	??	??	??	??(P)	??(S)	??	??
Riverbed [WKM19]	○	○	●	○	○	○	○	●	○	●
Autoarmor [LCL+21]	●	○	●	● _S	○	N/A	○	○	○	●
Alastor [DPI+22]	●	●	●	● _D	○	●	○	●	○	N/A
DataProVe [TE22]	○	●	●	○	○	○	●	●	●	N/A
PrivGuard [WKN+22]	○	○	○	● _S	○	○	○	○	○	●
Model-driven privacy [KNB24]	○	●	●	○	●	○	●	●	●	N/A
Prose	●	●	●	● _D	●	●	●	●	●	●

application to future work.

5.9 Related work

We summarize recent literature on language-based approaches to data minimization, and contrast it with our privacy policy workflow. We then describe policy enforcement systems, including access control and auditing frameworks, for microservice-based and other recent architectures. We compare each work against the applicable requirements, out of those outlined in ?????. We present related work for Prose in Table ?? in chronological order.

5.9.1 Formal language-based approaches to data minimization

Many machine-readable privacy policy languages have been proposed, towards different goals. Kieran’s Fides is a declarative privacy language [Kie22; Eth23] that can be used to detect whether a declared privacy policy of a service violates a target privacy policy; both of these policies are set by the service developer. Fides trusts developers to correctly annotate services with privacy declarations, as well as update them whenever the source code changes. Sen et al. [SGD+14] present a privacy policy language, namely Legalese, that organizes policy field values into a lattice structure, placing privacy-sensitive purposes of use, such as advertising, to the top of the lattice. They also present lattices for policy fields, such as the data type and the purpose of use.

Another line of work involves formally describing the architecture, or a model of the system, and comparing it to the privacy policy that the system is intended to adhere to. Thong Ta et al. present a privacy policy language and an architecture description language (ADL) in DataProVe [TE22]. The requirements that the architecture should satisfy, such as obtaining consent from the user to process a data item, are encoded in a target privacy policy. Privacy engineers manually specify the entire system architecture in the ADL. Thong Ta et al. present conformance relations to automatically compare the privacy policy with the architecture specification, through an automated reasoning tool, and ultimately identify privacy policy violations (??, ?? holds). They evaluate DataProVe to identify privacy policy violations in a Covid Tracing App, with a client and a server entity.

Contrasting our privacy policy workflow with DataProVe, we find that each service must be specified manually in the ADL. In other words, the observed privacy profile for each service cannot be derived automatically, and so ?? does not hold. Moreover, in the absence of an ADL composition operator, DataProVe cannot compute the entire system’s architecture specification from the specifications of each service, and thus ?? does not hold. Therefore, scaling DataProVe to a distributed back-end architecture system with hundreds of microservices as entities would be infeasible, and thus ?? does not hold. (Requirement ?? is trivially satisfied since each service, irrespective of the language that it will be written in, needs to be described in the ADL. Requirement ?? is also trivially satisfied, since DataProVe does not consider the source code of the system at all.)

In model-driven development, software engineers model web applications using UML diagrams, and develop a data model, which specifies the data items, and an object model, which specifies how the application acts on those data items. Krstic et al. [KNB24] propose a model-driven development methodology towards data mini-

mization, by having privacy engineers and software engineers extend this process with a privacy model. Their system analyzes the information flows across the data and object models, and assesses their compliance with the privacy model. Based on these models, they automatically generate an enforcement mechanism that checks that the set of actual purposes is a subset of the set of declared purposes in the privacy policy, as well as that the user’s consent is provided for the given data type and purpose of use. If the business logic deviates from the model, then the enforcement mechanism detects this violation and raises an exception. Krstic et al.’s work improves over DataProVe in that they output a privacy policy that the application adheres to (assuming users’ consent is provided for all actions) akin to our observed profile (??). A drawback of model-driven development methodologies is that they implicitly expect software engineers to update the model with changes in the source code. In particular, changes in the source code should be reflected in updated data and object models, whereas changes in the privacy policy, such as allowing a previously prohibited data flow, are expressed by updating the privacy policy model. We emphasize here that Prose does not require software engineers to separately maintain a data or an object model.

Information flow control languages such as Jacqueline [YHA+16] ensure that confidential data inputs to programs are only output to appropriate channels or users. Programs typically need to be modified to handle and propagate confidentiality labels. Lehmann et al.’s Storm web framework [LKB+21] reduces the amount of annotations that developers insert in order to maintain confidentiality labels, using type theory. Hubert et al.’s WebTTC [HBK24] extends Jacqueline and Storm, by supporting user-defined and time-dependent policies.

As such, a common drawback of current work in language-based enforcement systems is that they operate within a single application. For instance, Storm governs access control policies within a single microservice that follows the model-view-controller engineering pattern. Extending these systems to a distributed microservices architecture (in light of ??) would require accounting for inter-service communication. A second challenge in extending these language-based enforcement systems to a polyglot architecture (in light of ??) is that they require modifying the compiler or interpreter of each language that the microservices are implemented in. For example, Krstic et al.’s enforcement mechanism depends on the language of implementation of the web application; they instrumented the Python interpreter to automatically generate enforcement checks for a Python Flask web application. Similarly, integrating Storm into services written in multiple languages would require extending their Haskell type checker to each language. In general, we find that existing literature in this area does not consider an iterative privacy policy workflow, and does not support privacy engineers in complying

with a target privacy policy over multiple iterations through fixing strategies.

5.9.2 Policy enforcement systems

Riverbed [WKM19] focuses on the compliance of web applications, written in Python, to user-defined policies, using trusted hardware platforms. These policies encapsulate storing a data item or sharing it with a third party; as such, they are less expressive than WebTTC’s user-defined policies. Riverbed first instruments the Python interpreter with taint tracking before the web applications are deployed. In order to enforce these policies, Riverbed deploys the application in a new “silo” for each new clause, such as a combination of prohibited third-party domains, encountered in an end user’s privacy policy. The modified interpreter, with taint tracking, serves as a runtime monitor that implements the same access control rules within each silo (?? holds).

We observe that Riverbed applications consist of very few microservices in containers, thus requirement ?? is not satisfied. Moreover, their applications do not involve extensive communication between these services. They also do not process users’ data for secondary purposes such as advertising, marketing, and analytics, which are common in modern web applications. Though Riverbed can support unmodified web applications, satisfying requirement ??, it is limited to Python applications, since it requires instrumentation of the Python interpreter, and thus it does not meet requirement ?. Riverbed does not encapsulate the notion of observed profiles (?, ?), or distinguish between per-service and system-wide profiles (?). Since the target policy enforced at runtime is equal to the user policy, Riverbed does not identify violations of the source code with a given target policy at development time (?). In Riverbed, Wang et al. seek to completely prevent any modifications to the source code (?? holds), including for the purposes of improving privacy, and do not consider any cases wherein the source code may need to be modified in order to meet a target policy.

Several access control and auditing frameworks have been proposed for distributed microservices architectures; these later frameworks satisfy ?. Xing et al. [LCL+21] provide an access control framework for inter-service communication, namely Autoarmor. Their goal is to prevent a container with an exploited microservice from accessing other microservices’ APIs. By design, Autoarmor does not change the microservices’ source code in achieving this goal (?? holds). However, Autoarmor does not mediate data shared with third parties. At build time, Autoarmor uses static analysis to extract inter-service requests from microservices’ code in different languages, and outputs access control policies in manifests. Once microservices are deployed, the manifests are distributed to the service mesh (Istio) or orchestrator control plane (Kubernetes), which is

configured to enforce the policies (?? holds). So, ?? holds but ?? only holds partially since each supported language requires instrumentation.

Datta et al. [DPI+22] develop a provenance-based auditing framework for serverless architectures named Alastor, with the goal of detecting data exfiltration attacks through provenance data. They modify the container runtime, such that each container (function) is instrumented with system call tracing and a transparent network proxy, to arrive at a local fine-grained provenance graph detailing network and file events (?? holds). Through this dynamic analysis, Alastor detects malicious requests based on keys passed in the request, which is not possible with the static analysis approach in Autoarmor. Importantly, Datta et al. compose local provenance graphs across containers to arrive at a global provenance graph (?? holds). Since Alastor is an auditing framework, it does not require changes to microservices' source code for its operations (?? holds) and also does not enforce any policies at runtime; requirement ?? is inapplicable for an auditing framework.

In the context of Prose, we consider the example microservices in Figure ?? . Suppose that the MARKETING service is compromised in production and privacy policy depicted in Listing ?? is being enforced. The compromised MARKETING service calls the GETUSERSBYEMAIL API of the USERS service, with the user's email as input and expects to receive the user's name, street address, age, gender and phone number. Since the privacy policy does not allow processing the latter PII types for the MARKETING purpose, this data flow will be dropped. We note that the callee (USERS) service will still be called; that is, the inbound request handler within its associated service proxy will not report any violations and dutifully allow the response. However, the outbound response handler within the (trusted) service proxy at the compromised caller service (REGISTRATION) will mediate the response, report an indirect purpose-of-use violation, and drop the data flow. In contrast, Autoarmor would block this flow within the outbound request handler of the service proxy of the compromised caller service. On the other hand, suppose that this compromised service now makes a call to BADWEBSITE.COM, with a user's phone number and email. For the purpose of registration, the aforementioned privacy policy does not explicitly allow sharing the user's phone number or email with BADWEBSITE.COM. Thus, in Prose, the authorization agent within the outbound request handler (Figure ??) of the compromised service would flag this data flow as a data sharing violation, and it would drop this data flow in enforce mode. As mentioned above, the access control policies generated by Autoarmor limit inter-service communication, and as such, Autoarmor has not been designed to limit data sharing with third parties.

Even if Autoarmor were to be extended to fix this drawback, both Autoarmor and

Alastor suffer from the following fundamental limitation: they use the observed behaviour at build time as the ground truth for the target behaviour at runtime, whether in the form of target access control policies for Autoarmor or the target provenance graphs for Alastor. In other words, in terms of the observed privacy profile and the target privacy policy in Prose, Autoarmor and Alastor assume that the target privacy policy is equal to the observed privacy profile. None of these systems is designed to identify (privacy or security) violations before deployment to production, and thus ?? does not hold. However, Prose categorically differentiates between the observed privacy profile and the target privacy policy, handles the fact that they may not be equal and assists software engineers in complying with a target privacy policy that is stricter than the observed privacy profile.

Wang et al.’s PrivGuard [WKN+22] assesses compliance of Python data analysis scripts to Legalese [SGD+14] policies. PrivGuard requires each data analyst to set a guard policy for each script, which serves as a per-script target policy. Privacy managers are required to set a system-wide target (base) policy, and users may set their own policies. All of these policies are enforced at runtime as follows. PrivGuard loads users’ data and policies into a trusted hardware platform, and ensures that the following three criteria hold before decrypting the users’ data to the analyst. First, the guard policy should be at least as strict as the base policy. Second, the script only obtains the data of users whose policies are at most as strict as the guard policy. Finally, they develop a static analyzer, PrivAnalyzer, to assess whether the script complies with the guard policy. For PrivAnalyzer to analyze library calls in the script, Wang et al. instrument library functions in statistical analysis, machine learning and differential privacy libraries, to account for what clauses of the policy may be satisfied through each function call. Any policy clauses that are not satisfied by the script are output as a residual policy, which should be satisfied by sequential scripts, before the users’ data is decrypted to the analyst. Wang et al.’s Legalese policies govern row-level and column-level filtering operations, de-identification through k-anonymity or DP, and machine learning operations.

Though PrivGuard enforces target policies at runtime, satisfying requirements ??, it does not satisfy many other requirements, limiting the applicability of its policy model and system implementation to our work. First, PrivGuard does not operate within a polyglot microservice architecture, and thus requirements ?? and ?? are not satisfied. Second, though the source code of scripts does not need to be modified, library functions called by the scripts need to be manually instrumented for PrivGuard to work, and thus requirement ?? is only satisfied partially. Third, their policies do not model communication within services or with third parties, and thus they cannot identify data sharing violations. Though requirement ?? can be satisfied in the context of compliance

of individual scripts to the per-script target policy, namely the guard policy, in general the PrivGuard policy model does not encapsulate the observed profiles (??) independently of the target policy, and this design choice impacts the usability of PrivGuard, as follows. PrivGuard does not deduce the weakest privacy policy that each script or the entire system adheres to, and thus does not satisfy requirement ??. Consequently, privacy managers adopting PrivGuard are expected to set a strong base policy, without knowing this observed privacy profile, and thus without knowledge of the number of scripts that would violate this base policy at runtime.

5.10 Conclusion

Using our privacy policy workflow, privacy engineers can iteratively realize Hoepman’s MINIMIZE strategy for a microservice architecture. Prose realizes all three tactics of the ENFORCE design strategy, which we describe in Section ??, for a microservice architecture: privacy engineers can set a target privacy policy, and thus Prose realizes the CREATE tactic of the ENFORCE strategy. Prose also implements the MAINTAIN tactic for this strategy as follows. First, Prose identifies services that violate the target privacy policy and informs software engineers of these violations. Second, through the two iterative fixing strategies, Prose assists privacy engineers in prioritizing fixing services and thereby setting an effective privacy policy that the fixed services are expected to follow in production. Third, violating services are deployed into production only after being fixed, and Prose also enforces the effective privacy policy in production. Our privacy policy workflow also enables privacy engineers to change the target and effective privacy policies, and Prose continues to identify violations as the services’ source code changes, thereby realizing the UPHOLD tactic for that strategy. Importantly, by directing privacy engineers and software engineers to fix new violations that arise due to changes in the source code, Prose continues to realize the MINIMIZE strategy, even as the microservices’ source code evolves.

Additionally, Prose implicitly operationalizes all tactics of the DEMONSTRATE strategy by serving as an auditing framework. Prose records data flows through the microservice architecture through distributed tracing, and thus implements the RECORD tactic, then processes the traces into per-service observed profiles and compares these profiles against the privacy policy, and thereby executes the AUDIT tactic. The per-service and system-wide observed profiles can serve as software-based evidence for data protection impact assessments, as Hoepman envisions in the REPORT tactic for this strategy. Moreover, observed profiles can be used to automatically derive textual privacy policies that reflect the back-end software. Thus, through its observed profiles,

Prose also enables privacy engineers to realize the INFORM strategy. Prose can be used to demonstrate compliance with data privacy regulations such as the GDPR and the CCPA by setting a strict target policy derived from these legislations.

Finally, in relation to our thesis statement, we demonstrate the efficiency of our data plane when it experiences large loads, and we examine the effectiveness our PII identification unit in detecting different PII types, while providing further advice to privacy engineers intending to deploy our system.

Chapter 6

Conclusion

We substantiate our defense of the thesis statement and conclude with future work.

6.1 Defense of the thesis statement

Through this thesis, we have built and integrated four efficient and effective PETs that realize Hoepman’s privacy design strategies within back-end web architectures.

In Chapter ??, we apply differential privacy to realize Hoepman’s ABSTRACT strategy for the back-end web architecture of centralized relational DBMSes. Within CacheDP, we compose differential privacy (DP) with a component of the back-end architecture, namely caches. In doing so, we address challenges such as estimating the privacy budget to answer a workload while using the cache and simultaneously meeting accuracy requirements, by formulating and solving a novel cost optimization problem. We also include modules that provide other functionalities that can be expected of a cache-aware system, such as prefilling the cache and maximizing cache reuse. Our differentially private cache is effective in that it reduces the privacy budget spent to answer a variety of single and multiple attribute workloads, as compared to state-of-the-art baselines. We thus demonstrate that features that are built in to back-end web architectures, such as caches, can be extended towards improving the effectiveness of a PET. We further examine the effectiveness of each of our modules in saving privacy budget for different tasks, and we find that each of our modules was relied upon for at least one task. We also illustrate that our modules are efficient in that they incur reasonable storage and computation overheads.

In Chapters ?? and ??, we consider peer-to-peer networks and distributed hash tables

as our back-end web architecture, and we address the problem of fetching content across peer-to-peer networks without revealing what was fetched. Both DHTPIR (Chapter ??) and Peer2PIR (Chapter ??) realize the MINIMIZE strategy, and they do so by using Private Information Retrieval (PIR). However, both of these systems realize the MINIMIZE strategy in different ways, since they use different types of PIR schemes, and they realize additional privacy design strategies based on the type of PIR scheme that they use. DHTPIR distributes the content of the query across peers in a quorum, and thus realizes Hoepman’s SEPARATE strategy. In contrast, Peer2PIR hides the content of the query from each peer in the query path, and thus realizes Hoepman’s HIDE strategy. We thus observe that the MINIMIZE strategy is often concomitant with other privacy design strategies.

In DHTPIR, we design a private file retrieval algorithm that uses an information-theoretic PIR (IT-PIR) scheme to realize the SEPARATE strategy. Information-theoretic PIR requires peers of quorums that do not collude but do share a copy of the content that is to be queried over. We then show that quorums with at most $\frac{1}{4}$ Byzantine nodes, which have been used independently in related work to form robust DHTs, can be used to satisfy the aforementioned prerequisite for IT-PIR schemes for DHTs. Our security analysis of our private file retrieval scheme shows its effectiveness, contingent on quorums having at most $\frac{1}{4}$ Byzantine nodes. We then further address challenges of integrating DHTPIR within the back-end web architecture, such as by navigating a tradeoff between the quorum size and the fraction of Byzantine nodes that it can withstand, and by providing a join protocol to enable new peers to answer queries privately. We then show the efficiency of DHTPIR through our complexity analysis as well as through our simulation. We analyze how the communication overheads of DHTPIR scale as the distributed hash network stores more content. We then simulate the computation and communication overheads of DHTPIR and other related work, and we find that the communication overheads match our analysis and that DHTPIR outperforms the closest related work in terms of computation overheads.

In Peer2PIR, we provide algorithms for peers to privately retrieve routing information, to discover content and to retrieve content within a distributed file system, namely IPFS. We use computational PIR, and specifically, homomorphic encryption (HE) schemes, to hide what is being queried in each case, such as the peer ID of a target peer or the content ID of a content block. We identify and address two main challenges when integrating computational PIR schemes within the DFS. First, each of the three aforementioned use cases involve tables of different shapes, and thus one computational PIR scheme would not be efficient for all three use cases. Second, existing computational PIR schemes are implicitly designed for a client-server model, wherein

one or more clients repeatedly query a single (logical) server, and thus amortize the communication overheads of PIR schemes across thousands of queries made to that server. However, we find that this assumption does not bear out for distributed hash tables, since each peer is designed to contact different peers for different content. We evaluate existing PIR schemes for our use cases, concluding that they do not provide a reasonable communication-computation tradeoff for two of our three cases, namely private peer routing and content discovery. We then provide two HE-based PIR schemes to address the aforementioned challenges. Our algorithms then integrate our proposed as well as existing PIR schemes to address each of the three aforementioned use cases, and we observe that we needed to reshape the data structures used natively within the DHT for PIR, as evidenced through our normalization algorithm. We demonstrate the efficiency of our system through its modest latency overheads over the baselines, and we justify its effectiveness through the proofs of correctness, security and convergence of our algorithms. We also analyze the impact of node and content churn on these overheads. We document our implementation process and then discuss how existing tests and tooling can be used to integrate PETs within an existing codebase, and how additional properties such as backwards compatibility and interoperability can be provided. These takeaways may be of independent interest to privacy engineers and researchers.

In Chapter ??, we examine a distributed microservices architecture, and we design a PET that software engineers and privacy engineers can use to ensure that their microservices architecture complies with a privacy policy. We define and exemplify secondary purpose-of-use and data sharing violations in this architecture. We formulate architectural and privacy engineering requirements that our system should follow. We then design a novel privacy policy workflow that is integrated within the software engineering lifecycle, and incorporates these requirements. Importantly, this workflow enables privacy engineers to continually realize the MINIMIZE and ENFORCE privacy design strategies, as both the underlying microservices evolve, and as the privacy engineers seek to adhere to stricter privacy policies. We then present the design of our system that meets these requirements, in terms of its data plane and its control plane. Our data plane consists of distributed runtime monitors, in that there is a service proxy attached to each microservice to monitor and mediate traffic, along with a PII identification unit to detect PII types, and an authorization agent to determine whether there is a violation of the privacy policy. Each service proxy is augmented with different handlers to detect and prevent all the aforementioned types of violations. Through its design components, including distributed tracing, the data plane implicitly realizes the DEMONSTRATE privacy design strategy. Our control plane enables privacy engineers to glean the data processing practices of individual microservices or the entire system through per-service

and system-wide observed privacy profiles respectively. Through our control plane, privacy engineers can also efficiently prototype multiple privacy policies and prioritize which services to fix in a limited time period. We show that our implementation experiences modest latencies under large request rates, and we thus demonstrate its efficiency. We examine the soundness and completeness of our PII identification unit in detecting various PII types, and demonstrate the effectiveness of our implementation.

6.2 Future work

CacheDP operates as an extension on top of an unmodified DBMS; cloud-based DBMSes such as Snowflake [Sno24b] that offer Differential Privacy as an end-user feature integrate the DP implementation *within* the implementation of the DBMS. Extending the implementation of cloud-based DBMSes to use a DP cache would involve architectural challenges. Relatedly, designing a cache-aware DP system for streaming data is an important research direction. Differential privacy can also be used to incentivize businesses to share data about their microservice architectures, as follows. Distributed tracing providers such as Jaeger allow DevOps engineers within an organization to view a service map of their microservices architecture. We remind the reader that nodes in a service map represent services or third parties, and edges represent communication between services or between a service and a third party. Service maps are useful for system reliability research, however, businesses do not wish to publish these maps, since they reveal confidential data, such as how many services are deployed and their communication patterns. *Graph differential privacy* can be applied on the service map graph, resulting in a perturbed map that can be published, without revealing the aforementioned sensitive characteristics of the original map. The benefit would be for a distributed tracing provider to be able to analyze these different service maps and provide reliability insights to their customers.

As the IPFS ecosystem evolves, strengthening it against other network attacks remains relevant. Further testing and deployment of Peer2PIR is an important line of future work. We aim to integrate and evaluate the implementations of PIR schemes published after our work. In order to deploy Peer2PIR, we would also benefit from experimentally evaluating the end-to-end latency for private peer routing, provider routing and content retrieval. In the context of DHTPIR, we can revisit combining single and multiple-server PIR schemes within the distributed setting, in order to forego the requirement that multiple servers need to share a copy of the database while not sharing users' requests or responses [DCH22].

We leave implementation of our Prose control plane to future work. A user study

involving software and privacy engineers would be useful in evaluating the effectiveness of Prose as a whole. We would be interested in the utility of our iterative fixing strategies and how privacy engineers set and change the effective privacy policy over time. We would also seek feedback from software engineers' experience in fixing violations, especially indirect purpose-of-use violations, which may require two teams to collaborate. We expect this user study would assist in improving Prose.

References

- [AAAG22] Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. “Pantheon: Private Retrieval from Public Key-Value Store”. In: *Proc. VLDB Endow.* 16.4 (2022), pp. 643–656. ISSN: 2150-8097. DOI: [10.14778/3574245.3574251](https://doi.org/10.14778/3574245.3574251). URL: <https://doi.org/10.14778/3574245.3574251>.
- [ACC+18] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. *Homomorphic Encryption Security Standard*. Tech. rep. Toronto, Canada: HomomorphicEncryption.org, 2018. URL: <https://homomorphicencryption.org/wp-content/uploads/2024/08/Homomorphic-Encryption-Standard-v1.1.pdf>.
- [ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. “PIR with Compressed Queries and Amortized Query Processing”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 962–979. DOI: [10.1109/SP.2018.00062](https://doi.org/10.1109/SP.2018.00062).
- [ADG+22] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. “How to Abuse and Fix Authenticated Encryption Without Key Commitment”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022, pp. 3291–3308. ISBN: 978-1-939133-31-1. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/albertini>.
- [Adm24] International Trade Administration. *Key Requirements for DPF Program Participating Organizations*. 2024. URL: <https://www.dataprivacyframework.gov/key-requirements> (visited on 2024-09-30).

- [Ala22] Alan S Wernick. *Twitter’s Deceptive Use of Customer Account Security Data*. 2022. URL: https://www.americanbar.org/groups/business_law/resources/business-law-today/2022-june/twitter-s-deceptive-use-of-customer-account-security-data/ (visited on 2023-12-20).
- [ALP+21] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. “Communication–Computation Trade-offs in PIR”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 1811–1828. ISBN: 978-1-939133-24-3. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/ali> (visited on 2023-09-28).
- [Ama19] Amazon Web Services. *Delivering Excellence With Microservices On AWS*. 2019. URL: <https://www.youtube.com/watch?v=otADklyugzY> (visited on 2024-05-04).
- [Ama24a] Amazon Web Services, Inc. *Amazon EC2 Documentation*. 2024. URL: <https://aws.amazon.com/documentation-overview/ec2/> (visited on 2024-09-25).
- [Ama24b] Amazon Web Services, Inc. *Amazon EKS architecture*. 2024. URL: <https://docs.aws.amazon.com/eks/latest/userguide/eks-architecture.html> (visited on 2024-10-18).
- [Apa24] The Apache Software Foundation. *Mesos Architecture*. 2024. URL: <https://mesos.apache.org/documentation/latest/architecture/> (visited on 2024-08-10).
- [ARA+19] Brooke Auxier, Lee Rainie, Monica Anderson, Andrew Perrin, Madhu Kumar, and Erica Turner. *Americans and Privacy: Concerned, Confused and Feeling Lack of Control Over Their Personal Information*. 2019-11-15. URL: <https://www.pewresearch.org/internet/2019/11/15/americans-and-privacy-concerned-confused-and-feeling-lack-of-control-over-their-personal-information/> (visited on 2023-03-10).
- [Arg24] Argo. *What Is Argo CD?* 2024-11-30. URL: <https://argo-cd.readthedocs.io/en/stable/#what-is-argo-cd/> (visited on 2025-02-11).
- [AS09] Baruch Awerbuch and Christian Scheideler. “Towards a Scalable and Robust DHT”. In: *Theory of Computing Systems* 45.2 (2009), pp. 234–260. DOI: [10.1007/s00224-008-9099-9](https://doi.org/10.1007/s00224-008-9099-9).

- [Aut22] Envoy Project Authors. *Tracing*. 2022. URL: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/observability/tracing (visited on 2021-02-15).
- [AYA+21] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. “Addra: Metadata-private voice communication over fully untrusted infrastructure”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 2021-07, pp. 313–329. ISBN: 978-1-939133-22-9. URL: <https://www.usenix.org/conference/osdi21/presentation/ahmad>.
- [Azu25] Azure. *CI/CD for microservices architectures*. 2025-02-11. URL: <https://learn.microsoft.com/en-us/azure/architecture/microservices/ci-cd> (visited on 2025-02-11).
- [Bar20] Elaine Barker. *Recommendation for Key Management: Part 1 - General*. Tech. rep. NIST SP 800-57pt1r5. Gaithersburg, MD: National Institute of Standards and Technology, 2020-05, NIST SP 800-57pt1r5. DOI: [10.6028/NIST.SP.800-57pt1r5](https://doi.org/10.6028/NIST.SP.800-57pt1r5). (Visited on 2024-04-11).
- [BBB+22] Chris Baumohl, Suzanne Bernstein, Alan Butler, John Davisson, Caitriona Fitzgerald, Grant Fergusson, Christopher Frascella, Sara Geoghegan, Calli Schroeder, and Ben Winters. *Disrupting Data Abuse: Protecting Consumers from Commercial Surveillance in the Online Ecosystem*. Comments of the Electronic Privacy Information Center. Electronic Privacy Information Center, 2022-11-01. URL: <https://epic.org/documents/disrupting-data-abuse-protecting-consumers-from-commercial-surveillance-in-the-online-ecosystem/>.
- [Bec15] Martin Beck. “Randomized decryption (RD) mode of operation for homomorphic cryptography - increasing encryption, communication and storage efficiency”. In: *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2015, pp. 220–226. DOI: [10.1109/INFOCOMW.2015.7179388](https://doi.org/10.1109/INFOCOMW.2015.7179388).
- [BEM+17] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. “Prochlo: Strong Privacy for Analytics in the Crowd”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. 2017, pp. 441–459. DOI: [10.1145/3132747.3132769](https://doi.org/10.1145/3132747.3132769).

- [BGKT12] Michael Backes, Ian Goldberg, Aniket Kate, and Tomas Toft. “Adding Query Privacy to Robust DHTs”. In: *7th ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’12. 2012, pp. 30–31.
- [BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) fully homomorphic encryption without bootstrapping”. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS ’12. New York, NY, USA: Association for Computing Machinery, 2012-01, pp. 309–325. ISBN: 978-1-4503-1115-1. DOI: [10.1145 / 2090236.2090262](https://doi.org/10.1145/2090236.2090262). URL: <https://dl.acm.org/doi/10.1145/2090236.2090262> (visited on 2023-09-22).
- [BJPM16] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O’Reilly Media, Inc, 2016. URL: <http://landing.google.com/sre/book.html>.
- [BM07] Ingmar Baumgart and Sebastian Mies. “S/Kademlia: A practicable approach towards secure key-based routing”. In: *2007 International Conference on Parallel and Distributed Systems*. 2007, pp. 1–8.
- [BPZ13] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. “Practical Perfect Hashing in Nearly Optimal Space”. In: *Information Systems* 38.1 (2013), pp. 108–131. ISSN: 0306-4379. DOI: [10.1016/j.is.2012.06.002](https://doi.org/10.1016/j.is.2012.06.002).
- [Bra12] Zvika Brakerski. “Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP”. en. In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 868–886. ISBN: 978-3-642-32009-5. DOI: [10.1007/978-3-642-32009-5_50](https://doi.org/10.1007/978-3-642-32009-5_50).
- [Bro20] Justin Brookman. *Understanding the Scope of Data Collection by Major Technology Platforms*. 2020. URL: https://digital-lab.consumerreports.org/wp-content/uploads/2021/02/Understanding-the-scope-of-data-collection-by-major-platforms_2020_FINAL.pdf (visited on 2023-03-10).
- [But24] Alan Butler. *Congress Re-enters the Debate on Data Minimization with APRA*. 2024-04-09. URL: <https://epic.org/congress-re-enters-the-debate-on-data-minimization-with-apra/> (visited on 2024-09-30).
- [Cav11] Ann Cavoukian. *The 7 Foundational Principles Implementation and Mapping of Fair Information Practices*. 2011. URL: https://iapp.org/media/pdf/resource_center/pbd_implement_7found_principles.pdf (visited on 2025-05-25).

- [CCR19] Hao Chen, Ilaria Chillotti, and Ling Ren. “Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’19. New York, NY, USA: Association for Computing Machinery, 2019-11, pp. 345–360. ISBN: 978-1-4503-6747-9. DOI: [10.1145/3319535.3354226](https://doi.org/10.1145/3319535.3354226). (Visited on 2023-06-26).
- [CE22] Consumer Reports and Electronic Privacy Information Center. *How the FTC Can Mandate Data Minimization Through a Section 5 Unfairness Rule-making*. 2022-01-26. URL: https://epic.org/wp-content/uploads/2022/01/CR_Epic_FTCDDataMinimization_012522_VF_.pdf (visited on 2024-09-30).
- [CF14] Tobias Clemson and Martin Fowler. *Testing strategies in a Microservice Architecture*. 2014-11-18. URL: <https://martinfowler.com/articles/microservice-testing/> (visited on 2025-02-12).
- [CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. “Towards Doubly Efficient Private Information Retrieval”. In: *Theory of Cryptography*. Ed. by Yael Kalai and Leonid Reyzin. Cham: Springer International Publishing, 2017, pp. 694–726. ISBN: 978-3-319-70503-3. DOI: [10.1007/978-3-319-70503-3_23](https://doi.org/10.1007/978-3-319-70503-3_23).
- [Chu25] Manish Chughtu. *Demystifying Istio’s Sidecar Injection Model*. 2025. URL: <https://istio.io/v1.14/blog/2019/data-plane-setup/> (visited on 2019-01-31).
- [CK20] Henry Corrigan-Gibbs and Dmitry Kogan. “Private Information Retrieval with Sublinear Online Time”. In: *Advances in Cryptology – EUROCRYPT 2020*. Ed. by Anne Canteaut and Yuval Ishai. Cham: Springer International Publishing, 2020, pp. 44–75. ISBN: 978-3-030-45721-1.
- [CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. “Private Information Retrieval”. In: *Journal of the ACM* 45.6 (1998), pp. 965–981. ISSN: 0004-5411. DOI: [10.1145/293347.293350](https://doi.org/10.1145/293347.293350).
- [CL22] Ling Cao and Yue Li. “IPFS keyword search based on double-layer index”. In: *International Conference on Electronic Information Engineering and Computer Communication (EIECC 2021)*. Ed. by Zhiyuan Zhu and Fengxin Cen. Vol. 12172. International Society for Optics and Photonics. SPIE, 2022, p. 1217209. DOI: [10.1117/12.2634670](https://doi.org/10.1117/12.2634670).
- [CL99] Miguel Castro and Barbara Liskov. “Practical Byzantine Fault Tolerance”. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. 1999.

- [Clo21] Cloud Native Computing Foundation. *Cloud Native Observability: hurdles remain to understanding the health of systems*. 2021. URL: https://www.cncf.io/wp-content/uploads/2022/03/CNCF_Observability_MicroSurvey_030222.pdf (visited on 2025-02-11).
- [Clo23] Cloud Native Computing Foundation. *Cloud Native 2023: The Undisputed Infrastructure of Global Technology*. 2023. URL: <https://www.cncf.io/reports/cncf-annual-survey-2023/> (visited on 2024-05-04).
- [Clo24a] Cloud Native Computing Foundation. *Graduated and Incubating Projects*. 2024. URL: <https://www.cncf.io/projects/> (visited on 2024-05-04).
- [Clo24b] Cloud Native Computing Foundation. *Projects*. 2024. URL: <https://landscape.cncf.io/stats> (visited on 2024-05-04).
- [Clo24c] Cloud Native Computing Foundation. *Scheduling & Orchestration*. 2024. URL: <https://landscape.cncf.io/guide#orchestration-management--scheduling-orchestration> (visited on 2024-08-10).
- [Clo24d] Cloudflare. *What is an API endpoint?* 2024. URL: <https://www.cloudflare.com/learning/security/api/what-is-api-endpoint/> (visited on 2024-03-13).
- [CLS24] Leo de Castro, Kevin Lewi, and Edward Suh. “WhisPIR: Stateless Private Information Retrieval with Low Communication”. In: *Cryptology ePrint Archive* (2024).
- [Com18] European Commission. *How much data can be collected?* 2018. URL: https://ec.europa.eu/info/law/law-topic/data-protection/reform/rules-business-and-organisations/principles-gdpr/how-much-data-can-be-collected_en (visited on 2022-06-16).
- [Com22] NYC Taxi & Limousine Commission. *TLC Trip Record Data*. 2022. URL: <https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page> (visited on 2022-11-25).
- [Cyp24] Cypress.io. *Why Cypress?* 2024-11-08. URL: <https://docs.cypress.io/app/continuous-integration/overview> (visited on 2025-02-22).
- [Dam20] Damien Giry. *Cryptographic Keylength Recommendation*. 2020-05-24. URL: <https://www.keylength.com/en/4/> (visited on 2025-04-07).
- [Dav23] John Davisson. *Data Minimization: A Pillar of Data Security, But More Than That Too*. 2023-06-22. URL: <https://epic.org/data-minimization-a-pillar-of-data-security-but-more-than-that-too/> (visited on 2024-09-30).

- [DCH22] Natnatee Dokmai, L. Jean Camp, and Ryan Henry. “Assisted Private Information Retrieval”. In: *Cryptology ePrint Archive* (2022). Paper: [2022 / 1082](https://eprint.iacr.org/2022/1082). URL: <https://eprint.iacr.org/2022/1082>.
- [DDP21] Alfonso De la Rocha, David Dias, and Yiannis Psaras. “Accelerating content routing with bitswap: A multi-path file transfer protocol in IPFS and Filecoin”. In: *San Francisco, CA, USA* (2021), p. 11.
- [DET23] Erik Daniel, Marcel Ebert, and Florian Tschorsch. “Improving Bitswap Privacy with Forwarding and Source Obfuscation”. In: *2023 IEEE 48th Conference on Local Computer Networks (LCN)*. IEEE. 2023, pp. 1–4.
- [Dil21] Newmeyer Dillion. *The CCPA and CPRA: Risks and Obligations with Third Parties*. 2021-06-23. URL: <https://www.newmeyerdillion.com/publications/the-ccpa-and-cpra-risks-and-obligations-with-third-parties/> (visited on 2023-03-10).
- [DKY17] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. “Collecting Telemetry Data Privately”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. 2017, pp. 3574–3583. DOI: [10.5555/3294996.3295115](https://doi.org/10.5555/3294996.3295115).
- [DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. “Calibrating Noise to Sensitivity in Private Data Analysis”. In: *Theory of Cryptography*. Ed. by Shai Halevi and Tal Rabin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 265–284. ISBN: 978-3-540-32732-5.
- [DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. “Tor: The Second-generation Onion Router”. In: *Proceedings of the 13th USENIX Security Symposium*. San Diego, CA, 2004, pp. 21–21.
- [Doc25a] Docker Inc. *Swarm mode overview*. 2025-01-29. URL: <https://docs.docker.com/engine/swarm/> (visited on 2024-03-04).
- [Doc25b] Docker, Inc. *Docker Hub*. 2025-01-21. URL: <https://docs.docker.com/docker-hub/> (visited on 2025-03-04).
- [DPC23] Alex Davidson, Gonalo Pestana, and Sofia Celi. “FrodoPIR: Simple, Scalable, Single-Server Private Information Retrieval”. In: *Proceedings on Privacy Enhancing Technologies* 2023.1 (2023), pp. 365–383. DOI: <https://doi.org/10.56553/popets-2023-0022>.

- [DPI+22] Pubali Datta, Isaac Polinsky, Muhammad Adil Inam, Adam Bates, and William Enck. “ALASTOR: Reconstructing the Provenance of Serverless Intrusions”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 2443–2460. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/datta>.
- [DR14] Cynthia Dwork and Aaron Roth. “The Algorithmic Foundations of Differential Privacy”. In: *Foundations and Trends in Theoretical Computer Science* 9.3–4 (2014), pp. 211–407. DOI: [10.1561/04000000042](https://doi.org/10.1561/04000000042).
- [DT23] Erik Daniel and Florian Tschorsch. “Privacy-Enhanced Content Discovery for Bitswap”. In: *2023 IFIP Networking Conference (IFIP Networking)*. IEEE. 2023, pp. 1–9.
- [Duf23] Philippe Dufresne. *Submission of the Office of the Privacy Commissioner of Canada on Bill C-27, the Digital Charter Implementation Act*, 2022. 2023. URL: https://www.priv.gc.ca/en/opc-actions-and-decisions/submissions-to-consultations/sub_indu_c27_2304/#toc3-3 (visited on 2024-09-30).
- [Env23] Envoy Project Authors. *What is Envoy*. 2023. URL: https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy (visited on 2023-03-10).
- [Env25] Envoy Project Authors. *Golang*. 2025. URL: https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/golang_filter#golang (visited on 2025-03-09).
- [Eth23] Ethyca. *Fides Language Documentation*. 2023. URL: <https://ethyca.github.io/fideslang/> (visited on 2023-03-10).
- [Exp25] Explosion. *Named Entity Recognition*. 2025. URL: <https://spacy.io/usage/linguistic-features#named-entities> (visited on 2025-03-04).
- [EZBK20] Philipp Eichmann, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. “IDEBench: A Benchmark for Interactive Data Exploration”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1555–1569. DOI: [10.1145/3318464.3380574](https://doi.org/10.1145/3318464.3380574).
- [Fai23] Lesley Fair. *FTC says online counseling service BetterHelp pushed people into handing over health information – and broke its privacy promises*. 2023-03-03. URL: <https://www.ftc.gov/business-guidance/blog/2023/03/ftc-says-online-counseling-service-betterhelp-pushed-people-handing-over-health-information-broke> (visited on 2023-03-18).
- [Fal25] The Falco Authors. *Falco Rules*. 2025-01-14. URL: <https://falco.org/docs/concepts/rules/> (visited on 2025-01-24).

- [Fed22] Federal Trade Commission. *FTC Charges Twitter with Deceptively Using Account Security Data to Sell Targeted Ads*. 2022-05-25. URL: <https://www.ftc.gov/news-events/news/press-releases/2022/05/ftc-charges-twitter-deceptively-using-account-security-data-sell-targeted-ads> (visited on 2023-03-10).
- [Fed23] Federal Trade Commission. *FTC Finalizes Order with 1Health.io Over Charges it Failed to Protect Privacy and Security of DNA Data and Unfairly Changed its Privacy Policy*. 2023. URL: <https://www.ftc.gov/news-events/news/press-releases/2023/09/ftc-finalizes-order-1healthio-over-charges-it-failed-protect-privacy-security-dna-data-unfairly> (visited on 2023-11-05).
- [Fed24a] Federal Trade Commission. *A Look Behind the Screens: Examining the Data Practices of Social Media and Video Streaming Services*. 2024-09. URL: https://www.ftc.gov/system/files/ftc_gov/pdf/Social-Media-6b-Report-9-11-2024.pdf (visited on 2024-09-30).
- [Fed24b] Federal Trade Commission. *The Federal Trade Commission 2023 Privacy and Data Security Update*. 2024-03-28. URL: https://www.ftc.gov/system/files/ftc_gov/pdf/2024.03.21-PrivacyandDataSecurityUpdate-508.pdf (visited on 2024-09-30).
- [Fle22a] Fleek. *Overview*. 2022. URL: <https://docs.fleek.co/space-daemon/overview/> (visited on 2023-03-10).
- [Fle22b] Fleek. *Site Deployment*. 2022. URL: <https://docs.fleek.co/hosting/site-deployment/> (visited on 2023-03-10).
- [Flu25] The Flux authors. *Core Concepts*. 2025. URL: <https://fluxcd.io/flux/concepts/> (visited on 2025-02-11).
- [FSY05] Amos Fiat, Jared Saia, and Maxwell Young. “Making Chord Robust to Byzantine Attacks”. In: *Proceedings of the 13th Annual European Conference on Algorithms*. ESA’05. Palma de Mallorca, Spain: Springer-Verlag, 2005, pp. 803–814.
- [FTG+24] Konstantin Fischer, Ivana Trummová, Phillip Gajland, Yasemin Acar, Sascha Fahl, and Angela Sasse. “The Challenges of Bringing Cryptography from Research Papers to Products: Results from an Interview Study with Experts”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, 2024. ISBN: 978-1-939133-44-1. URL:

<https://www.usenix.org/conference/usenixsecurity24/presentation/fischer>.

- [FV12] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2012/144. 2012.
- [Gal17] Eva Galperin. *Access Now and EFF Condemn the Arrest of Tor Node Operator Dmitry Bogatov in Russia*. <https://www.eff.org/deeplinks/2017/04/access-now-and-eff-condemn-arrest-tor-node-operator-dmitry-bogatov-russia>. 2017. (Visited on 2021-07-03).
- [GDM22] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. “On the rise and fall of CI services in GitHub”. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022, pp. 662–672. DOI: [10.1109/SANER53432.2022.00084](https://doi.org/10.1109/SANER53432.2022.00084).
- [GHIM19] Chang Ge, Xi He, Ihab F. Ilyas, and Ashwin Machanavajjhala. “APEX: Accuracy-Aware Differentially Private Data Exploration”. In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. 2019, pp. 177–194. DOI: [10.1145/3299869.3300092](https://doi.org/10.1145/3299869.3300092).
- [GHK+16] Marco Gaboardi, James Honaker, Gary King, Jack Murtagh, Kobbi Nissim, Jonathan Ullman, and Salil Vadhan. “Psi: a private data sharing interface”. In: (2016). URL: <http://arxiv.org/abs/1609.04340>.
- [GHV19] Marco Gaboardi, Michael Hay, and Salil Vadhan. *A Programming Framework for OpenDP*. 2019. URL: <https://projects.iq.harvard.edu/opendp> (visited on 2022-11-25).
- [Git24a] Github, Inc. *Building and testing*. 2024-02-03. URL: <https://docs.github.com/en/actions/use-cases-and-examples/building-and-testing> (visited on 2025-02-12).
- [Git24b] Github, Inc. *Deploying to Azure Kubernetes Service*. 2024-12-12. URL: <https://docs.github.com/en/actions/use-cases-and-examples/deploying/deploying-to-google-kubernetes-engine> (visited on 2025-02-12).
- [Git24c] Github, Inc. *Deploying to Google Kubernetes Engine*. 2024-12-12. URL: <https://docs.github.com/en/actions/use-cases-and-examples/deploying/deploying-to-google-kubernetes-engine> (visited on 2025-02-12).
- [Git24d] Github, Inc. *Understanding GitHub Actions*. 2024-01-17. URL: <https://docs.github.com/en/actions/about-github-actions/understanding-github-actions> (visited on 2025-02-12).

- [Git25a] Github, Inc. *Working with the Container registry*. 2025-02-20. URL: <https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry> (visited on 2025-03-04).
- [Git25b] GitLab. *Get started with GitLab CI/CD*. 2025-02-07. URL: https://gitlab.com/gitlab-org/gitlab/-/blob/master/doc/ci/_index.md (visited on 2025-02-12).
- [Git25c] GitLab. *GitLab CI/CD examples*. 2025-02-11. URL: <https://docs.gitlab.com/ee/ci/examples/> (visited on 2025-02-12).
- [Git25d] Gitlab. *What is GitOps?* 2025. URL: <https://about.gitlab.com/topics/gitops/> (visited on 2025-02-11).
- [GMHI21] Chang Ge, Shubhankar Mohapatra, Xi He, and Ihab F. Ilyas. "Kamino: Constraint-Aware Differentially Private Data Synthesis". In: *Proceedings of the VLDB Endowment* 14.10 (2021), pp. 1886–1899. DOI: [10.14778/3467861.3467876](https://doi.org/10.14778/3467861.3467876).
- [Gol07] Ian Goldberg. "Improving the Robustness of Private Information Retrieval". In: *2007 IEEE Symposium on Security and Privacy (SP '07)*. IEEE, 2007, pp. 131–148.
- [Gol15] Kevin Goldsmith. *Microservices at Spotify*. 2015. URL: <https://www.youtube.com/watch?v=7LGPeBgNFuU> (visited on 2024-05-04).
- [Goo23] Google Cloud Platform. *GoogleCloudPlatform/microservices-demo*. 2023. URL: <https://github.com/GoogleCloudPlatform/microservices-demo> (visited on 2023-10-03).
- [Goo24a] Google Cloud Platform. *GKE overview*. 2024-10-15. URL: <https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview> (visited on 2024-09-25).
- [Goo24b] Google Cloud Platform. *Jump Start Solution: Ecommerce web app deployed on Kubernetes*. 2024. URL: <https://cloud.google.com/architecture/application-development/ecommerce-microservices> (visited on 2024-09-25).
- [GP25] Christoph U. Günther and Krzysztof Pietrzak. "Putting Sybils on a Diet: Securing Distributed Hash Tables using Proofs of Space". In: *Cryptology ePrint Archive* (2025). Paper: [2025/804](https://eprint.iacr.org/2025/804). URL: <https://eprint.iacr.org/2025/804>.
- [gpe18] gpestana. *Privacy preserving DHTs*. <https://github.com/gpestana/notes/issues/8>. 2018. (Visited on 2020-01-12).

- [Gra24] Grafana. *Considerations*. 2024. URL: <https://grafana.com/docs/k6/latest/testing-guides/test-types/stress-testing/#considerations> (visited on 2024-12-30).
- [Gre20] Brendan Gregg. *Flame Graphs*. 2020. URL: <https://www.brendangregg.com/flamegraphs.html> (visited on 2023-03-09).
- [Gre23] Marcel Gregoriadis. “Analysis and Comparison of Deduplication Strategies in IPFS”. MA thesis. Humboldt-Universität zu Berlin, 2023.
- [GTD15] Seda Gürses, Carmela Troncoso, and Claudia Diaz. “Engineering Privacy by Design Reloaded”. eng. In: *Amsterdam Privacy Conference (APC)*. 2015. URL: <https://www.esat.kuleuven.be/cosic/publications/article-2589.pdf>.
- [Has25] Hashicorp. *What is Nomad?* 2025-02-10. URL: <https://developer.hashicorp.com/nomad/intro#what-is-nomad> (visited on 2024-03-12).
- [HBK24] François Hublet, David Basin, and Srđan Krstić. “User-Controlled Privacy: Taint, Track, and Control”. In: *Proceedings on Privacy Enhancing Technologies*. 1. 2024. DOI: [10.56553/popets-2024-0018](https://doi.org/10.56553/popets-2024-0018).
- [HDG+24] Stefan Albert Horstmann, Samuel Domiks, Marco Gutfleisch, Mindy Tran, Yasemin Acar, Veelasha Moonsamy, and Alena Naiakshina. “Those things are written by lawyers, and programmers are reading that. Mapping the Communication Gap Between Software Developers and Privacy Experts”. In: *Proceedings on Privacy Enhancing Technologies 2024* (1 2024). DOI: [10.56553/popets-2024-0010](https://doi.org/10.56553/popets-2024-0010). URL: <https://petsymposium.org/popets/2024/popets-2024-0010.pdf>.
- [Hel24] Helm Authors. *Charts*. 2024-04-20. URL: <https://helm.sh/docs/topics/charts/> (visited on 2025-06-17).
- [Hen15] Anne Henochowicz. “The Human Side of Censorship: Keyword Filtering and Censorship Directives on the Chinese Internet”. In: *Proceedings of the 5th USENIX Workshop on Free and Open Communications on the Internet (FOCI 2015)*. Washington D.C., USA, 2015. URL: https://www.usenix.org/sites/default/files/conference/protected-files/foci15_slides_henochowicz.pdf.

- [HHC+23] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. “One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, 2023, pp. 3889–3905. ISBN: 978-1-939133-37-3. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/henzinger>.
- [HKR15] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. “Robust Authenticated-Encryption AEZ and the Problem That It Solves”. In: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by E. Oswald and M. Fischlin. Lecture Notes in Computer Science. Springer, 2015, pp. 15–44.
- [Hoe14] Jaap-Henk Hoepman. “Privacy Design Strategies”. In: *ICT Systems Security and Privacy Protection*. Ed. by Nora Cuppens-Boulahia, Frédéric Cuppens, Sushil Jajodia, Anas Abou El Kalam, and Thierry Sans. 2014, pp. 446–459. DOI: [10.1007/978-3-642-55415-5_38](https://doi.org/10.1007/978-3-642-55415-5_38).
- [Hoe22] Jaap-Henk Hoepman. *Privacy Design Strategies (The Little Blue Book)*. Tech. rep. Radboud University, 2022. URL: <https://cs.ru.nl/~jhh/publications/pds-booklet.pdf>.
- [HRMS10] Michael Hay, Vibhor Rastogi, Gerome Miklau, and Dan Suciu. “Boosting the Accuracy of Differentially Private Histograms through Consistency”. In: *Proceedings of the VLDB Endowment* 3.1–2 (2010), pp. 1021–1032. DOI: [10.14778/1920841.1920970](https://doi.org/10.14778/1920841.1920970).
- [HSE19] Michelle Hertzfield, Jessica Schilling, and Oli Evans. *How IPFS Works - IPFS Documentation*. <https://docs.ipfs.io/introduction/how-ipfs-works/>. 2019.
- [IBM21] IBM. *Microservices in the enterprise, 2021: Real benefits, worth the challenges*. 2021. URL: <https://www.ibm.com/downloads/cas/OQG4AJAM> (visited on 2024-01-28).
- [Int18a] Intersoft Consulting. *Art. 15 GDPR Right of access by the data subject*. 2018. URL: <https://gdpr-info.eu/art-15-gdpr/> (visited on 2023-03-09).
- [Int18b] Intersoft Consulting. *Chapter 5 Transfers of personal data to third countries or international organisations*. 2018. URL: <https://gdpr-info.eu/chapter-5/> (visited on 2023-03-10).
- [Int18c] Intersoft Consulting. *General Data Protection Regulation GDPR*. 2018. URL: <https://gdpr-info.eu/> (visited on 2023-03-10).

- [Ist22] Istio Authors. *Bookinfo Application*. 2022. URL: <https://istio.io/latest/docs/examples/bookinfo/> (visited on 2021-11-13).
- [Ist23a] Istio Authors. *Istio Architecture*. 2023. URL: <https://istio.io/latest/docs/ops/deployment/architecture/> (visited on 2023-03-10).
- [Ist23b] Istio Authors. *The Istio service mesh*. 2023. URL: <https://istio.io/latest/about/service-mesh/> (visited on 2023-03-10).
- [Jac25] Mike Jacobs. *CI/CD baseline architecture with Azure Pipelines*. 2025-01-24. URL: <https://learn.microsoft.com/en-us/azure/devops/pipelines/architectures/devops-pipelines-baseline-architecture> (visited on 2025-02-11).
- [Jae23a] The Jaeger Authors. *Architecture*. 2023. URL: <https://www.jaegertracing.io/docs/latest/architecture/> (visited on 2023-03-09).
- [Jae23b] The Jaeger Authors. *Introduction*. 2023. URL: <https://www.jaegertracing.io/docs/latest/> (visited on 2023-03-10).
- [jbe15] jbenet. *IPFS Feedback*. <https://github.com/ipfs/notes/issues/318>. 2015. (Visited on 2020-01-12).
- [JNS18] Noah Johnson, Joseph P. Near, and Dawn Song. “Towards Practical Differential Privacy for SQL Queries”. In: vol. 11. 5. VLDB Endowment, 2018-01, pp. 526–539. DOI: [10.1145/3177732.3177733](https://doi.org/10.1145/3177732.3177733).
- [KCN+15] Jeffrey Knockel, Masashi Crete-Nishihata, Jason Q. Ng, Adam Senft, and Jedidiah R. Crandall. “Every Rose Has Its Thorn: Censorship and Surveillance on Social Video Platforms in China”. In: *Proceedings of the 5th USENIX Workshop on Free and Open Communications on the Internet (FOCI 2015)*. Washington D.C., USA, 2015. URL: <https://www.usenix.org/system/files/conference/foci15/foci15-paper-knockel.pdf>.
- [KG09] Aniket Kate and Ian Goldberg. “Distributed Key Generation for the Internet”. In: *29th IEEE International Conference on Distributed Computing Systems*. 2009, pp. 119–128. DOI: [10.1109/ICDCS.2009.21](https://doi.org/10.1109/ICDCS.2009.21).
- [KHP16] Fragkiskos Koufogiannis, Shuo Han, and George J. Pappas. “Gradual Release of Sensitive Data under Differential Privacy”. In: *Journal of Privacy and Confidentiality* 7.2 (2016). DOI: [10.29012/jpc.v7i2.649](https://doi.org/10.29012/jpc.v7i2.649).
- [Kie22] Cillian Kieran. *An Open-Source Taxonomy for Ex-ante Privacy*. Presented at Enigma ’22. 2022. URL: <https://www.usenix.org/conference/enigma2022/presentation/kieran>.

- [Kle17] Matt Klein. *Service mesh data plane vs. control plane*. 2017. URL: <https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc> (visited on 2023-03-10).
- [KNB24] Srđan Krstić, Hoang Nguyen, and David Basin. “Model-driven Privacy”. In: *Proceedings on Privacy Enhancing Technologies*. 1. 2024. DOI: [10.56553/po-pets-2024-0018](https://doi.org/10.56553/po-pets-2024-0018).
- [Koh96] Ron Kohavi. “Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid”. In: *KDD*. Vol. 96. 1996, pp. 202–207.
- [Kra10] Hugo Krawczyk. “Cryptographic Extraction and Key Derivation: The HKDF Scheme”. In: *Advances in Cryptology – CRYPTO 2010*. Ed. by Tal Rabin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 631–648. ISBN: 978-3-642-14623-7.
- [KTC+23] Kelly Kostopoulou, Pierre Tholoniāt, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. “Turbo: Effective Caching in Differentially-Private Databases”. In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP ’23. Association for Computing Machinery, 2023, pp. 579–594. DOI: [10.1145/3600006.3613174](https://doi.org/10.1145/3600006.3613174).
- [KTH+19] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. “PrivateSQL: A Differentially Private SQL Query Engine”. In: *Proceedings of the VLDB Endowment* 12.11 (2019-07), pp. 1371–1384. DOI: [10.14778/3342263.3342274](https://doi.org/10.14778/3342263.3342274).
- [Kub23] The Kubernetes Authors. *Overview*. 2023. URL: <https://kubernetes.io/docs/concepts/overview/> (visited on 2024-03-12).
- [Kub24] The Kubernetes Authors. *The Kubernetes network model*. 2024-09-18. URL: <https://kubernetes.io/docs/concepts/services-networking/#the-kubernetes-network-model> (visited on 2025-02-23).
- [Kub25] The Kubernetes Authors. *Minikube*. 2025-01-15. URL: <https://minikube.sigs.k8s.io/docs/> (visited on 2025-03-09).
- [Kyv24] The Kyverno Authors. *Policies and Rules*. 2024-04-08. URL: <https://kyverno.io/docs/kyverno-policies/> (visited on 2025-01-24).
- [LAH+12] Jiali Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. “Expectation and Purpose: Understanding Users’ Mental Models of Mobile App Privacy through Crowdsourcing”. In: *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. UbiComp ’12. 2012, pp. 501–510. DOI: [10.1145/2370216.2370290](https://doi.org/10.1145/2370216.2370290).

- [LCL+21] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. “Automatic Policy Generation for Inter-Service Access Control of Microservices”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 3971–3988. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/li-xing>.
- [Lev22] Alexandra S. Levine. *Suicide hotline shares data with for-profit spinoff, raising ethical questions*. 2022. URL: <https://www.politico.com/news/2022/01/28/suicide-hotline-silicon-valley-privacy-debates-00002617> (visited on 2023-03-10).
- [LG15] Wouter Lueks and Ian Goldberg. “Sublinear Scaling for Multi-Client Private Information Retrieval”. In: *Financial Cryptography and Data Security*. 2015, pp. 168–186.
- [LKB+21] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. “STORM: Refinement Types for Secure Web Applications”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2021-07. URL: <https://www.usenix.org/conference/osdi21/presentation/lehmann>.
- [LMH+15] Chao Li, Gerome Miklau, Michael Hay, Andrew McGregor, and Vibhor Rastogi. “The Matrix Mechanism: Optimizing Linear Counting Queries under Differential Privacy”. In: *The VLDB Journal* 24.6 (2015), pp. 757–781. DOI: [10.1007/s00778-015-0398-x](https://doi.org/10.1007/s00778-015-0398-x).
- [LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. “Doubly Efficient Private Information Retrieval and Fully Homomorphic RAM Computation from Ring LWE”. In: *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*. Orlando FL USA: ACM, 2023-06, pp. 595–608. ISBN: 978-1-4503-9913-5. DOI: [10.1145/3564246.3585175](https://doi.org/10.1145/3564246.3585175). (Visited on 2024-07-12).
- [LNR06] J. Liang, N. Naoumov, and K. Ross. “The Index Poisoning Attack in P2P File Sharing Systems”. In: *Proceedings IEEE International Conference on Computer Communications (INFOCOMM)*. 2006, pp. 1–12. DOI: [10.1109/INFOCOM.2006.232](https://doi.org/10.1109/INFOCOM.2006.232).
- [Lou21] Mike Loukides. *The Cloud in 2021: Adoption Continues*. 2021-12-07. URL: <https://www.oreilly.com/radar/the-cloud-in-2021-adoption-continues/> (visited on 2024-09-25).

- [LPPG20] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. “Code smells and refactoring: A tertiary systematic review of challenges and observations”. In: *Journal of Systems and Software* 167 (2020), p. 110610. ISSN: 0164-1212. DOI: [10.1016/j.jss.2020.110610](https://doi.org/10.1016/j.jss.2020.110610). URL: <https://www.sciencedirect.com/science/article/pii/S0164121220300881>.
- [LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. “On Ideal Lattices and Learning with Errors over Rings”. In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 1–23. ISBN: 978-3-642-13190-5. DOI: [10.1007/978-3-642-13190-5_1](https://doi.org/10.1007/978-3-642-13190-5_1).
- [LPX+22] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. “Enjoy your observability: an industrial survey of microservice tracing and analysis”. In: *Empirical Software Engineering* 27 (2022), pp. 1–28.
- [McL16] Craig McLuckie. *From Google to the world: The Kubernetes origin story*. 2016. URL: <https://cloud.google.com/blog/products/containers-kubernetes/from-google-to-the-world-the-kubernetes-origin-story> (visited on 2024-05-04).
- [MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. “OnionPIR: Response Efficient Single-Server PIR”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2292–2306. ISBN: 9781450384544. DOI: [10.1145/3460120.3485381](https://doi.org/10.1145/3460120.3485381).
- [McS10] Frank McSherry. “Privacy Integrated Queries: An Extensible Platform for Privacy-Preserving Data Analysis”. In: *Communications of the ACM* 53.9 (2010-09), pp. 89–97. DOI: [10.1145/1810891.1810916](https://doi.org/10.1145/1810891.1810916).
- [MEB+22] David McCandless, Tom Evans, Paul Barton, Dr Stephanie Starling, and Duncan Geere. *World’s Biggest Data Breaches & Hacks*. 2022. URL: <https://informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/> (visited on 2023-03-09).
- [Mer18] Sam Meredith. *Facebook-Cambridge Analytica: A timeline of the data hijacking scandal*. English. 2018-04-10. URL: <https://www.cnn.com/2018/04/10/facebook-cambridge-analytica-a-timeline-of-the-data-hijacking-scandal.html> (visited on 2023-03-09).

- [MF21] Charity Majors and Liz Fong-Jones. *Observability Engineering*. 1st edition. O'Reilly Media, Inc., 2021. ISBN: 9781492076438.
- [MGG21a] Miti Mazmudar, Stan Gurtler, and Ian Goldberg. “Do You Feel a Chill? Using PIR against Chilling Effects for Censorship-resistant Publishing”. In: *Proceedings of the 20th Workshop on Privacy in the Electronic Society*. WPES '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 53–57. ISBN: 9781450385275. DOI: [10.1145/3463676.3485612](https://doi.org/10.1145/3463676.3485612).
- [MGG21b] Miti Mazmudar, Stan Gurtler, and Ian Goldberg. *Do you feel a chill? Using PIR against chilling effects for censorship-resistant publishing*. Cryptology ePrint Archive, Report 2021/1195. <https://eprint.iacr.org/2021/1195>. 2021.
- [mgo18] mgoelzer. *[Protocol Design] How to create a fully private DHT*. <https://github.com/libp2p/developer-meetings/issues/6>. 2018. (Visited on 2020-01-12).
- [MHL+22] Miti Mazmudar, Thomas Humphries, Jiaxiang Liu, Matthew Rafuse, and Xi He. *Cache Me If You Can: Accuracy-Aware Inference Engine for Differentially Private Data Exploration*. Technical Report. University of Waterloo, 2022. DOI: [10.48550/arXiv.2211.15732](https://doi.org/10.48550/arXiv.2211.15732).
- [MHL+23] Miti Mazmudar, Thomas Humphries, Jiaxiang Liu, Matthew Rafuse, and Xi He. “Cache Me If You Can: Accuracy-Aware Inference Engine for Differentially Private Data Exploration”. In: *Proceedings of the VLDB Endowment* 16.4 (2023), pp. 574–586. DOI: [10.14778/3574245.3574246](https://doi.org/10.14778/3574245.3574246). URL: <https://www.vldb.org/pvldb/vol16/p574-mazmudar.pdf>.
- [Mic22a] Guillaume Michel. *Double-Hashing as a way to increase reader privacy*. <https://www.youtube.com/watch?v=VBlx-VvIZqU>. Accessed on 2024-02-08. 2022-11-20.
- [Mic22b] Microsoft. *Sidecar pattern*. 2022. URL: <https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar> (visited on 2023-03-09).
- [Mic22c] Microsoft Corporation. *PII entities supported by Presidio*. 2022-09-22. URL: https://microsoft.github.io/presidio/supported_entities/ (visited on 2025-02-24).
- [Mic23a] Guillaume Michel. *IPIP-373: Double Hash DHT Spec*. <https://github.com/ipfs/specs/pull/373>. Accessed on 2024-02-04. 2023-01-18.
- [Mic23b] Microsoft Corporation. *Presidio: Data Protection and De-identification SDK*. 2023. URL: <https://microsoft.github.io/presidio/> (visited on 2023-03-10).

- [Mic24] Microsoft Corporation. *PII entities supported by Presidio*. 2024-09-22. URL: https://microsoft.github.io/presidio/supported_entities/ (visited on 2025-06-17).
- [Mic25] Microsoft Corporation. *Installation*. 2025. URL: <https://playwright.dev/docs/intro> (visited on 2025-02-22).
- [MK22] Rasoul Akhavan Mahdavi and Florian Kerschbaum. “Constant-weight PIR: Single-round Keyword PIR via Constant-weight Equality Operators”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022-08, pp. 1723–1740. ISBN: 978-1-939133-31-1. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/mahdavi>.
- [MKA+08] Ashwin Machanavajjhala, Daniel Kifer, John Abowd, Johannes Gehrke, and Lars Vilhuber. “Privacy: Theory meets practice on the map”. English (US). In: *2008 IEEE 24th international conference on data engineering*. IEEE. Cancun, Mexico: IEEE, 2008, pp. 277–286. ISBN: 9781424418374. DOI: [10.1109/ICDE.2008.4497436](https://doi.org/10.1109/ICDE.2008.4497436).
- [MM02] Petar Maymounkov and David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. In: *Peer-to-Peer Systems*. Ed. by Peter Druschel, Frans Kaashoek, and Antony Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65.
- [Mor24] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. 2024-05-03. DOI: [10.6028/NIST.SP.800-38D](https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf). URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf> (visited on 2025-04-16).
- [MSM19] Ryan McKenna, Daniel Sheldon, and Gerome Miklau. *Graphical-model based estimation and inference for differential privacy*. 2019-01-25. arXiv: [1901.09136](https://arxiv.org/abs/1901.09136). (Visited on 2019-09-30).
- [MTS+12] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. “GUPT: Privacy Preserving Data Analysis Made Easy”. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2012, pp. 349–360. ISBN: 9781450312479. DOI: [10.1145/2213836.2213876](https://doi.org/10.1145/2213836.2213876). URL: <https://doi.org/10.1145/2213836.2213876>.

- [MVM25] Miti Mazmudar, Shannon Veitch, and Rasoul Akhavan Mahdavi. “Peer2PIR: Private Queries for IPFS”. In: *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2025, pp. 4068–4086. DOI: [10.1109/SP61157.2025.00231](https://doi.org/10.1109/SP61157.2025.00231).
- [MW22] Samir Jordan Menon and David J. Wu. “SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 930–947. DOI: [10.1109/SP46214.2022.9833700](https://doi.org/10.1109/SP46214.2022.9833700).
- [NBH+22] Priyanka Nanayakkara, Johes Bater, Xi He, Jessica Hullman, and Jennie Rogers. “Visualizing Privacy-Utility Trade-Offs in Differentially Private Data Releases”. In: *CoRR* (2022). arXiv: [2201.05964](https://arxiv.org/abs/2201.05964).
- [New20] Sam Newman. *Monolith to microservices : evolutionary patterns to transform your monolith*. English. First edition. 2020.
- [New22] BBC News. *Mental health helpline ends data-sharing relationship*. 2022-02-02. URL: <https://www.bbc.com/news/technology-60218894> (visited on 2023-03-09).
- [Off19] Office of the Privacy Commissioner of Canada. *PIPEDA in brief*. 2019. URL: https://www.priv.gc.ca/en/privacy-topics/privacy-laws-in-canada/the-personal-information-protection-and-electronic-documents-act-pipeda/pipeda_brief/ (visited on 2023-03-09).
- [OFI21] Kate O’Flaherty. *Is it time to leave WhatsApp – and is Signal the answer?* 2021. URL: <https://www.theguardian.com/technology/2021/jan/24/is-it-time-to-leave-whatsapp-and-is-signal-the-answer> (visited on 2023-09-03).
- [Ope22a] Open Policy Agent. *Overview*. 2022. URL: <https://www.openpolicyagent.org/docs/latest/#overview> (visited on 2022-04-02).
- [Ope22b] Open Policy Agent. *What is Policy Decoupling?* 2022. URL: <https://www.openpolicyagent.org/docs/latest/philosophy/#policy-decoupling> (visited on 2023-03-10).
- [Ope22c] The OpenTelemetry Authors. *What is OpenTelemetry?* 2022-06-10. URL: <https://opentelemetry.io/docs/concepts/what-is-opentelemetry/> (visited on 2022-07-18).
- [Ope23a] Open Policy Agent. *Introduction*. 2023. URL: <https://www.openpolicyagent.org/docs/latest/> (visited on 2023-03-09).
- [Ope23b] The OpenTelemetry Authors. *Data sources*. 2023-02-27. URL: <https://opentelemetry.io/docs/concepts/data-sources/> (visited on 2023-03-09).

- [Ope23c] The OpenTelemetry Authors. *Traces*. 2023. URL: <https://opentelemetry.io/docs/reference/specification/overview/#traces> (visited on 2023-03-09).
- [Ope23d] OpenZipkin. *Zipkin*. 2023-03-09. URL: <https://github.com/openzipkin/zipkin/> (visited on 2023-03-09).
- [Ope24] Open Policy Agent. *Integrating with the Go SDK*. 2024. URL: <https://www.openpolicyagent.org/docs/latest/integration/#integrating-with-the-go-sdk> (visited on 2024-12-30).
- [OPPW25] Hiroki Okada, Rachel Player, Simon Pohmann, and Christian Weinert. *Towards Practical Doubly-Efficient Private Information Retrieval*. 2025. DOI: [10.1007/978-3-031-78679-2_14](https://doi.org/10.1007/978-3-031-78679-2_14).
- [Ora24] Oracle. *1.2.1 What is MySQL?* 2024. URL: <https://dev.mysql.com/doc/refman/8.4/en/what-is-mysql.html> (visited on 2024-09-25).
- [Pag25] PagerDuty, Inc. *Incidents*. 2025-03-01. URL: <https://support.pagerduty.com/main/docs/incidents> (visited on 2025-03-07).
- [Pai99] Pascal Paillier. “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”. In: *Advances in Cryptology – Eurocrypt’99*. Ed. by Jacques Stern. Vol. 1592. Berlin, Heidelberg: Springer, 1999. DOI: [10.1007/3-540-48910-X_16](https://doi.org/10.1007/3-540-48910-X_16).
- [Par24] Parliament of Canada. *C-27 An Act to enact the Consumer Privacy Protection Act, the Personal Information and Data Protection Tribunal Act and the Artificial Intelligence and Data Act and to make consequential and related amendments to other Acts*. 2024. URL: <https://www.parl.ca/legisinfo/en/bill/44-1/c-27> (visited on 2024-11-05).
- [PMZ22] Bernd Prünster, Alexander Marsalek, and Thomas Zefferer. “Total eclipse of the heart—disrupting the InterPlanetary file system”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 3735–3752.
- [Pos24] The PostgreSQL Global Development Group. *PostgreSQL: The World’s Most Advanced Open Source Relational Database*. 2024. URL: <https://www.postgresql.org/> (visited on 2024-09-25).
- [Pri25] Proceedings on Privacy Enhancing Technologies Symposium. *PoPETS Artifact Review*. 2025. URL: <https://petsymposium.org/artifacts.php> (visited on 2025-04-23).
- [Pro20] The Tor Project. *What is GetTor?* <https://gettor.torproject.org/>. 2020. (Visited on 2020-11-10).

- [Pro23a] Protocol Labs. *Ecosystem directory*. 2023. URL: <https://ecosystem.ipfs.tech/> (visited on 2023-03-09).
- [Pro23b] Protocol Labs. *How IPFS works*. <https://docs.ipfs.tech/concepts/how-ipfs-works/>. Accessed on 2024-02-04. 2023-09-25.
- [Pro23c] Protocol Labs. *Privacy and Encryption*. 2023-03-02. URL: <https://docs.ipfs.tech/concepts/privacy-and-encryption/> (visited on 2025-04-04).
- [Pro24a] Protocol Labs. *Chunking*. 2024-08-05. URL: <https://docs.ipfs.tech/concepts/file-systems/#chunking> (visited on 2025-04-04).
- [Pro24b] Protocol Labs. *IPFS Specifications*. <https://github.com/ipfs/specs>. Accessed on 2024-02-04. 2024-02-01.
- [Pro24c] Protocol Labs. *Peering with content providers*. <https://docs.ipfs.tech/how-to/peering-with-content-providers/>. Accessed on 2024-02-04. 2024.
- [PSMI20] Austin Parker, Daniel Spoonhower, Jonathan Mace, and Rebecca Isaacs. *Distributed Tracing in Practice*. O'Reilly Media, Inc., 2020.
- [PYZ+13] S. Peng, Y. Yang, Z. Zhang, M. Winslett, and Y. Yu. "Query optimization for differentially private data management systems". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. Brisbane, QLD, Australia: IEEE, 2013, pp. 1093–1104. DOI: [10.1109/ICDE.2013.6544900](https://doi.org/10.1109/ICDE.2013.6544900).
- [QYL13] Wahbeh Qardaji, Weining Yang, and Ninghui Li. "Understanding hierarchical methods for differentially private histograms". In: *Proceedings of the VLDB Endowment*. Vol. 6. 14. 2013, pp. 1954–1965. URL: <http://www.vldb.org/pvldb/vol6/p1954-qardaji.pdf>.
- [Red24] Red Hat Inc. *What is a GitOps workflow?* 2024-05-14. URL: <https://www.redhat.com/en/topics/devops/what-is-gitops-workflow#introduction> (visited on 2025-02-11).
- [RSS+16] Ashwini Rao, Florian Schaub, Norman Sadeh, Alessandro Acquisti, and Ruogu Kang. "Expecting the Unexpected: Understanding Mismatched Privacy Expectations Online". In: *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. 2016, pp. 77–96. URL: <https://www.usenix.org/conference/soups2016/technical-sessions/presentation/rao>.
- [Sap24] Saptarshi Dutta Gupta, Statistics Canada. *Identifying Personal Identifiable Information (PII) in Unstructured Data with Microsoft Presidio*. 2024-03-15. URL: <https://www.statcan.gc.ca/en/data-science/network/identifying-personal-identifiable-information> (visited on 2025-02-24).

- [Sen24] Tomas Senart. *Vegeta*. 2024-07-29. URL: <https://github.com/tsenart/vegeta> (visited on 2024-12-30).
- [SF12] Siddhartha Sen and Michael J. Freedman. "Commensal Cuckoo: Secure Group Partitioning for Large-Scale Services". In: *ACM SIGOPS Operating Systems Review* 46.1 (2012), pp. 33–39. DOI: [10.1145/2146382.2146389](https://doi.org/10.1145/2146382.2146389).
- [SGD+14] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. "Bootstrapping Privacy Compliance in Big Data Systems". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. 2014, pp. 327–342. DOI: [10.1109/SP.2014.28](https://doi.org/10.1109/SP.2014.28).
- [Sig16] Signal. *Grand jury subpoena for Signal user data, Eastern District of Virginia*. 2016-10-04. URL: <https://signal.org/bigbrother/eastern-virginia-grand-jury/> (visited on 2024-11-19).
- [Sig21] Signal. *MLAT order from Luxembourg for Signal user data*. 2021-11-02. URL: <https://signal.org/bigbrother/northern-california-order/> (visited on 2024-11-19).
- [Sig24] Signal. *Search warrants for Signal user data, Santa Clara County*. 2024-08-08. URL: <https://signal.org/bigbrother/santa-clara-county/> (visited on 2024-11-19).
- [SJL+23] Chen Sun, Evan Jacobs, Daniel Lehmann, Andrew Crouse, and Supreeth Shastri. "GDPRxiv: Establishing the State of the Art in GDPR Enforcement". In: *Proceedings on Privacy Enhancing Technologies* 2023 (4 2023). DOI: [10.56553/popets-2023-0121](https://doi.org/10.56553/popets-2023-0121).
- [SM20] Frank M. Shipman and Catherine C. Marshall. "Ownership, Privacy, and Control in the Wake of Cambridge Analytica: The Relationship between Attitudes and Awareness". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. 2020, pp. 1–12. DOI: [10.1145/3313831.3376662](https://doi.org/10.1145/3313831.3376662).
- [SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '01. San Diego, California, USA, 2001, pp. 149–160.
- [SN20] Hamza Saleem and Muhammad Naveed. "SoK: Anatomy of Data Breaches". In: *Proceedings on Privacy Enhancing Technologies* 2020.4 (2020), pp. 153–174. DOI: <https://doi.org/10.2478/popets-2020-0067>.

- [SNDW06] A Singh, T-W Ngan, P Druschel, and DS Wallach. “Eclipse Attacks on Overlay Networks: Threats and Defenses”. In: *Proceedings IEEE International Conference on Computer Communications (INFOCOMM)*. IEEE. 2006, pp. 1–12.
- [Sno24a] Snowflake, Inc. *Key Concepts & Architecture*. 2024. URL: <https://docs.snowflake.com/en/user-guide/intro-key-concepts> (visited on 2024-09-25).
- [Sno24b] Snowflake, Inc. *Using privacy policies for differential privacy*. 2024. URL: <https://docs.snowflake.com/en/user-guide/diff-privacy/differential-privacy-admin-privacy-policies> (visited on 2024-09-25).
- [Son24] Jimmy Song. *Understanding How Envoy Sidecar Intercept and Route Traffic in Istio Service Mesh*. 2024. URL: <https://jimmysong.io/en/blog/understanding-how-envoy-sidecar-intercept-and-route-traffic-in-istio-service-mesh/> (visited on 2025-04-11).
- [Sta23] State of California Department of Justice. Office of the Attorney General. *California Consumer Privacy Act (CCPA)*. 2023. URL: <https://www.oag.ca.gov/privacy/ccpa> (visited on 2023-03-10).
- [Ste20] Graham Sterling. *Choosing a format*. 2020. URL: <https://www.confluent.io/blog/distributed-domain-driven-architecture-data-mesh-best-practices/#format> (visited on 2022-05-30).
- [STUD22] Marlene Saemann, Daniel Theis, Tobias Urban, and Martin Degeling. “Investigating GDPR Fines in the Light of Data Flows”. In: *Proceedings on Privacy Enhancing Technologies*. Vol. 2022. 2022. DOI: [10.2478/popets-2022-0082](https://doi.org/10.2478/popets-2022-0082).
- [SW02] Adam Stubblefield and Dan S. Wallach. *Dagster: Censorship-Resistant Publishing Without Replication*. Tech. rep. Rice University, 2002.
- [SWF20] Jessica Schilling, Chris Waring, and Bertrand Falguiere. *Modify the bootstrap peers list*. <https://docs.ipfs.io/how-to/modify-bootstrap-list/>. 2020.
- [TAV13] Edith Tom, Aybüke Aurum, and Richard Vidgen. “An exploration of technical debt”. In: *Journal of Systems and Software* 86.6 (2013), pp. 1498–1516. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2012.12.052>.
- [TCZ+20] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. “Towards Scalable Threshold Cryptosystems”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020.

- [TE22] Vinh Thong Ta and Max Hashem Eiza. “DataProVe: Fully Automated Conformance Verification Between Data Protection Policies and System Architectures”. In: *Proceedings on Privacy Enhancing Technologies*. Vol. 2022. 1. 2022, pp. 565–585. DOI: [10.2478/popets-2022-0028](https://doi.org/10.2478/popets-2022-0028).
- [TLV22] Mohammad Tahaei, Tianshi Li, and Kami Vaniea. “Understanding Privacy-Related Advice on Stack Overflow”. In: *Proceedings on Privacy Enhancing Technologies*. Vol. 2022. 2. 2022, pp. 114–131. DOI: [10.2478/popets-2022-0038](https://doi.org/10.2478/popets-2022-0038).
- [TMH+21] Yuchao Tao, Ryan McKenna, Michael Hay, Ashwin Machanavajjhala, and Gerome Miklau. “Benchmarking Differentially Private Synthetic Data Generation Algorithms”. In: *TPDP* (2021). URL: <https://tpdp.journalprivacyconfidentiality.org/2021/papers/NingUQKH21.pdf>.
- [Tra22a] United States Department of Transportation. *Bureau of Transportation Statistics*. 2022. URL: <https://transtats.bts.gov> (visited on 2022-11-25).
- [Tra22b] Dennis Trautwein. 2022-09-20 *Hydras Analysis*. 2022-09-20. URL: <https://github.com/probe-lab/network-measurements/blob/master/results/rfm21-hydras-performance-contribution.md#provider-distribution> (visited on 2024-02-04).
- [Tra22c] Dennis Trautwein. 2022-09-20 *Hydras Analysis*. 2022-09-20. URL: <https://github.com/probe-lab/network-measurements/blob/master/results/rfm21-hydras-performance-contribution.md#churn> (visited on 2024-02-04).
- [Tra25] Dennis Trautwein. *Nebula*. 2025-03-31. URL: <https://github.com/dennis-tra/nebula> (visited on 2025-04-04).
- [TRT+22] Dennis Trautwein, Aravindh Raman, Gareth Tyson, Ignacio Castro, Will Scott, Moritz Schubotz, Bela Gipp, and Yiannis Psaras. “Design and evaluation of IPFS: a storage layer for the decentralized web”. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 739–752.
- [TWB+21] Uthaipon Tantipongpipat, Chris Waites, Digvijay Boob, Amaresh Siva, and Rachel Cummings. “Differentially private synthetic mixed-type data generation for unsupervised learning”. In: *Intelligent Decision Technologies* (2021).
- [Uni23] University of Iowa. *Enforcement Database*. 2023. URL: <https://gdprxiv.org/> (visited on 2024-09-30).
- [Vah25] Anjo Vahldiek-Oberwagner. *Security Research Artifacts*. 2025-04-19. URL: <https://secartifacts.github.io/> (visited on 2025-04-23).

- [Var21] Ketan Varshneya. *Understanding design of microservices architecture at Netflix*. 2021. URL: <https://www.techaheadcorp.com/blog/design-of-microservices-architecture-at-netflix/> (visited on 2023-03-09).
- [VHKH11] Eugene Y. Vasserman, Victor Heorhiadi, Yongdae Kim, and Nicholas J. Hopper. *Censorship resistant overlay publishing*. Tech. rep. 11-027. University of Minnesota, 2011.
- [VRG19] Elisabet Lobo Vesga, Alejandro Russo, and Marco Gaboardi. “A Programming Framework for Differential Privacy with Accuracy Concentration Bounds”. In: *CoRR* (2019). arXiv: [1909.07918](https://arxiv.org/abs/1909.07918).
- [Wea23] Weaveworks. *CNCF On-Demand Webinar: Monolith to Microservices with Bite-Sized Kubernetes*. 2023. URL: <https://www.cncf.io/online-programs/cncf-on-demand-webinar-monolith-to-microservices-with-bite-sized-kubernetes/> (visited on 2024-01-28).
- [Wil24] Kara Williams. *Data minimization is the key to a meaningful privacy law*. 2024-09-05. URL: <https://epic.org/data-minimization-is-the-key-to-a-meaningful-privacy-law/> (visited on 2024-09-30).
- [WKM19] Frank Wang, Ronny Ko, and James Mickens. “Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 615–630. URL: <https://www.usenix.org/conference/nsdi19/presentation/wang-frank>.
- [WKN+22] Lun Wang, Usmann Khan, Joseph Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. “PrivGuard: Privacy Regulation Compliance Made Easier”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/wang-lun>.
- [WLS+21] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. “Design, monitoring, and testing of microservices systems: The practitioners’ perspective”. In: *Journal of Systems and Software* 182 (2021). ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2021.111061>.
- [WM01] Marc Waldman and David Mazières. “Tangler: A Censorship-Resistant Publishing System Based On Document Entanglements”. In: *Proceedings of the ACM Conference on Computer and Communications Security*. 2001, pp. 126–135.

- [WRC00] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. “Publius: A robust, tamper-evident, censorship-resistant, web publishing system”. In: *Proceedings of the 9th USENIX Security Symposium*. 2000, pp. 59–72.
- [WTH+22] Stefan Winter, Christopher S. Timperley, Ben Hermann, Jürgen Cito, Jonathan Bell, Michael Hilton, and Dirk Beyer. “A retrospective study of one decade of artifact evaluations”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 145–156. ISBN: 9781450394130. DOI: [10.1145/3540250.3549172](https://doi.org/10.1145/3540250.3549172).
- [WZL+20] Royce J Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. “Differentially Private SQL with Bounded User Contribution”. In: *Proceedings on Privacy Enhancing Technologies* 2020 (2 2020), pp. 230–250. DOI: [10.2478/popets-2020-0025](https://doi.org/10.2478/popets-2020-0025).
- [XDW+21] Yingtai Xiao, Zeyu Ding, Yuxin Wang, Danfeng Zhang, and Daniel Kifer. “Optimizing Fitness-for-Use of Differentially Private Linear Queries”. In: *Proceedings of the VLDB Endowment* 14.10 (2021), pp. 1730–1742. DOI: [10.14778/3467861.3467864](https://doi.org/10.14778/3467861.3467864).
- [YHA+16] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. “Precise, Dynamic Information Flow for Database-backed Applications”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2016. 2016, pp. 631–647.
- [YK GK10] Maxwell Young, Aniket Kate, Ian Goldberg, and Martin Karsten. “Practical robust communication in DHTs tolerating a byzantine adversary”. In: *2010 IEEE 30th International Conference on Distributed Computing Systems*. IEEE. 2010, pp. 263–272.
- [ZPZS23] Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. “PIANO: Extremely Simple, Single-Server PIR with Sublinear Server Computation”. In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 54–54. DOI: [10.1109/SP54263.2024.00055](https://doi.org/10.1109/SP54263.2024.00055). (Visited on 2024-04-21).
- [ZXX16] Jun Zhang, Xiaokui Xiao, and Xing Xie. “PrivTree: A Differentially Private Algorithm for Hierarchical Decompositions”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. ACM, 2016, pp. 155–170. ISBN: 978-1-4503-3531-7. DOI: [10.1145/2882903.2882928](https://doi.org/10.1145/2882903.2882928).