Abstract of "Compliant and Secure Databases" by Marilyn George, Ph.D., Brown University, October 2022.

Personal data is under constant threat in the modern world — from corporations looking to profit from over-collection and sale of personal data, to criminal interests who steal data for ransom, identity theft, and personal and corporate secrets. In response to the alarming rise in the exploitation of data, governments worldwide have begun enacting privacy legislation to give users more control over their personal data. However, there are technological constraints to making current systems compliant. Legacy systems are unlikely to have been designed with privacy considerations in mind. As such, it is difficult to instrument them in order to support the degree of transparency and access that are mandated by privacy legislation. Even if systems are instrumented to support user control over their data, they are still vulnerable to insider attacks and large-scale data breaches. The only fool-proof method to protect against such breaches is to use encryption for private data. However, plain encryption reduces the utility of outsourcing data, in that it does not allow the user to operate on their own data without downloading all of it. Then we turn to cryptographic primitives such as fully-homomorphic encryption (FHE), structured encryption (STE), property-preserving encryption (PPE) and oblivious RAM (ORAM). These primitives have all been widely studied and used to build systems that support various degrees of operation over encrypted data. Each of them also offers different trade-offs in efficiency and security. The security of these primitives can be quantified in terms of the leakage *i.e.*, meaningful information that is visible to an adversarial server. In this thesis, we describe work that advances the state-of-the-art in compliant and secure databases. We present: (1) a tool that will largely automate GDPR access requests on legacy databases, thereby reducing the manual work required to deal with custom schema and application logic; (2) a general leakage suppression framework for structured encryption schemes that support updates to the data structure, and, (3) efficient leakage suppression techniques for dictionary encryption schemes that do not support updates to the underlying dictionary structure.

Compliant and Secure Databases

by
Marilyn George
B. Tech., National Institute of Technology Calicut, 2013
M. E., Indian Institute of Science, 2016
M. S., Brown University, 2019

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
October 2022

This dissertation by Marilyn George is accepted in its present form by
the Department of Computer Science in partial fulfillment of the dissertation requirement
for the degree of Doctor of Philosophy.

Date ______________ ______________________________

Seny Kamara, Advisor
Brown University

Date ______________ ______________________________

David Cash, Reader
The University of Chicago

Date ______________ ______________________________

Malte Schwarzkopf, Reader
Brown University

Approved by the Graduate Council

Date ______________ ______________________________

Thomas A. Lewis
Dean of the Graduate School

Acknowledgements

This dissertation was made possible with the guidance and support of a very large number of people, all of whom I cannot possibly thank in this acknowledgements section, but I will try my best.

First, I would like to thank my advisor, Seny Kamara, for being an invaluable source of guidance and support throughout this journey. I have learned a lot about being a better researcher and collaborator from him. He has also been an exceptional mentor, caring not only about the quality of my work and presentation, but also about me as a human being, and working with me through several phases of exhaustion. I would have given up at many of those points if it were not for you, so thank you, once again.

I would also like to thank the faculty and staff of the Computer Science department at Brown, who have made the department a warm and welcoming space, and have always been willing to talk and offer advice and help whenever I needed it. I would like to thank the Kanellakis Family, the department, and the Coline M. Makepeace Fellowship for their generous support during my PhD. Further, I would like to thank the department for their continued support of the CS PhD Mentorship Program, which I have had the privilege of being part of for a large part of my PhD. I would like to thank Amy Greenwald in particular, for her guidance and belief in me all through the years. Amy is one of the most curious and energetic researchers I know, and I have been inspired just by watching her do what she loves. She also reached out to me during a period of burnout and made sure that I was doing okay, and it meant a lot that I could count on her support and concern for my wellbeing. I would also like to thank Malte Schwarzkopf, for being an enthusiastic collaborator and for teaching a private systems course that was kind to beginners like me. Our project together was born from this class, and the rest is history. A special note of thanks to Tarik Moataz, for being a very kind and patient mentor, and a reassuring presence in the group.

I would also like to thank all my other collaborators during my PhD, I have learned a lot from each of you, and I hope to continue learning in the future: Aaron Jeyaraj, Archita Agarwal, Eleanor Tursman, Enrique Areyan, Grace Boghosian, James Tompkin, Lucy Qin, Megumi Ando, Sam Zhao, and Zachary Espiritu, thank you! I would like to thank all the CS department faculty, in particular the faculty members who I have studied with and interacted with, and who helped me feel like a part of the Brown community: Anna Lysyanskaya, Bena Tshishiku, David Laidlaw, Eli Upfal, Eugene Charniak, George Konidaris, Jeff Huang, John Hughes, John Savage, Lorenzo De Stefani, Maurice Herlihy, Philip Klein, Roberto Tamassia, Shriram Krishnamurthi, Stefanie Tellex, Steven Reiss, Suresh Venkatasubramanian, Tom Doeppner, Ugur Cetintemel, and Vasileios Kemerlis. Further, I would like to thank David Cash, from the University of Chicago, for agreeing to be part of my committee, for his helpful feedback on this dissertation, and for his encouragement and support of my research.

I must thank Lauren Clarke, for her tireless support of the PhD students and patience with all my endless questioning and emails. I would also like to thank Jane McIlmail, for her friendship and positivity, and many coffee and tea chats that helped me get through the PhD. Further, I would also like to thank Eugenia DeGouveia and Dawn Reed for their help and support during the program. I would also like to thank the Data Science Initiative for their wonderful office space that I used in my last year, with a special note of thanks to Jane Albro and Nathaniel Gill for being warm and welcoming to our little band of CS graduate students. I would also like to thank the Sheridan Center, especially Charles Carroll, Eric Caldor, Jessica Metzler, and Stacey Lawrence, for helping me become a better teacher and learner.

Over the years the Brown CS community has been a source of friendship and support, and I would like to thank all my friends in the department. To my academic siblings, past and present, thank you for making the lab such a fun and supportive place to be a part of. I will fondly remember all our outings and discussions, and look forward to more! Archita Agarwal, Ghous Amjad, Kweku Kwegyir-Aggrey, Leah Rosenbloom, Lucy Qin, Sam Zhao, and Victor Youdom-Kemmoe, thank you! Also to my friends from department socials and activities, academic cousins, and my unforgettable GCB friends: Alejandro Scaffa, Amariah Becker, Amos Treiber, Apoorvaa Deshpande, Akhil Bagaria, Daniel Engel, Denizalp Goktas, Elizabeth Crites, Enrique Areyan, Francesca Falzon, Jasper Lee, Jiwon Choe, Lizzie Kumar, Lilika Markatou, Lucas Lehnert, Kaiyu Zheng, Megumi Ando, Michael Markovitch, Mira Nikolova, Qian Zhang, Saket Tiwari, Sam Thomas, Sam Zhao, Scott Griffy, Tarik Moataz, Theresa Barton, Thao Nguyen, Vikram Saraph, and Yingjie Xue, thank you for being part of my PhD journey!

I must also say my thanks to the people in the CS PhD Mentorship program, some of my best experiences in the department have been a part of this program. To my mentor, Nediyana Daskalova, for being there for me, and for founding the official mentorship program. To Jasmine Liu and Natasha Danas, for their help and support running the program, to Jasmine especially, for her friendship and encouragement all the way from opening convocation to scheduling our defenses. I would also like to thank Nihal Nayak and Yanyan Ren, who have helped us run the program, and will continue to run the program after our graduation. To everyone else who volunteered to be a mentor or mentee, or participated in the program at all, thank you, I had a great time getting to know all of you!

I would also like to thank all my other friends at Brown, my roommates Archita Agarwal and Aditi Singh, for their friendship and energy for social events and outings, and for the long conversations about Life, my friends from the Indian Community at Brown: Aarushi Kalra, Alka Pai, Chiraag Nataraj, Dhananjay Bhaskar, Geetika Nagpal, Harkirat Singh, Huma Siddiqui, Mrityunjay Kothari, Nupur Jain, Pinkesh Malhotra, Rashi Jain, Rhea George, Sagar Wadhwa, Srijan Neogi, Srinivas

Yadavalli, Vaishali Garga, thank you for your friendship, the celebrations, random chai and food days, and game nights. To the badminton group led by Roberto Zenit and Asimanshu Das, thank you for your energy and enthusiasm, and for all the fun times I had playing with all of you!

I would also like to thank my mentors and friends from various places and times in Life, scattered all over the globe, and a very crucial part of my support system. To all my teachers and mentors, for encouraging me to be better and go further, the space I have is not enough, but I am very thankful for each of you. I would like to thank Bhavana Kanukurthi and Nishanth Chandran in particular, for always being there to talk about the PhD experience and Life in general, and Vivek Seshadri for reaching out at a particularly bad time to tell me that everything would get better. And very importantly to all my friends who listened to my late-night and early-morning rants about everything, and were very understanding and patient with my graduate school lifestyle and general ineptness with time zones: Abhishek Shetty, Aishraj Dahal, Ambily Thomas, Akhila MR, Angelline Ann Koshy, Anjali Jayakumar, Arya N, Cressida Hamlet, Divya Mary Jose, Gopika S, Keerthana Kirupakaran, Koshy Thomas, Monica Yalla, Nidhasha Jose, Nithin V Nath, Pranav Ashok, Rohit Chatterjee, Santana Pegu, Sneha Mondal, Sruthi Sekar, Sudhina Abraham, Sushrut Karmalkar, Varsha Murali, and Veena Vijai, thank you!

To my therapist, who also listened to several rants and helped a lot with my ups and downs, thank you. To the owners of the excellent South Indian restaurant Jaya's Kitchen, which helped me maintain my sanity during the lockdown, thank you. To the K-pop band, BTS, who also helped a lot during the lockdown, thank you.

To conclude, I would like to thank my family. To my parents, Ann and Biju Carmel George, and my brother Mervin Carmel George, thank you for being understanding and supporting me in this journey. It has been a long way and I could not have done any of this without you, thank you for always believing in me and trusting me. I would also like to thank my extended family for always keeping me in their prayers and wanting the best for me. This dissertation would not have been possible without all these wonderful people, so once again, thank you very much, happy to be here.

Contents

Chapter 1

Introduction

1.1 Motivation

In the past decade or so, the pace of data generation has been growing exponentially. The total volume of data/information created, captured, copied, and consumed worldwide in 2010 was 2 zettabytes[1], and it is projected to grow to over 180 zettabytes in 2025 [41]. With wearable technologies and socially connected devices and applications, the nature of the data generated is also more fine-grained and personal. Data is also lucrative — with purposes ranging from commercial such as advertisements or promotions, to feedback and behavioral data to fine-tune products and services. Given the crucial importance and value of data, it is naturally prone to exploitation in many forms. Legitimate service providers could collect far more customer data than necessary to provide services, and criminal interests could steal data from any of the many services that collect them. The large data breaches that have made headlines over the past decade [59] are but a symptom of a deeper issue — the systems that collect and manipulate user data were not designed with long-term user privacy or data security in mind. As a society and scientific community there are several ways to deal with this ever-present and ever-growing challenge.

Governments can enact data privacy legislation with monetary consequences for non-compliance. This is already in motion around the world, ranging from the European Union's GDPR [32], to similar legislations in Brazil [64], California [22], China [83], India [42] and Thailand [35]. As a reaction, corporations are being forced to examine their privacy practices, which would most commonly manifest in our daily lives as a popup on every website asking for consent to use cookies. However, the solution is not as simple as that — data is a complicated subject, and there are technological challenges to compliance. For instance, the GDPR enshrines a "Right of Access" which allows any user to request their data from a system, and a "Right to be Forgotten" that allows the user to ask to be removed from that system. While new systems can be designed to incorporate these access requirements, older systems in operation today are built on legacy infrastructure, and cannot

[1] 1 zettabyte = 1 billion terabytes!

immediately support the identification and retrieval of such user data. It is therefore important to create both design principles and tools for new systems while simulateously retrofitting existing applications to be compliant.

While legislation and the threat of punishment can help speed up the adoption of privacy-friendly technologies, we still have to deal with the threat of insider access to data and large-scale data breaches. One solution is to use cryptography to secure data. However, the use of cryptography must not take away from the utility of that data. In particular, it is desirable to still support operations over the encrypted data. Current cryptographic solutions to this challenge use primitives such as fully-homomorphic encryption (FHE), structured encryption (STE) and oblivious RAM (ORAM). Each of these primitives present various trade-offs in expressivity, efficiency and security, are being widely studied in the research community.

1.2 Our Contributions

This thesis describes the following contributions to compliant and secure databases.

- **Retroactive GDPR Compliance.** We study the question of fulfilling a data access request from a data subject on a legacy database. In order to fulfill such a request, the data owner must be able to identify the data relevant to a data subject in the application database. This is not an easy task, especially when database schemas do not follow strict normal forms, and there are application-specific conventions for how data is related across tables. We posit that when the schema is not sufficiently informative, the application query logs can serve as a source of information about implicit relationships between the data. We present a mostly-automated tool, GDPRizer, that can take as input a schema, a query log and some coarse annotations from a database administrator (DBA) to extract data relevant to a data subject from a database. We conclude from our experiments with 3 commonly-used web applications that a fully automated solution is unlikely to exist, but our tool significantly reduces the manual effort required to facilitate a data access request.

- **Leakage Suppression for Structured Encryption.** Structured encryption schemes encrypt data structures in such a way that they can be privately queried. Like all sub-linear query time solutions, STE allows a persistent adversarial server to derive some information about the input data. This derived information is referred to as the *leakage* of the STE scheme. To address this, a line of work on *leakage suppression* was recently initiated that focuses on techniques to mitigate the leakage of STE schemes. A notable example is the query equality suppression framework (Kamara et al. *CRYPTO'18*) which does not reveal to the server if two queries to the data structure are equal. The framework takes as input a dynamic scheme (one that supports changing data) and produces a static scheme. It was left as an open question to design a solution that could yield dynamic constructions. We present a dynamic query equality suppression framework that transforms volume-hiding weakly-dynamic STE schemes that leak the query equality into new *fully-dynamic* constructions that do not. We then use our framework to design three new fully-dynamic STE schemes that are "almost" and fully zero-leakage which, under natural assumptions on the data and query distributions, are asymptotically more efficient than using black-box ORAM.

- **Practically Efficient Leakage Suppression.** The query equality leakage is a critical part of a structured encryption scheme's leakage profile. It has been used for cryptanalytic attacks [68], and it is correlated with other leakage patterns. The efficiency costs of hiding the query equality are quite high. For instance, our dynamic query equality suppression framework, when applied to volume-hiding multi-map schemes, is orders of magnitude more inefficient when compared to an optimal encrypted multi-map scheme. We design new techniques to suppress the query equality leakage, focussing on efficiency and practicality. We present a dictionary transform that suppresses the query equality leakage of any static dictionary data structure, and use it to design the first query equality suppressed dictionary scheme with optimal query complexity.

Chapter 2

Compliant Databases

In the recent years, governments around the world have introduced privacy legislation in order to give end users greater control over the use of their data. The most prominent of these has been the European Union's General Data Protection Regulation (GDPR) [32], which provides protection for the generators of personal data, or *data subjects* by codifying their rights such as the right to access and the right to erasure. Other laws such as the California Consumer Privacy Act (CCPA) [22] or in Virgina's Consumer Data Protection Act (VCDPA) [84, $\mathcal{S}$59.1-573] also contain similar provisions. Countries such as China [83], India [42], Brazil [64] and Thailand [35] have also enacted legislation that will allow their citizens to receive information about and control how their data is processed.

For the research community, privacy legislation and its impacts on the systems that are currently in place is a subject of great interest. Prior work has analyzed the impact of GDPR on storage systems [75], and in particular, database systems [77]. Their results indicate that strict compliance adversely affect the efficiency of existing database systems. Given the efficiency costs of compliance, the monetary costs to non-compliance [9], and the rising scale of data breaches, it is the need of the hour to come up with better solutions for systems that currently store user data. With this view, we study retrofitting GDPR compliance for legacy databases.

In order to satisfy data access or erasure requests, a database administrator (DBA) will have to identify all the data pertaining to the individual in an existing database. This can be a daunting task, especially when the databases themselves have not been designed with compliance considerations and therefore lack the necessary secondary indexes or metadata to look up data by the associated individual [77]. In this work, we investigate what it takes to provide mostly-automated tools that assist DBAs in GDPR-compliant data extraction for legacy databases. We find that a combination of techniques is needed to realize a tool that works for the databases of real-world applications, such as web applications, which may violate strict normal forms or encode data relationships in application-specific ways. Our tool, GDPRizer, relies on foreign keys from the database schema, query logs that identify implied relationships, and coarse-grained annotations provided by the DBA to extract an individual's data from a database. In a case study with three popular web applications, GDPRizer achieves 100% precision and 96–100% recall. GDPRizer saves work compared to hand-written queries, and while manual verification of its outputs is required, GDPRizer simplifies the process of privacy compliance for legacy databases.

2.1 Overview

Consider the classic TPC-H benchmark: its `supplier` and `customer` tables identify individuals, who are linked to orders, parts, and addresses stored in other tables. To satisfy a data access request on behalf of a customer, the DBA must (at least) query the tables connected to `customer` via either direct foreign keys, such as `orders`, or via indirect ones, such as `lineitem`. These queries require more than a simple transitive closure over foreign keys. Querying *all* tables connected via foreign keys might return more data than required (*e.g.*, revealing the personal details of a supplier to a customer); data might need post-processing to remove internal or private details; or data might be missing, as applications have imperfect foreign key specifications in their schema. These challenges show up in the databases of real-world web applications, such as Lobsters [57] and HotCRP [53]. The goal of our tool, GDPRizer, is to generate a set of queries that extract or delete an individuals's information in accordance with data access requests. GDPRizer must be practical for real applications' databases, whose schemas have evolved over time and are often messy.

Our work shows that high accuracy and complete data extraction hinges upon solving two challenges. First, GDPRizer must identify how data is related across tables, and ensure that a data access request returns rows from all relevant tables. Missing a relationship between tables results in missing rows in the output, can make the data access request fall short of legal compliance. Yet, the dependencies can be non-obvious, as an application might *e.g.*, encode a relationship using particular attribute values. For example, HotCRP indicates co-authorship on a paper via an entry in the `PaperConflict` table, with the `conflictType` column set to a special numeric constant indicating a "co-authorship" conflict type. GDPRizer should—with suitable inputs—understand this sort of dependency. Second, GDPRizer must avoid extracting too much data. Even though a table may store or reference data associated with an individual, returning that data might overreach. For example, an author's data access request in HotCRP should return reviews for their papers, but the `Review` table rows also contain the identity of the reviewer. To preserve reviewer anonymity, the rows returned to an author must have the reviewer ID erased.

GDPRizer relies on two key ideas to solve these challenges: a *relationship graph analysis* helps identify implicit dependencies across tables, and *schema-oriented customizations* limit the data extracted based on coarse-grained schema annotations provided by the DBA. Depending on the application, GDPRizer uses up to four types of input: (*i*) explicit foreign keys, if present; (*ii*) a log of runtime queries the application executes, which helps infer relationships between the columns; (*iii*) schema annotations that specify connections between tables that are connected by implicit data or those whose relationships cannot be inferred from queries or foreign keys; and (*iv*) schema annotations that specify what connections across tables to prune, and how to filter the extracted data. GDPRizer uses these inputs to traverse the database contents, extracting the information required to satisfy a data access request. It also provides warnings to the DBA if the extracted data might be incomplete.

2.1.1 Our Contributions

We implemented a prototype of GDPRizer and evaluated it with a synthetic schema (TPC-H) as well as three real applications: Lobsters [57], a Reddit-style news aggregator application that declares some foreign keys in its schema; HotCRP [53], a conference paper review application without any explicit foreign keys; and WordPress [8], a popular blogging platform with a non-traditional, performance-optimized schema. Our experiments show that GDPRizer achieves 62–100% precision (fraction of extracted records that are correct) and 66–100% recall (fraction of records extracted) without manual input for these applications. Manual customizations increase this to 100% precision and 96–100% recall.

In summary, we make the following key contributions:

1. We investigate what satisfying data access requests over legacy schemas entails, and what information beyond existing RDBMS abstractions (such as foreign keys) is needed.
2. We describe an algorithm to traverse a database schema and extract the information needed to satisfy a data access request.
3. We present GDPRizer, a tool that implements this algorithm and interactively guides a developer or DBA in generating the queries for data access requests.
4. We evaluate a GDPRizer prototype, demonstrating high accuracy on three real web applications' databases, and compare GDPRizer to custom GDPR compliance plugins for the WordPress blogging platform.

GDPRizer's automation is fundamentally limited by the fact that legacy databases' schemas may fail to reflect key application semantics. However, our work shows that it is possible to much reduce the manual labor required to satisfy data access requests.

2.2 Background and Related Work

High-profile data breaches and an increasing interest in consumer privacy regulation have led to a glut of new privacy laws in recent years. Failure to comply with these laws can lead to reputational damange, revenue loss, and substantial fines [9].

Data Access Requests. Most privacy laws grant individuals the right to request a copy of their data from those who store or process it, and to ask for its removal. In the GDPR, for example, these rights of a "data subject" (a natural person) are codified in Articles 15 ("Right to Access") and 17 ("Right to Erasure"). Other laws contain similar provisions, such as the CCPA's and VCDPA's "Right to Know". We refer to the power granted by these provisions as a *data access request*. A data access request requires the party controlling the data (a "data controller" in GDPR lingo) to identify all information they hold about the requester. Satisfying the request requires care, as the data returned must not violate the privacy of other individuals [32, Art. 15, $\mathcal{S}$4], so post-processing of the data identified is typically required.

Web Applications. Privacy laws have a broader scope than just web applications, but their impact is particularly serious for organizations that operate web services. Web applications often use a database backend coupled with stateless frontend logic. Popular frameworks like Ruby on Rails [4] and Django [7] use a relational database for storage by default. Yet, legacy web applications' schemas were developed without attention to data access requests, and the relational storage paradigm's strength—organizing records by type in tables—fundamentally mixes different individuals' data. This makes the task of identifying the data associated with an individual complicated and application-specific. Working out what information a data access request needs to return requires application developers or DBAs to navigate application-specific table and column names, as well as underspecified or implicit relationships that indicate records' association with individual users. This requires substantial manual, error-prone labor for many applications.

One might be tempted to believe that abstractions for relating entities across tables (such as foreign keys) could help. While this is true in theory—a data access request is, essentially, a recursive traversal of related entities from a starting entity in a table—real-world application schemas frequently fail to conform to third normal form (3NF) or lack the required keys. For example, we studied nine open-source web applications, ranging from chat plugins to social networks, blogging and conference review platforms[1] and found that only two of them specify foreign keys in their schema.

Compliance plugins. For some popular frameworks, application-specific, third-party privacy compliance plugins are available. For example, the WordPress blogging platform's plugin registry lists dozens of GDPR plugins related to cookie consent [82] or GDPR compliance [30, 65]. However, such plugins can have serious deficiencies (as we will show in $\mathcal{S}$2.7.5), but the DBA must blindly trust their correctness. For custom web applications or less popular frameworks, no such plugins are available.

Other approaches to compliance. Some researchers have proposed entirely new database systems [74, 54] or storage hardware [44] to achieve privacy compliance. While helpful for future deployments, these systems do not help legacy databases comply with data access requests. Other research has studied the performance costs of adding metadata structures (*e.g.*, secondary indexes) to existing databases to help satisfy data access [77, 75]. These techniques come with high overhead and without any automation. Odlaw [58] helps retrofit data access requests to legacy databases by building a graph of foreign key dependencies across tables and providing a graphical interface for DBAs to identify relevant data for a data subject. However, Odlaw assumes that the database schema contains explicit foreign keys, which many real applications lack.

[1]Lobsters [57], ghChat [10], Schnack [73], an Instagram clone [76], Socify [61], HotCRP [53], Commento [27], PrestaShop [72], and OpenCart [66].

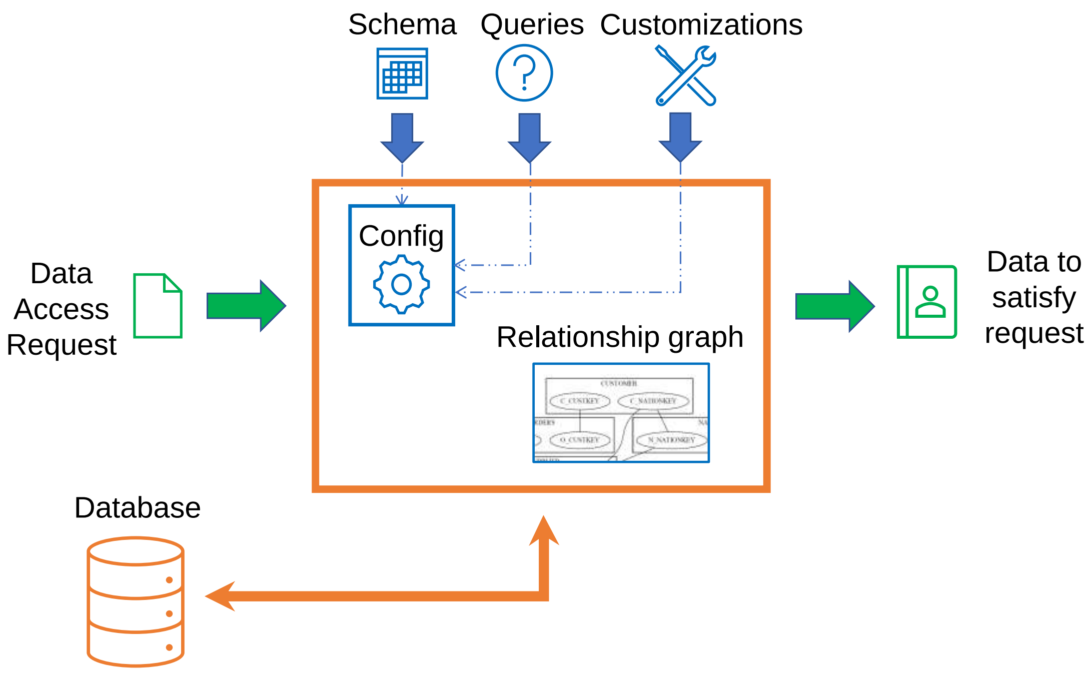


Figure 2.1: GDPRizer overview: schema, query log, and customization inputs configure the GDPRizer to handle data access requests for individual users. On request, GDPRizer queries the database and post-processes the data retrieved.

2.3 GDPRizer

We present GDPRizer, a tool that retrofits compliance with data access requests onto legacy databases. GDPRizer explores a trade-off between fully manual, application-specific scripts that must be written with great care and human effort, and automated—but likely imperfect—general-purpose solutions. Our goal is to investigate the degree of automation that a tool can provide for data access requests over legacy databases, while minimizing any manual inputs.

At a very high level, GDPRizer uses the database schema and a query log the application to extract semantic relationships between columns in the database. It then uses these relationships and manual customizations to generate a configuration that helps the tool satisfy data access requests for an individual by querying the database (Figure 2.1).

2.3.1 Automated Relationship Detection

A well-formed database schema in 3NF will indicate semantic relationships between tables via foreign keys: a foreign key indicates that the source table references objects in the destination table. This information is crucial to serve a data access request. Such a request starts with a data subject ID (DS ID), which typically corresponds to a row in some table—*e.g.*, `customer` or `supplier` in TPC-H, since both customers and suppliers are data subjects under laws like the GDPR. A foreign key into

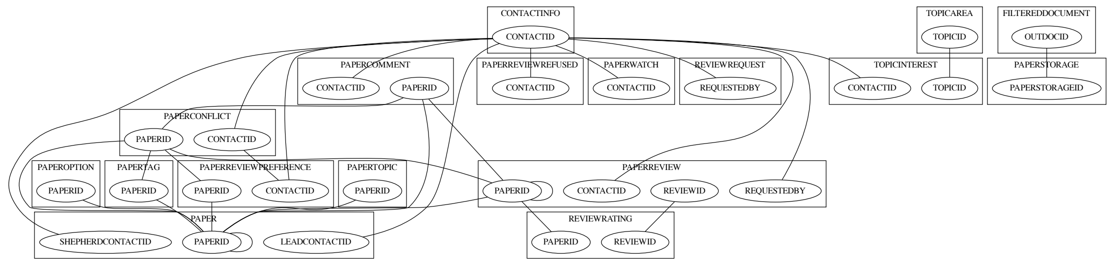


Figure 2.2: Relationship graph of HotCRP [53]. Boxes correspond to tables, round vertices to columns, and edges to foreign keys in the schema or joins observed in the queries. Since HotCRP lacks foreign keys, this graph is identical to HotCRP's join graph ($\mathcal{S}$2.4).

the table that contains data subjects indicates that records in another table are associated with the data subjects. A transitive foreign key (often) indicates the same about an object that is two or more steps away from a data subject table. When present, GDPRizer therefore uses foreign keys to detect data related to an individual data subject. However, practical application database schemas often lack FKs. Any real-world GDPR compliance solution therefore has to tackle challenges such as lack of referential integrity and implicit or conditional relationships between columns.

If the database schema lacks sufficient information about foreign keys, GDPRizer uses alternate sources of information to identify relationships between tables. One such source is the set of queries made to the database at application runtime. The idea behind using application queries is that a runtime join between two tables frequently implies a foreign key relationship between these tables, particularly if the destination column is a table's primary key column. For example, TPC-H joins `customer.c_custkey` with `orders.o_custkey` for a query that summarizes a customer's order information, matching the foreign key constraint between the two columns. While joins on non-foreign key columns are possible, they are fairly rare in practical web applications. Keeping this in mind, GDPRizer uses application query logs to supplement the relationship information provided by explicit foreign keys in the schema. Query logs are easy for DBAs to obtain (*e.g.*, by sampling some fraction of runtime queries, or by enabling query logging).

Relationship Graph. Together, the foreign keys (if present) and the joins observed in a query log constitute GDPRizer's *relationship graph*, a raw, unprocessed set of known relationships between columns across tables. Each column in the database is represented by a vertex in this graph. The relationships between a pair of columns—foreign key constraints or joins—are represented by an edge between the columns. A table is represented by multiple, grouped vertices in the relationship graph. Figure 2.2 shows GDPRizer's relationship graph for HotCRP.

Data Extraction. When GDPRizer receives a data access request, it traverses the relationship graph to extract data relevant to the data subject. The traversal begins at the key column that contains the data subject's primary identifier. GDPRizer then proceeds to extract records directly or indirectly connected to this key column. In other words, GDPRizer explores the transitive closure of all the connected tables in the graph to extract the relevant data.

2.3.2 Manual Customizations

However, GDPRizer (and likely any general-purpose tool) needs additional input to identify application-specific semantic structure in the database. When the generated graph fails to express some aspects of the application semantics, GDPRizer allows for a domain expert, such as the application developer or DBA, to intervene and customize either the relationship graph itself, or the data output after traversal. We aim to minimize this intervention in GDPRizer, but our case studies show that a modicum of manual input is often required. GDPRizer's customizations fall into four categories:

(1) Edge pruning. The relationship graph for a practical application sometimes contains relationships that are not relevant to a data access request. This can happen because the application joined columns that are not semantically related, or because there exists a foreign key that connects internal application data rather than user data. For GDPRizer to ignore these irrelevant relationships when it extracts the data, the relationship graph itself needs customizing. To achieve this, GDPRizer supports edge pruning annotations, which allow the DBA or developer to indicate that all edges (*i.e.*, relationships) incident on a particular column should be ignored.

(2) Adding missing edges. On the other hand, even after using information from both the schema and the runtime queries, the relationship graph may still be missing relationships. For example, the relationship graph for WordPress has no edge between the table with user information and the table that holds comments, even though it is clear from the application semantics that they contain related columns. When tables appear to be disconnected in the relationship graph, GDPRizer will prompt the DBA to manually "connect" the disjoint components. GDPRizer uses a datatype-matching heuristic to suggest edges (column relationships) that might have semantic significance. The DBA considers the list of suggestions and adds the missing relationships.

(3) Data-dependent and conditional relationships. The most complex customization may be necessary if the application has implicit or conditional relationships that cannot be expressed in terms of the existing columns. This happens *e.g.*, if a second column indicates the "type" (semantic meaning) of a foreign key that is present; for example, this occurs with paper conflict types in HotCRP: co-author conflicts have different semantics for data access requests than other conflicts. To address this, the DBA provides an input that transforms the data such that the relationship is direct and explicit. In particular, GDPRizer supports the creation of views that contain rows from a source table only if a predicate over the rows holds true. These views become part of the relationship graph, replacing other tables and edges, and GDPRizer uses them for data extraction.

(4) Output filtering. Once GDPRizer has completed the relationship graph traversal and queried the database, it may be necessary to filter the resulting records to remove personal information of other individuals (*e.g.*, reviewer details on HotCRP) or unrelated data (*e.g.*, internal supplier information in TPC-H). The DBA specifies columns to filter from the output by annotating the schema, and GDPRizer removes or rewrites these columns.

GDPRizer only needs to be configured once and the manual customizations are one-off for a given database. Once a relationship graph is setup, GDPRizer saves the customizations as a configuration for all future data access requests, essentially creating an application-specific GDPR compliance tool with less effort than would have been necessary to write the queries manually.

2.4 Relationship Graph

When GDPRizer receives a data access request, it identifies all data relevant to the individual making the request (the "data subject"). GDPRizer assumes that the data subject is uniquely described by a row in a *primary table*. This is common: many applications have a users table with their users's details, or represent individuals as rows in tables associated with their role (*e.g.*, TPC-H's `customer` and `supplier`, or Lobsters's invited users in `invitations` and registered users in `users`). Other entities in the database refer to these primary table rows, establishing a *relationship*. For example, TPC-H has rows in the `order` table refer to customers by their unique key in the `customer` table (a foreign key constraint). GDPRizer can assume that these related rows might also be relevant to the customer, so GDPRizer must identify and use the relationship between `customer.c_custkey` and `order.o_custkey` to eventually be able to extract the data. GDPRizer represents these relationships as edges in the relationship graph. The relationship graph combines relationships specified explicitly in the database schema with inferred relationships determined from application execution.

Foreign keys. The most reliable source of relationships in a database is the database schema. If the schema is well-formed and in 3NF, it contains all the foreign key constraints in the database, *i.e.*, all by-key relationships between columns across tables. In TPC-H, for example, there exists a foreign-key constraint between `customer.c_custkey` and `order.o_custkey`. These constraints can be conceptualized as a graph whose vertices are columns, and whose edges are foreign-key constraints. We refer to this graph as the *foreign-key graph*, and it forms one of the bases of the relationship graph. However, as previously discussed, many real-world database schemas fail to conform to strict 3NF, so the foreign key graph alone is insufficient. GDPRizer must be capable of inferring relationship information that is missing in the database schema.

Inferring relationships. One possible heuristic for inferring column relationships might be to compare the column names and datatypes. However, there are no standard naming conventions, and even though foreign keys sometimes follow specific naming schemes, it is hard for GDPRizer to determine the application-specific naming convention in use. Another approach, which requires less human effort, is to use the application semantics expressed in runtime queries to infer relationships within the database. For example, if the application joins two columns in the database at runtime, GDPRizer can infer that the columns are likely related, as the application assumes that they share data values. GDPRizer's approach is therefore to use query log of the application, provided by the developer or DBA, to identify columns that are joined at runtime. Such a log is easy to obtain, *e.g.*, by enabling query logging in the database, or by instrumenting the application's DB access code. (Note that the log does not need to be complete—a sample is often sufficient.) These resulting inferred relationships can also be represented as a graph, and we refer to this graph as a *join graph*.

Finally, GDPRizer combines the foreign-key graph with the join graph to create the relationship graph.. Figure 2.2 shows the relationship graph of the HotCRP conference review application. Since

HotCRP lacks foreign key constraints in its schema, this relationship graph is identical to the join graph.

2.5 Graph Traversal

Given the relationship graph, GDPRizer uses it to retrieve a data subject's records from the database. This requires GDPRizer to traverse the relationship graph, starting with a row in the primary table, and to generate meaningful queries as the traversal proceeeds.

A naïve graph traversal, which traverses all the edges, might extract too much or too little data:

1. if several paths from the primary table to another table exist, each of them could lead to a different set of extracted rows, which might be too much data; and
2. since there are (usually) no edges between the columns of the same table, the graph consists of many disconnected components, as shown by the colors in Figure 2.3. Then any edge-based traversal that begins in one component will fail to extract data from the unreachable components.

GDPRizer addresses these challenges with heuristics based on *proximity* and *implied relationships*, as explained in the following.

To avoid overextraction and duplicate data, GDPRizer only visits each column once. When multiple paths to a column are available, GDPRizer picks the shortest one. Prioritizing in this way makes sense because, intuitively, columns that are "closer" to the starting column, *i.e.*, the primary key of the primary table in the relationship graph, are more relevant to the data subject. Therefore, GDPRizer traverses the graph in breadth-first manner than the depth-first manner from the starting column. To address the second challenge, GDPRizer traverses the graph via two types of relationships in the graph (Figure 2.3):

- relationship edges, based on foreign keys or application joins, such as the edge between columns A and B; and
- *implied relationships* between the columns of the same table, *e.g.*, columns B and C.

GDPRizer uses the relationship edges for data extraction, and the implied relationships to connect the components of the graph. Figure 2.3 shows a high-level overview of the graph traversal.

Relationship edges. Whenever possible, GDPRizer follows the graph's relationship edges. For example, in Figure 2.3, column A is the primary key column of the primary table and therefore the graph traversal's starting column. It follows that the columns accessible by a relationship edge from A, like B and E, are directly related to the data subject since they were joined in the query set or have a foreign key. For columns not directly linked to the primary key of the primary table, GDPRizer uses their distance from the starting column as a metric to decide which paths to explore. We call a column's distance from the starting column its *proximity*. Computing the proximity helps GDPRizer traverse columns nearer the starting column before ones that are further away, and ensures that it prefers shorter paths over longer ones. The proximity also naturally imposes a direction on the traversal of the edge between any two columns—the traversal proceed from the column closer to the starting column to the column that is further away. GDPRizer uses a standard breadth-first traversal (BFT) from the starting column to compute the proximity of columns in that component of the graph. Starting the BFT from A, the proximity of columns B and E is 1, the proximity of F is 2 and that of G is 3. Notice that after exploring these columns, there are no more relationship edges to traverse, and the BFT cannot continue.

Implied relationships and inferred proximity. So far, GDPRizer has visited only one component of the graph, containing the columns A, B, E, F and G. To continue the traversal through the remaining disconnected components, GDPRizer chooses a *secondary* starting column for each of the components, and repeats the distance computation from this secondary starting column.

To choose the starting column in a component, GDPRizer relies on another proximity-based heuristic. Having already computed the proximity for the columns in the connected component that contains the starting column, GDPRizer now considers the untraversed *siblings* of these traversed columns. (Two columns are siblings if they belong to the same table.) This set of untraversed siblings must be part of disconnected components, as they would have been traversed already if they were connected to the starting column component. GDPRizer uses implied relationships between siblings to infer the proximity of an untraversed sibling column: it sets the proximity of an untraversed column to the minimum proximity over all its siblings $+1$. In Figure 2.3, columns C, D, J are siblings of B, E, G respectively. Then their proximities are set as shown due to the implied relationship between siblings. Setting the proximity in this manner is equivalent to traversing an (implied) relationship edge from the sibling with minimum proximity. Note that GDPRizer *only* uses implied relationships if no relationship edges to the column existed, *i.e.*, if the column is untraversed. After this augmentation, GDPRizer continues with the proximity rule used for the first component, and picks the column with the minimum proximity as the *secondary* starting column. GDPRizer then repeats a breadth-first traversal using the relationship edges in that component. The secondary starting columns in the figure are then C, D, J and the respective BFTs are indicated on the graph. This process of using the relationship edges and the implied relationships alternately continues until GDPRizer has traversed all the columns in the graph, or no more viable sibling columns exist.

Data extraction. GDPRizer's data extraction proceeds alongside the graph traversal. GDPRizer starts by issuing a query for the all records associated with the data subject identifier (DS ID) in the primary table, and then associates the value of DS ID with the starting column. In subsequent steps, for each relationship edge between columns A and B, traversed as $A \rightarrow B$, column A already has some associated value that was queried in the previous step of the traversal. GDPRizer issues a SQL query for all the records with this value in column B, as the relationship edge requires the values of A and B to be identical. This process repeats for all relationship edges. When GDPRizer uses an implied relationship, say from B to C, it already knows the value for B (which the traversal reached via the relationship edge from A to B). GDPRizer queries the records with that value in B and obtains the matching values in sibling column C. These associated values then initiate the traversal (and thus, data extraction) in that component of the graph. Finally, GDPRizer combines the output of all the SQL to produce the data associated with the data subject.

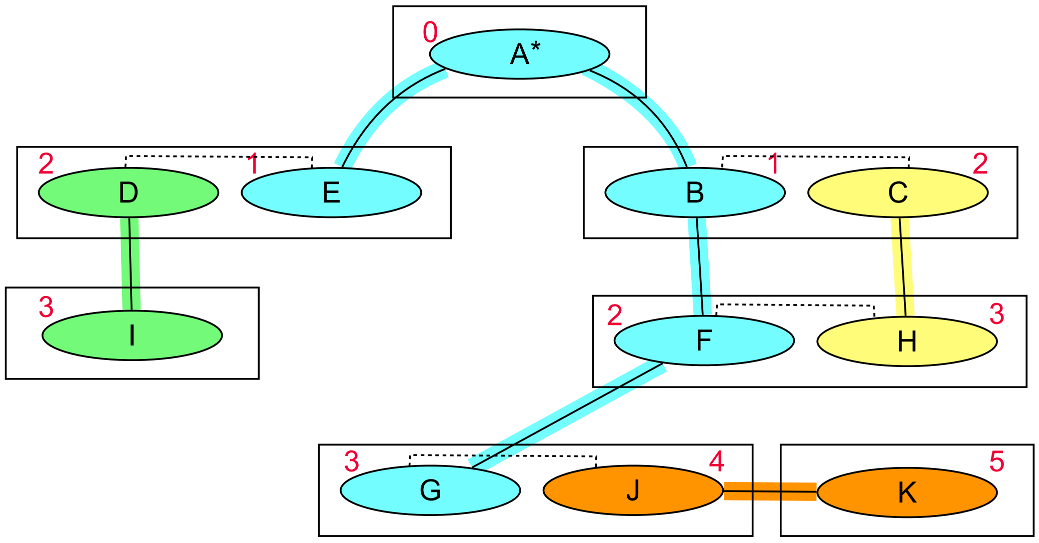


Figure 2.3: An example graph traversal. Column A is the primary column of the primary table (*). Each column is annotated with its proximity. The first BFT is marked in blue. Subsequent BFTs after using implied relationships are marked in green, orange and yellow.

2.6 Customizing the Graph Traversal

In practice, application databases' structure may have semantic properties that the relationship graph fails to capture. GDPRizer offers manual customization options to modify the relationship graph and the extracted data. These options are semi-automated: the developer or DBA adds customizations either in response to prompts from GDPRizer (*e.g.*, if there are disconnected components of the relationship graph that GDPRizer's traversal cannot reach) or after inspecting the data returned by GDPRizer's extraction.

2.6.1 Graph customization

GDPRizer supports three graph customizations: (*i*) edge removal or pruning; (*ii*) edge addition; and (*iii*) vertex addition.

Edge Pruning. When a database schema contains columns that GDPRizer should not use to extract data, a DBA can annotate the columns to avoid traversal (and further data extraction) via these columns. For instance, in the HotCRP database, the conflicts on a paper link rows in the `Paper` table to conflicted individuals' records via a relationship to `PaperConflict`, which in turn has a relationship with `ContactInfo`. GDPRizer should not extract information about the conflicted individuals. To avoid this, the DBA might prune the `contactId` column in the `PaperConflict` table, removing all edges incident on it. In our experiments, edge pruning was the most commonly needed customization for application databases.

Edge Addition. Edge addition becomes necessary when the relationship graph lacks edges to some tables in the schema. This happens if the tables are neither related via any foreign keys, nor did they ever get joined by application queries. In such cases, GDPRizer will prompt the DBA to "connect" these tables to the rest of the relationship graph. The prompt provides a list of plausible edges based on matching column datatypes and primary key constraints.

Vertex Addition. Vertex addition is the most complex customization GDPRizer supports. It is required when the database contains conditional or implicit relationships. These relationships are computed programmatically, rather than being expressed as simple foreign keys or joins. One example of this is how HotCRP represents the co-author relationship on a paper. A co-author is specified using a row in the `PaperConflict` table, with the `conflictType` column set to a specific bitfield value. Based on this relationship, GDPRizer must extract the data for papers that a user has co-authored. Rows with other conflict types (*e.g.*, institutional, advisor-advisee) also have relationships with the `Paper` and `ContactInfo` tables, but GDPRizer should refrain from extracting their data.

To support this, GDPRizer allows a DBA to add a *virtual column* to a database table, effectively defining a view. The virtual column transforms the implicit or conditional constraint into an explicit column *i.e.*, a vertex in the relationship graph. The virtual column is derived from a source column, and GDPRizer copies all edges of the source column to the virtual column. In the case of HotCRP, the DBA provides GDPRizer with the query to create a view of the `Paper` table that contains a row for each co-author user ID, and bases co-author ID column on the source column `leadContactId`. Hence, GDPRizer copies all the relationships of the `leadContactId` to the virtual co-authors column. The graph traversal then uses the view in place of the `Paper` table.

2.6.2 Output customization

GDPRizer must also take care to avoid returning internal information or the information of other data subjects as part of the data it extracts for a data access request. For this purpose, and to reduce unnecessary output, GDPRizer supports post-processing of the data extracted.

Filtering. After the data is extracted, GDPRizer allows the DBA to filter out any unnecessary columns from the output using filtering annotations. This is expressed as a list of columns to drop or rewrite in the output. In order to reduce manual input, GDPRizer automates filtering for one specific kind of type: a *mapping table*, which is a table that consists entirely of relationship columns (*i.e.*, all columns are foreign keys). For example, in HotCRP, the `PaperTopic` table maps paper ids to the topic ids that they concern, but contains no other information. Since GDPRizer will return records from both the `Paper` and the `TopicArea` tables, it is unnecessary to return the mapping table records as well. Consequently, GDPRizer automatically drops such mapping tables from the output.

Roles. Finally, GDPRizer supports *roles*, which allow applying different customizations by data subject type. In TPC-H, for instance, `customer` rows or `supplier` rows can represent data subjects who can issue data access requests, but GDPRizer should return different information depending on whether the request originated with a customer or a supplier. The customer's primary table is the `customer` table and the supplier's primary table is the `supplier` table. GDPRizer's graph traversal must account for these varied roles in the database, and avoid extracting more data than is necessary for each role. Therefore, GDPRizer allows the DBA to specify multiple roles, each associated with a different primary table and set of per-role customizations, which specify a custom traversal for each role. Since roles are application-specific, GDPRizer requires manual input to specify them and their customizations.

2.7 Evaluation

We evaluated GDPRizer with a synthetic benchmark (TPC-H) and with three real web applications (Lobsters [57], HotCRP [53], and WordPress [8]). Our evaluation seeks to answer these questions:

1. Does GDPRizer correctly handle data access requests over real web applications' databases? ($\mathcal{S}$2.7.2)
2. How many manual inputs do applications require? ($\mathcal{S}$2.7.3)
3. What impact do manual customizations have on GDPRizer's correctness, and when are they required? ($\mathcal{S}$2.7.4)
4. How does GDPRizer compare to the third-party GDPR compliance plugins available for some applications? ($\mathcal{S}$2.7.5)

Setup. We prototyped GDPRizer in approximately $1,200$ lines of Python. Our implementaiton uses moz-sql-parser [2] to parse SQL queries. Since this parser can only handle some subset of SQL-92 queries, GDPRizer skips over the queries that moz-sql-parser cannot handle. This only affects a small number of queries.

Accuracy measurements. We measure GDPRizer's accuracy for four applications: TPC-H, HotCRP, Lobsters, and Wordpress. None of these applications currently have native support for data access requests. Hence, they lack a *ground truth* on the data that should be returned for each data subject. We used our knowledge of the applications to determine the data that we believe a data access request should return. We studied the application's schema and for each table in its database, we manually wrote a set of "ground truth" queries. For Wordpress, which has publicly-available GDPR plugins, we compare our results to the data extracted by these plugins.

We then compare the rows that GDPRizer extracts with the rows included in these ground truths. To measure GDPRizer's accuracy, we compute *precision* and *recall* relative to the ground truth. We denote precision by P, recall by R and define them as follows:

$$P = \frac{\mathsf{tp}}{\mathsf{tp} + \mathsf{fp}}, \qquad R = \frac{\mathsf{tp}}{\mathsf{tp} + \mathsf{fn}},$$

where tp, fp, fn, respectively are the number of true positives, false positives and false negatives in GDPRizer's results. We calculate both metrics for each table. Table-level analysis helps us measure the performance of GDPRizer on different parts of the database and uncover any specific shortcomings. For each application, we run data access requests for all data subjects in the database and report accuracy results averaged over all data subjects.

Inflated averages. Before we discuss our results, we consider a problem that could skew the precision and recall metrics when a majority of data subjects have no data in certain tables.

Suppose that GDPRizer's extraction should avoid querying a table T for any data subject with a particular role, *e.g.*, the `customer` table for suppliers. If GDPRizer queries T for data subjects with no data in T, the database will return no data, and hence GDPRizer would appear to have 100% precision. For such data subjects, GDPRizer did the right thing—extracting no data—but for the wrong reasons. Specifically, in this example, GDPRizer extracts no data because T had no matching records, not because GDPRizer did not query it. Averaging over a large number of such data subjects, gives us an inflated averaged precision value, which is an issue. A similar issue can occur with recall. Therefore, to ensure our results meaningfully report GDPRizer's correctness, we exclude the set of data subjects with no data in T from our reported averages.

2.7.1 Applications

We evaluate our prototype with the TPC-H benchmark and three applications: HotCRP, Lobsters and Wordpress. While TPC-H serves as a sanity check, the three applications evaluate GDPRizer's real-world performance. In this section, we describe the setup of each application: how we populate the application database, how we collect its queries, and how we establish the ground truth.

TPC-H. The TPC-H benchmark models data and events between suppliers and customers in a warehousing system [5]. We generated 100MB of data, consisting of 150 customers and 10 suppliers. TPC-H's 22 SQL queries contain a total of 62 joins, and the schema has 13 foreign-key contraints [6, Fig. 2]. GDPRizer can use the foreign-key relationships, but also successfully extracts them from the queries.

We run experiments for the customer and supplier roles. For a customer, the `customer` table is the primary table and its primary key, `c_custkey`, the primary column. The ground truth consists of queries for each table except the `supplier` and `partsupp` tables. We excluded these tables because `supplier` contains supplier's private information (*e.g.*, account balance), while the `partsupp` table contains sales information of a supplier, such as supply cost and available quantity. For suppliers, `supplier` is the primary table and its primary key, `s_suppkey`, the primary column. From the ground truth, we exclude data from the `customer`, `lineitem`, and `orders` table, since customer details and order processing details of the warehouse do not concern suppliers.

Lobsters. Lobsters is a link aggregator page, similar to HackerNews [1] or Reddit [3]. Its database has 25 tables which describe user posts, comments, votes, taggings, moderations, filters, etc. Lobsters comes with a sample dataset for 44 users, and we use this to populate the database. We created three additional users and logged the queries generated during interactions with the application. We attempted to exercise all possible actions in Lobsters and collected $3,960$ queries. We extracted 41 edges from foreign-key constraints in the Lobsters schema and 2 additional edges from joins in the queries.

For Lobsters, the `users` table is the primary table and its `id` column the primary column. We included 23 queries in the ground truth, covering 17 tables. We excluded 8 tables that contain Lobsters's Ruby-on-Rails metadata rather than user data (*e.g.*, `ar_internal_metadata`, `keystores` and, `schema_migrations`).

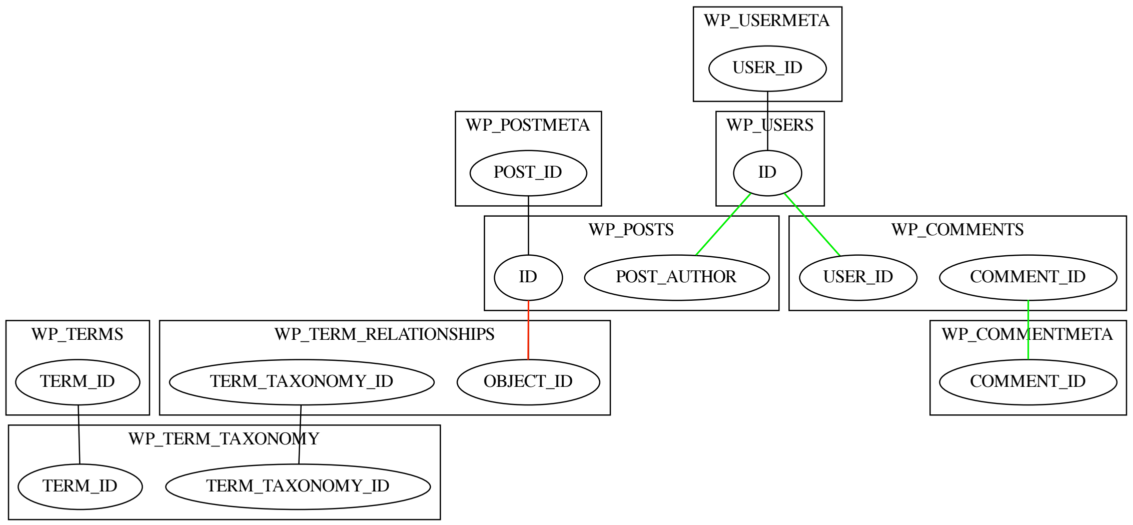


Figure 2.4: Relationship graph of Wordpress. For visual clarity we only show the columns which have edges on them. Edges in green are those added manually and edges in red are those that are pruned from the graph. Notice that pre-customizations, the graph contains four disconnected components.

HotCRP. HotCRP is a conference peer review application [53] and its database has 24 tables. In our experiments, we use an anonymized HotCRP dataset from an actual conference. The dataset contains data of $1,273$ authors and 507 papers. We also use a sample of 251 queries. Since HotCRP's schema lacks foreign-key constraints, GDPRizer uses the queries to build the relationship graph (Figure 2.2). GDPRizer extracts 30 edges in total. We set `ContactInfo` as the primary table and its primary key, `contactId`, as the primary column. For the ground truth, we wrote 17 queries that extract data from 12 tables and exclude data from 12. Tables excluded are either application management tables such as `Settings`, `MailLog`, `FilteredDocument`, and `Mimetype` or mapping tables such as `PaperTopic`.

Wordpress. WordPress is a popular blogging platform and content management system [8]. A key feature of WordPress is its plugin architecture, which allows users to add additional functionality to their WordPress installation. For example, the WooCommerce [16] plugin adds e-commerce functionality, allowing users to host online shops. The base WordPress database has 12 tables which describe users, comments, posts, terms and their taxanomies, and other metadata. The WooCommerce plugin adds 27 new tables to the base database to support online shops. We investigate how GDPRizer performs on the base installation of WordPress and how it adapts to the updated database when WooCommerce is added. We generated sample data for 46 users using another WordPress plugin, FakerPress [40]. For WooCommerce, we manually generated sample data. As with Lobsters, we logged the queries generated by users during their interactions with the WordPress site and attempted to replicate all possible actions. Overall, we collected $9,301$ queries.

As the WordPress schema does not specify any explicit foreign keys, the relationship graph includes 5 edges identified from joins in the queries (Figure 2.4). We treat `wp_users` as the pri-

	# edges in fk-graph	# edges in join-graph	# common edges	Pre-cust. precision avg	Pre-cust. recall avg	Post-cust. precision avg	Post-cust. recall avg
TPC-H (cust)	13	13	13	0.68	1	1	1
TPC-H (supp)	Same as for customer			0.62	1	1	1
Lobsters	41	17	15	0.69	0.99	1	1
HotCRP	0	30	0	0.88	0.76	1	0.96
Wordpress (base)	0	5	0	1	0.66	1	1
Wordpress (plugins)	0	11	0	1	0.69	1	1

Figure 2.5: Relationship graph statistics and high-level results for GDPRizer's performance by application: GDPRizer achieves 62% or higher precision and 66% or higher recall without any manual input, and 100% precision and recall with manual input, except for HotCRP. Values reported here averaged over per-table values, which in turn are averages over individual data subjects.

mary table and its `id` column the primary column. For the base installation, we included 6 queries over 6 tables in the ground truth. As usual, we excluded the tables which contain data not directly related to users. There are 6 such tables, `wp_links`, `wp_terms`, `wp_termmeta`, `wp_options`, `wp_term_relationships` and `wp_term_taxonomy`. For the WooCommerce plugin's 27 new tables, we included 9 new queries in our ground truth, covering 9 of the new tables.

2.7.2 High-level Accuracy Results

We first consider the accuracy of GDPRizer's results on a per-application basis, and both with and without manual customizations. Each application contains many data subjects and tables, and we aggregate the per-table and per-data subject results by averaging. A good result for GDPRizer would show high precision and recall both with and without manual customization.

Figure 2.5 shows the results. GDPRizer's relationship graphs have 5–43 edges depending on the application. Except for the synthetic TPC-H benchmark, all applications rely on the join graph to populate their relationship graph. Only Lobsters has explicit foreign keys, of which the join graph captures 15. But 26 foreign-key constraints are not reflected by the Lobsters join graph, which illustrates that it is helpful for GDPRizer to use explicit foreign keys when available. Without any manual customizations to the graph, GDPRizer achieves at least 62% precision and 66% recall for all the applications. Individual metrics reach 100% for some applications, but no application sees both perfect precision and recall. Since privacy compliance is all-or-nothing, this would be insufficient in a practical setting. The final columns in Figure 2.5 illustrate that, with some inputs from the DBA, GDPRizer achieves perfect precision and recall for all applications except HotCRP, whose recall is 96%. The reason for this imperfect recall is that there are two paths into the `TopicArea` table in HotCRP's relationship graph. GDPRizer ignores the longer path, and hence under-extracts from `TopicArea`. However, `TopicArea` only contains public information (the paper topic categories), so a DBA could plausibly just annotate the table to indicate that GDPRizer should always return it in its entirety.

	# cols added	# cols filtered (total cols in schema)	# edges added	# edge pruning annotations (total edges in rel. graph)
TPC-H (cust)	0	0 (61)	0	4 (13)
TPC-H (supp)	0	0 (61)	0	7 (13)
Lobsters	0	0 (192)	1	19 (43)
HotCRP	1	18 (200)	2	11 (30)
Wordpress (base)	0	0 (94)	3	1 (5)
Wordpress (w/ plugins)	0	0 (288)	9	2 (11)

Figure 2.6: Good GDPRizer performance requires annotating a subset of columns. Edge pruning annotations are most common.

2.7.3 Manual Customizations Needed

While minimal input from the DBA is desirable, it is often necessary, our experiments have shown that some manual input is usually necessary. We now investigate the inputs required for the four applications. We believe that, given the relationship graph and suggestions from GDPRizer, any DBA with some background knowledge of the database should be able to identify these customizations. Figure 2.6 summarizes the customizations we make for each application.

Columns added. Adding "virtual" columns helps GDPRizer deal with data-dependent or conditional relationship, but is rarely required. The only application that requires this customization is HotCRP. HotCRP does not store the contact IDs of paper authors in the `Paper` table, since papers can have an arbitrary number of co-authors. The exception to this is the lead author, whose ID HotCRP stores in the `leadContactId` column of `Paper`. Rather than storing co-authors as separate rows in an authors table with one row per (paperID, authorID) combination, HotCRP combines this information with information about conflicts of interest for a paper. (This makes sense because each co-author naturally has a conflict of interest for reviewing their own paper.) Specifically, HotCRP has a `PaperConflict` table, whose rows represent co-authorship if the `conflictType` column takes specific values. To handle this, the DBA defines a virtual column in GDPRizer. This adds a new column, named `v_author` to the `Paper` table, which is a direct foreign key into the `ContactInfo` table (*i.e.*, `v_author` stores the contact IDs of the co-authors of a paper). This results in a new relationship graph vertex, `v_author`, which captures the author relationship and lets GDPRizer traverse the relationship graph in its usual way. Since `v_author` behaves like the `Paper.leadContactId` column, we adopt the edges of `Paper.leadContactId` to connect `v_author` to the rest of the graph. Finally, we populate it with co-authorship information from the `PaperConflict` table.

Edges pruned. Edge pruning is the most common customization, and the one that affects the most tables and columns. Ideally, the reasons for pruning should be easily evident to a DBA. In our example applications, pruning annotations fall into three categories.

Tables without user relevance. The first kind of pruning prevents GDPRizer from extracting application data that is irrelevant to users. A DBA identifies all such internal tables and annotates all columns in these tables with incident edges for pruning.[2] For example, in Lobsters, we annotate three columns in management tables; in HotCRP, we annotate four columns; and in the base WordPress installation, we prune one column, while with the WooCommerce plugin, we prune another column that represents global product permissions. Note that these prunings are unnecessary if the DBA decides to return data from these tables—in that case, they can leave the edges as is. In our experiments, we considered data in management tables irrelevant and removed their data from the ground truth, so it was natural to prune the edges into these tables.

Other data subjects' information. The second kind of edges pruned are edges into tables that contain personal information of other data subjects. Privacy laws require avoiding to return personal information of other individuals when satisfying a data access request, meaning that such information cannot be returned by GDPRizer. For example, in TPC-H, if the data subject is a customer, we prune all the edges into tables with supplier data, whereas for a supplier-role data subject, we prune all the edges into tables with customer data. For a customer, we prune edges into the the `supplier` and the `partsupp` tables, whereas for a customer, we prune edges into the `customer`, `lineitem` and `order` tables (see $\mathcal{S}$2.6).

Avoiding over-extraction. The third kind of edge prunings is more involved and requires the DBA to identify individual edges which might extract incorrect data. For example, in Lobsters, we prune the `story_id` column from six tables to stop GDPRizer from retrieving information about stories that a data subject might have "acted upon" but does not "own". For instance, we prune `votes.story_id` to avoid information on stories that a data subject voted on but has not written (although GDPRizer still extracts the vote records themselves). Another example of this kind of pruning occurs in HotCRP, where we prune `Paper.shepherdContactId` to stop GDPRizer from extracting paper details of papers which a data subject shepherded, but did not write. We prune six columns of this type in HotCRP. The pruning annotations of this third type require a DBA to carefully consider and inspect GDPRizer's output, and to potentially iterate a few times until the output is correct.

Edge additions. Edge additions provide GDPRizer with missing relationships in the relationship graph. This customization is crucial if an application's schema lacks foreign keys and the join graph provides incomplete information. This may occur, *e.g.*, because application developers optimize for faster queries by computing joins in application code; because they try to improve scalability by avoiding joins that require taking locks on multiple tables; or because the application simply never needs the joined data. In our applications, between one and nine edges not captured in the join graph needed adding. In Lobsters, we add an edge between `users.id` and `messages.author_user_id` to capture the private messages that a data subject wrote. Lobsters lacks this edge because its schema avoids having two foreign keys between the same pair of tables (`users` and `messages`), but the relationship nevertheless exists. In HotCRP, we add an edge between `ContactInfo.contactId` and `ReviewRating.contactId`, to capture the review ratings a user contributed. We also add an edge

[2] A table-granularity annotation would make this easier, and we plan to add such an annotation to GDPRizer in the future.

between `PaperTopic.topicId` and `TopicArea.topicId` to capture papers' actual topics.

Wordpress has the least well-connected join graph, likely due to join-avoiding query optimizations. The relationship graph for the base Wordpress setup contains four disconnected components (Figure 2.4). To connect these components, we manually add the following three edges to the graph:

- `wp_users.ID` $\leftrightarrow$ `wp_comments.user_id`
- `wp_users.ID` $\leftrightarrow$ `wp_posts.post_author`
- `wp_comments.ID` $\leftrightarrow$ `wp_commentmeta.comment_id`

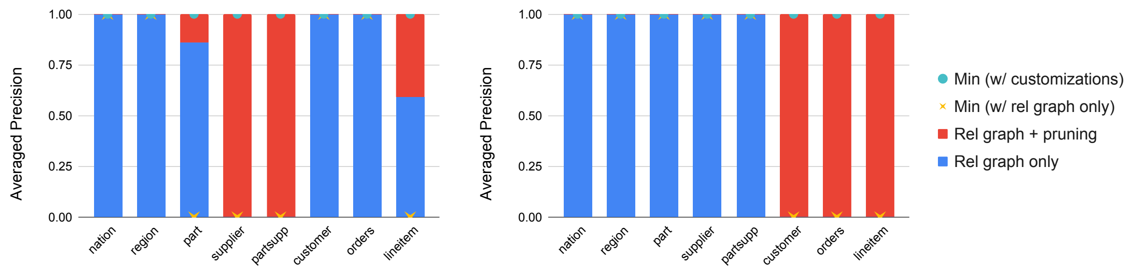
These relationships are fairly obvious when considering the semantics of the WordPress application and the column names, but they are hard for GDPRizer to determine automatically. With the WooCommerce plugin, we add six more edges in addition to the above. These edges essentially connect disconnected tables that have user IDs to the `wp_users` table. For example, we connect `wp_wc_payment_tokens.user_id` to `wp_users.id` to capture a user's payment tokens. In addition, we connect `wp_users.id` to the five columns in WooCommerce table, all with columns called `users_id`. GDPRizer recognizes the disconnected components and helps identifying the possible connecting edges based on datatypes; ultimately, it's the DBA's responsibility to know that columns named `user_id` map to `wp_users.id`.

Output filtering. GDPRizer's output filtering removes columns that contain sensitive data from the output. How often this customization is required depends on the application semantics; in the applications we looked at, only HotCRP requires filtering. For example, we filter 18 (out of 42) columns of the `PaperReview` table. These columns contain reviewer-specific information such as the reviewer's user ID, the reviewer's qualifications, their private comments to the program committee, etc. Relative to the total number of columns in the database (200 in HotCRP), filtering affects only a small number of columns. We now discuss in detail the impact of customizations on GDPRizer.

2.7.4 Impact of Customizations

We now evaluate how the individual customizations affect the accuracy of GDPRizer for the four applications. Generally, output filtering and edge pruning improve precision, while vertex and edge additions improve recall. Most applications benefit a lot from a single type of customization, while the others improve accuracy in smaller, but still important, ways. We present the results as stacked bar charts (Figures 2.7–2.9), with the blue part of the bar representing fully-automated extraction using GDPRizer's relationship graph only. Each customization stacks atop the blue bar and other customizations, indicating its relative impact. Our graphs show averages, by table, over the data subjects benchmarked, but we also indicate the *minimum* (*i.e.*, worst-case) per-data subject precision and recall with a yellow cross (no customizations) and a green dot (all customiations). A good result for GDPRizer would show the minimum at 100% with customizations enabled.

TPC-H. For the customer role, all the tables have 100% recall before pruning and all but four of them have 100% precision. The four tables with $<$ 100% precision are `part`, `supplier`, `partsupp`, and `lineitem`. After we prune the edges for tables that are irrelevant to customers, precision increases to 100% (see Figure 2.7a). For example, after we prune `supplier.s_suppkey` and supplier.s_nationkey, GDPRizer stops extracting suppliers' information and hence it's precision for the supplier table improves from 0% to 100%. The supplier role shows similar results, except that pruning edges to three tables with customer-related data increases precision to 100% (Figure 2.7b).



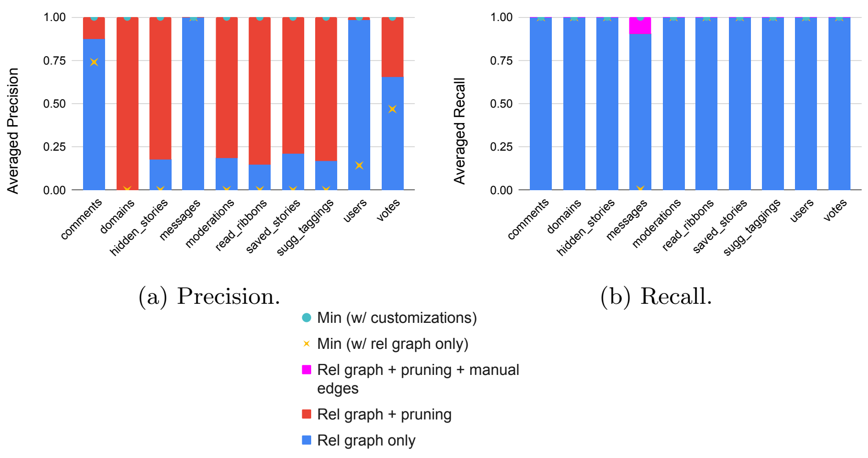
(a) Customer (b) Supplier

Figure 2.7: Precision of GDPRizer for TPC-H for data subjects with different roles: customers (a) and suppliers (b). Edge pruning improves both precision and recall (no other customizations are required for TPC-H).

Lobsters. We summarize our results for Lobsters in Figure 2.8. The figure only includes the tables that have sub-optimal precision/recall before customizations. After pruning, GDPRizer achieves 100% precision on all the tables (Fig. 2.8a) and 100% recall on all but the messages table, for which the recall is 92% (Fig. 2.8b). The reason for this imperfect recall is a missing edge between users.id and messages.author_user_id. Once we add the missing edge, GDPRizer's recall improves to 100%.

HotCRP. Figure 2.9 shows how precision and recall on HotCRP improve with successive customizations. Without customizations, per-table precision is at least 95%; with the pruning and filtering described earlier, precision improves to 100%. However, our experiments show that without any customizations, recall can be as low as 8% (Paper). The reason for the low recall value is the special encoding of the author information in the PaperConflict table. This in turn affects the recall of other tables that contain paper-related information, such as PaperConflict, PaperOption, and PaperReview. Once we add the v_author column to the Paper table, recall improves to 100% on all these tables. After these customizations, ReviewRating (99%) and TopicArea (13%) are the only tables left with imperfect recall. The missing relationship between tables is a missing edge between ContactInfo.contactId and ReviewRating.contactId. With that edge added, GDPRizer retrieves review ratings. The recall of TopicArea remains imperfect (96%) even after we add the edge between PaperTopic.topicId and TopicArea.topicId. This is because the relationship graph of HotCRP has two paths into TopicArea, one of which represents the topic areas for papers that a data subject submitted, while the other represents the areas of review interest of a data subject (meaningful only for program committee members). Since the former path is longer than the latter, GDPRizer ignores the former path and misses any topic areas associated with submitted papers that aren't also the data subject's preferred review areas.

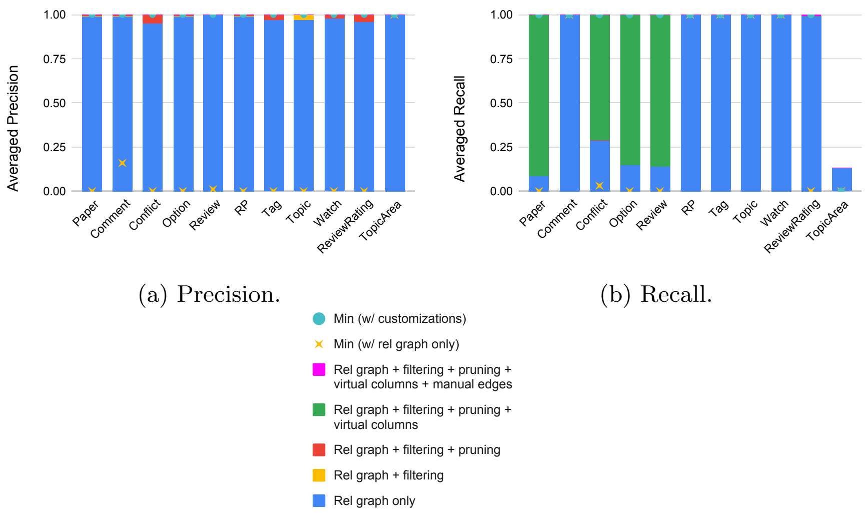
WordPress. For the base installation of WordPress, prior to customizations, GDPRizer achieves



(a) Precision.

(b) Recall.

Figure 2.8: Precision (a) and recall (b) for Lobsters tables with successive customizations. Edge pruning yields the largest improvement in precision, while manual edge addition is necessary to reach 100% recall on `messages`.



(a) Precision.

(b) Recall.

Figure 2.9: Precision (a) and recall (b) for HotCRP tables with successive customizations. Edge pruning and filtering improve precision, while the virtual column handling co-authorship is essential for recall.

perfect precision for all tables. However, it only achieves perfect recall on two tables, `wp_users` and `wp_usermeta`. For other tables with user data, such as `wp_comments` and `wp_posts`, recall is 0% because `wp_users` and `wp_usermeta` are in a component that is disconnected from the rest of the relationship graph (Figure 2.4). Therefore, GDPRizer's traversal starting from the `wp_users` table is unable to reach these tables, and hence extracts no data from them. Manually adding the missing edges to connect the components improves the recall to 100%, however. We see similar results for WordPress with the WooCommerce plugin: the original relationship graph has many disconnected

	[82]	[30]	[65]	GDPRizer
wp_user	✓ 🟩	🟥	✓ 🟩	✓ 🟩
wp_usermeta	✓ 🟩	✓ 🟩	🟥	✓ 🟩
wp_posts	🟥	🟥	🟥	✓ 🟩
wp_postmeta	🟥	🟥	🟥	✓ 🟩
wp_comments	✓ 🟩	✓ 🟩	✓ 🟩	✓ 🟩
wp_commentmeta	✓ 🟩	✓ 🟩	🟥	✓ 🟩

Figure 2.10: Comparison of GDPRizer with existing GDPR plugins for WordPress. For a table T, green boxes represent complete extraction of data while red represent no extraction.

components, but after edge addition, recall improves to 100% for all tables. We conclude that edge additions are crucial for GDPRizer to support applications with disconnected components in the their relationship graph (*e.g.*, because the application avoids join queries).

2.7.5 Comparison with GDPR Compliance plugins

WordPress's extensive collection of third-party plugins includes several plugins which are designed to aid administrators with GDPR compliance. Some of these plugins also support data access requests, and we compare GDPRizer to three of these existing GDPR plugins: GDPR Compliance and Cookie Consent[82], The GDPR Framework by Data443[30], and WP GDPR Compliance[65]. These plugins are quite popular: the first two have been installed over $30,000$ times and the last one over $200,000$ times.

We assess whether these plugins capture the information specified in our ground truth. The results for the base installation are in Figure 2.10 and those with the WooCommerce plugin in Figure 2.11. We find that GDPRizer successfully identifies user information from all the tables in the ground truth, while the existing plugins miss out on some of the tables. For example, all plugins fail to extract information from `wp_posts`. This may happen because the plugins are designed to serve data access requests from internet users who may have interacted with the WordPress site, but who do not have their own accounts on it (*e.g.*, casual commenters). But this illustrates that installing a plugin may be insufficient to achieve true compliance; GDPRizer, working at the level of the database schema, offers a broader set of options to the DBA.

Finally, with the WooCommerce plugin enabled, all plugins again miss out on data in some of the tables. This might be due to an oversight on the part of the plugin developers, or due to a different understanding of what information must be returned to users to comply with the GDPR. For example, some of the tables included in our ground truth, such as `download_log` and `api_keys`, contain backend information meant for the application and rather than end-users, even though this information is tied to a data subject. Under the GDPR, this information (*e.g.*, download events) must nevertheless be returned because it is identifiably associated with a data subject—a nuance that may have escaped the plugin developers, but which puts the plugins' users at risk of violating the GDPR.

	[82]	[30]	[65]	GDPRizer
`customer_information`	✓	✓	~	✓
`order_information`	✓	✓	~	✓
`order_to_product`	✓	✓	🟥	✓
`order_to_coupon`	🟥	🟥	🟥	✓
`download_log`	🟥	🟥	🟥	✓
`webhooks`	🟥	🟥	🟥	✓
`api_keys`	🟥	🟥	🟥	✓
`download_permissions`	🟥	🟥	🟥	✓
`payment_tokens`	🟥	🟥	🟥	✓

Figure 2.11: Comparison of GDPRizer with existing GDPR plugins for WordPress with the WooCommerce plugin. For a table T, green boxes represent complete extraction of data, red represent no extraction, and yellow represent partial extraction.

2.8 Conclusions

We explored retrofitting data access compliance onto legacy databases. The current state-of-the-art is to either use manual effort, or build plugins for specific applications. Given the sheer number of user-facing applications, it is worth asking if the process can be automated, or even partially automated. Our goal in building our data compliance tool, GDPRizer, was to understand the trade-offs between automation and manual effort in data access compliance. We started out with the basic components that we could expect for real-world applications—a database and a query log—and studied general-purpose approaches to compliance using these inputs. However, each real-world application we studied required a small but specific amount of manual customization from a DBA. Although this is much less effort than a completely manual solution, it still requires human intervention. The trade-off between manual effort and automation is worth exploring further, possibly along the lines of more "specialized" general-purpose tools. For instance, in web applications built using a framework such as Ruby on Rails or Django, there exists an Object Relational Model (ORM) that maps application objects to database tables. Further, there are standard conventions for table names and schema that could be exploited to decrease the amount of human effort required. While there are more questions to be explored in this regard, we believe it unlikely that a fully-automated solution exists. However, with the growing amount of privacy legislation, even partially automated solutions will go a long way in making the transition smoother for legacy systems.

GDPRizer partially automates data access compliance using information present in the schema and query log of an application. We analyzed the performance of GDPRizer with several applications and identified common customizations that were necessary for GDPRizer to extract user data correctly. We show that GDPRizer, along with the manual customizations, achieves compliance on the applications that we studied. We conclude that any compliance solution for legacy databases will require some manual effort from a domain expert, but the degree of effort can be minimized.

Chapter 3

Encrypted Databases

Cryptography offers many solutions for users to retain control over their private but outsourced data. The simplest of these solutions is to securely encrypt all data before uploading to any external cloud provider or service. However, this simple approach makes it impossible to perform any operations or computation on the data without downloading all of it. In particular, simple encryption does not support search or query capabilities over the encrypted data. The ability to efficiently search and query encrypted data has the potential to change how users store and process data and help increase the wide-scale deployment of end-to-end encryption. A key requirement for any practical encrypted search solution is handling search queries in sub-linear time. Sub-linear encrypted search can be achieved based on several cryptographic primitives, including property-preserving encryption (PPE), structured encryption (STE) and oblivious RAM (ORAM). Each of these primitives have been heavily investigated and are known to achieve different trade-offs between efficiency, expressiveness and security. In this thesis, we study structured encryption, which offers real-world efficiency and flexible trade-offs of functionality and security. STE is used to design encryption schemes for data structures, which in turn can be used to design bigger encrypted systems such as relational databases [46, 51].

3.1 Overview

In this work, we present new techniques to enhance the security of existing structured encryption (STE) schemes. STE schemes encrypt data structures in such a way that they can be privately queried. Special cases of STE include searchable symmetric encryption (SSE) and graph encryption. Like all sub-linear encrypted search solutions, STE allows a persistent adversarial server to derive some information about the input data. This derived information is referred to as the *leakage* of the STE scheme. To address this, a line of work on *leakage suppression* was recently initiated that focuses on techniques to mitigate the leakage of STE schemes. A notable example is the query equality suppression framework (Kamara et al. *CRYPTO'18*) which does not reveal to the server if two queries to the structure are equal. The framework takes as input a dynamic scheme (one that supports changing data) and produces a static scheme. It was left as an open question to design a solution that could yield dynamic constructions.

We propose a dynamic query equality suppression framework that transforms volume-hiding weakly dynamic STE schemes that leak the query equality into new *fully-dynamic* constructions that do not. We then use our framework to design three new fully-dynamic STE schemes that are "almost" and fully zero-leakage which, under natural assumptions on the data and query distributions, are asymptotically more efficient than using black-box ORAM.

However, our zero-leakage schemes are still inefficient as compared to more leaky, but optimally efficient structured encryption schemes. We then explore if it is possible to have more efficient STE schemes which still suppress the query equality. Our final contribution in this thesis is a practically efficient replica-based query equality suppression technique. The technique is based on the observation that an STE scheme could use information about the client's overall query distribution in order to create an encrypted data structure that optimizes query complexity and correctness, while still hiding the query equality pattern. We present a data transform that replicates an input dictionary according to a distribution, and use it to design a scheme that has optimal query complexity while suppressing the query equality.

3.2 Background and Related Work

Structured encryption Structured encryption was introduced by Chase and Kamara in [28] as a generalization of searchable symmetric encryption (SSE) [79, 29]. Several aspects of STE and SSE have been studied including dynamism [49, 48, 26, 63], expressiveness [24, 69, 33, 45, 46], locality and I/O-efficiency [25, 14, 26, 31, 15], security [80, 19, 34, 20, 12] and cryptanalysis [43, 23, 52, 87, 56, 18].

Leakage. All sub-linear encrypted search primitives leak information which has motivated the study of leakage attacks to investigate the real-world security of these primitives. In 2015, Naveed, Kamara and Wright [62] described data-recovery attacks in the snapshot setting against schemes that leak data equality and order. In 2012, Islam, Kuzu and Kantarcioglu [43] described a query-recovery attack against schemes that leak query co-occurrences (i.e., whether two keywords appear in the same document). The IKK attack was subsequently shown not to work in the standard adversarial model [23] but followup work described attacks in stronger adversarial models where the adversary is assumed to either know or choose a fraction of the client's data [23, 18]. The known-data attacks of [23] exploit co-occurrence leakage and require a large fraction of known data whereas the attacks of [18] require a smaller fraction of known-data and exploit response length leakage; making them applicable to ORAM-based solutions as well. The chosen-data attacks of [87] exploit the response identity (i.e., identifiers of the files that contain the keyword) whereas the recent attacks of [18] only exploit response lengths; again, making them applicable to ORAM-based solutions. Several works have also described leakage attacks on the profiles of known oblivious and encrypted range schemes [52, 56, 38, 39]. In [12], it is shown that highly-efficient STE schemes with zero-leakage queries can be achieved in the snapshot model.

Leakage suppression. Recently, Kamara, Moataz and Ohrimenko initiated the study of leakage suppression [50], which are methods to diminish and eradicate the leakage of STE schemes. There are two kinds of leakage suppression techniques: compilers and data transformations. Compilers take an STE scheme and transform it into a new scheme with similar efficiency but with an improved leakage profile. An example is the cache-based compiler (CBC) of [50] which is a generalization of the seminal Square Root ORAM construction of Goldreich and Ostrovsky [36]. The CBC takes any rebuildable STE scheme that leaks the query equality and possibly some other pattern patt, and transforms it into a new scheme that leaks only the non-repeating sub-pattern of patt. The non-repeating sub-pattern of a leakage pattern is the leakage it produces when queried only on non-repeating query sequences.

Data transformations change plaintext data structures in such a way that leakage is less harmful. The simplest example of a data transformation is padding, which mitigates response length leakage, but more sophisticated approaches include the clustering-based techniques of Bost and Fouque [21] and the transformation that underlies the PBS construction [50], both of which mitigate volume leakage. Recently, Kamara and Moataz also introduced computationally-secure transformations (as opposed to the previously mentioned approaches which are information-theoretic) to mitigate volume leakage [47]. In follow up work, Patel, Persiano, Yeo and Yung [71] proposed new volume-hiding constructions that achieve better query and storage efficiency.

Dynamic leakage suppression. The main advantage of suppression compilers over transformations is that they can be applied to large classes of schemes. For example, the CBC can be applied to any rebuildable STE scheme and, furthermore, [50] shows that any semi-dynamic STE scheme can be made rebuildable. An STE scheme is semi-dynamic if it supports additions but not deletions, and it is fully-dynamic if it supports both. The main limitation of the techniques from [50] is that they only produce *static* schemes even if the base construction is dynamic. While static STE schemes have several applications, dynamic schemes allow the encrypted data structure to adapt to changing data, which is more useful from a practical standpoint.

Oblivious RAM. Oblivious RAM was first proposed by Goldreich and Ostrovsky [36]. Several aspects of ORAM have been studied and improved in the last twenty years including its communication complexity, the number of rounds and client and server storage [67, 86, 37, 55, 78, 81, 34, 70]. Another line of work initiated by Wang et al. [85] considers the design of oblivious data structures, without making use of general-purpose ORAM techniques. These constructions are typically more efficient than using general-purpose ORAM but are usually static or require setting an upper bound the structure at setup time.

3.2.1 Our Contributions

We first address the main problem left open by [50] which is to design a dynamic leakage suppression framework for the query equality. As we will see, solving this open problem results in three new low- and zero-leakage dynamic constructions that, under natural conditions on the data and queries, are asymptotically more efficient than black-box ORAM simulation.

Dynamic leakage suppression. The suppression framework of [50], which includes the CBC and the rebuild compiler (RBC), can be used to compile any semi-dynamic STE scheme that leaks the query equality into a new scheme that does not. But, as discussed, this framework can only produce static schemes; i.e., it does not preserve the (semi-)dynamism of the base scheme. In this work, we propose dynamic variants of the CBC and RBC that suppress the query equality while preserving the dynamism of the base scheme.

Designing such compilers is challenging for several reasons. For example, consider that if the base scheme leaks the response length as well as the operation identity pattern (i.e., whether an operation is a query or an update), the adversary can learn the query equality as follows. Suppose that the largest response length observed is n and that it occurs at some time t. Furthermore, suppose that at time $t+1$ an update operation occurs and that at some time $t' > t+1$ another query occurs with response larger than n. For some datasets and query distributions, it would be reasonable for the adversary to infer that the two queries are for the same value which, effectively, is the query equality. Unfortunately, all currently-known fully-dynamic STE schemes leak both the response length and the operation identity patterns.

Our approach, therefore, is to start with schemes that do not leak the response length like PBS [50] and AVLH [47]. The challenge in using these schemes, however, is that they are not dynamic but only semi-dynamic or mutable (i.e., they only support edit operations). To address this, our compilers are designed to work with these limited forms of dynamism but this requires overcoming a set of additional technical challenges like "upgrading" the base scheme's dynamism from semi-dynamic or mutable to fully-dynamic without leaking any additional information.

Almost-zero leakage constructions. We apply our compilers to three base multi-map encryption schemes to construct dynamic zero- and almost zero-leakage multi-map encryption schemes. Our first construction results from applying our compilers to the PBS construction of [50]. This results in a dynamic variant of the AZL scheme [50] which, given a sequence of operations $(\mathsf{op}_1, \ldots, \mathsf{op}_t)$, reveals nothing on operations $(\mathsf{op}_1, \ldots, \mathsf{op}_{t-1})$ and then reveals the sum of the operations' response lengths on operation op_t. Similarly, our second construction results from applying our compilers to a variant of PBS and is a dynamic variant of the FZL scheme of [50]. This scheme has zero-leakage queries but only achieves probabilistic correctness. Our third construction, which results from applying our compilers to the AVLH construction of [47], also has ZL queries but achieves perfect correctness. We show that all three schemes are asymptotically more efficient than state-of-the-art black-box ORAM simulation under natural assumptions.

However, leakage-suppressed encrypted structures are significantly less efficient than their leakier counterparts. In fact, there exist asymptotically optimal constructions of both encrypted dictionaries and multi-maps that leak both the query equality and the volume patterns (e.g., $\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$ [[26]]). In order for leakage suppression to be practical, it is necessary to develop efficient techniques and explore the various trade-offs that leakage suppression introduces into structured encryption schemes.

Practical leakage suppression. We initiate the study of practical leakage suppression, and explore some of the trade-offs that are possible in this setting. We introduce the *query replication transform*, or QRT, to suppress the query equality, and present the first dictionary encryption scheme with optimal query complexity which does not leak the query equality pattern. Our scheme, RPL, uses information about the client's query distribution at setup time, and trades off server storage in order to improve query complexity. However, using information about the client's query distribution at setup can be tricky because the client need not follow the same distribution at query time. Even when the client deviates from the distribution used at setup, our scheme must preserve the security guarantee. This introduces a new trade-off into our design. We guarantee security by trading off correctness, suppressing the query equality regardless of the client's query distribution. We study the efficiency and correctness properties of our scheme, and show that it has optimal query complexity.

3.3 Preliminaries and Notation

Notation. We denote the security parameter as k, and all algorithms run in time polynomial in k. The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. $[n]$ is the set of integers $\{1,\ldots,n\}$, and $2^{[n]}$ is the corresponding power set. We write $x \leftarrow \chi$ to represent an element x being sampled from a distribution χ, and $x \overset{\$}{\leftarrow} X$ to represent an element x being sampled uniformly at random from a set X. The output x of an algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$. Given a sequence $\mathbf{v}$ of n elements, we refer to its i^{th} element as v_i or $\mathbf{v}[i]$. If S is a set then $\#S$ refers to its cardinality. If s is a string then $|s|_2$ refers to its bit length.

Sorting networks. A sorting network is a circuit of comparison-and-swap gates. A sorting network for n elements takes as input a collection of n elements $(a_1,\ldots,a_n)$ and outputs them in increasing order. Each gate g in an n-element network SN_n specifies two input locations $i,j \in [n]$ and, given a_i and a_j, returns the pair (a_i,a_j) if $i < j$ and (a_j,a_i) otherwise. Sorting networks can be instantiated with the asymptotically-optimal Ajtai-Komlos-Szemeredi network [11] which has size $O(n\log n)$ or Batcher's more practical network [17] with size $O(n\log^2 n)$ but with small constants.

The word RAM. Our model of computation is the word RAM. In this model, we assume memory holds an infinite number of w-bit words and that arithmetic, logic, read and write operations can all be done in $O(1)$ time. We denote by $|x|_w$ the word-length of an item x; that is, $|x|_w = |x|_2/w$. Here, we assume that $w = \Omega(\log k)$.

Abstract data types. An *abstract data type* specifies the functionality of a data structure. It is a collection of data objects together with a set of operations defined on those objects. Examples include sets, dictionaries (also known as key-value stores or associative arrays) and graphs. The operations associated with an abstract data type fall into one of two categories: query operations, which return information about the objects; and update operations, which modify the objects. If the abstract data type supports only query operations it is *static*, otherwise it is *dynamic*. We model a dynamic data type $\mathbf{T}$ as a collection of four spaces: the object space $\mathbb{D} = \{\mathbb{D}_k\}_{k\in\mathbb{N}}$, the query space $\mathbb{Q} = \{\mathbb{Q}_k\}_{k\in\mathbb{N}}$, the response space $\mathbb{R} = \{\mathbb{R}_k\}_{k\in\mathbb{N}}$ and the update space $\mathbb{U} = \{\mathbb{U}_k\}_{k\in\mathbb{N}}$. We also define the query map $\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}$ and the update map $\mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D}$ to represent operations associated with the dynamic data type. We refer to the query and update spaces of a data type as the operation space $\mathbb{O} = \mathbb{Q} \cup \mathbb{U}$. When specifying a data type $\mathbf{T}$ we will often just describe its maps $(\mathsf{qu}, \mathsf{up})$ from which the object, query, response and update spaces can be deduced. The spaces are ensembles of finite sets of finite strings indexed by the security parameter. We assume that $\mathbb{R}$ includes a special element $\perp$ and that $\mathbb{D}$ includes an empty object d_0 such that for all $q \in \mathbb{Q}$, $\mathsf{qu}(d_0, q) = \perp$.

Data structures. A type-$\mathbf{T}$ data *structure* is a representation of data objects in $\mathbb{D}$ in some computational model (as mentioned, here it is the word RAM). Typically, the representation is optimized to support qu as efficiently as possible; that is, such that there exists an efficient algorithm Query that computes the function qu. For data types that support multiple queries, the representation is often optimized to efficiently support as many queries as possible. As a concrete example, the dictionary type can be represented using various data structures depending on which queries one wants to support efficiently. Hash tables support Get and Put in expected $O(1)$ time whereas balanced binary search trees support both operations in worst-case $O(\log n)$ time.

Definition 3.3.1 (Structuring scheme)**.** *Let* $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D})$ *be a dynamic type. A type-*$\mathbf{T}$ *structuring scheme* $\mathsf{SS} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ *is composed of three polynomial-time algorithms that work as follows:*

- $\mathsf{DS} \leftarrow \mathsf{Setup}(d)$*: is a possibly probabilistic algorithm that takes as input a data object* $d \in \mathbb{D}$ *and outputs a data structure* DS*. Note that* d *can be represented in any arbitrary manner as long as its bit length is polynomial in* k*. Unlike* DS*, its representation does not need to be optimized for any particular query.*
- $r \leftarrow \mathsf{Query}(\mathsf{DS}, q)$*: is an algorithm that takes as input a data structure* DS *and a query* $q \in \mathbb{Q}$ *and outputs a response* $r \in \mathbb{R}$*.*
- $\mathsf{DS} \leftarrow \mathsf{Update}(\mathsf{DS}, u)$*: is a possibly probabilistic algorithm that takes as input a data structure* DS *and an update* $u \in \mathbb{U}$ *and outputs a new data structure* DS*.*

Here, we allow Setup and Update to be probabilistic but not Query. This captures most data structures but the definition can be extended to include structuring schemes with probabilistic query algorithms. We say that a data structure DS *instantiates* a data object $d \in \mathbb{D}$ if for all $q \in \mathbb{Q}$, $\mathsf{Query}(\mathsf{DS}, q) = \mathsf{qu}(d, q)$. We denote this by $\mathsf{DS} \equiv d$. We denote the set of queries supported by a structure DS as $\mathbb{Q}_{\mathsf{DS}}$; that is,

$$\mathbb{Q}_{\mathsf{DS}} \overset{def}{=} \left\{ q \in \mathbb{Q} : \mathsf{Query}(\mathsf{DS}, q) \neq \perp \right\}.$$

Similarly, the set of responses supported by a structure DS is denoted $\mathbb{R}_{\mathsf{DS}}$.

Definition 3.3.2 (Correctness)**.** *Let* $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D})$ *be a dynamic type. A type-*$\mathbf{T}$ *structuring scheme* $\mathsf{SS} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ *is perfectly correct if it satisfies the following properties:*

1. *(static correctness) for all* $d \in \mathbb{D}$*,*

$$\Pr\left[\, \mathsf{DS} \equiv d : \mathsf{DS} \leftarrow \mathsf{Setup}(d) \,\right] = 1,$$

where the probability is over the coins of Setup*.*

2. *(dynamic correctness) for all* $d \in \mathbb{D}$ *and* $u \in \mathbb{U}$*, for all* $\mathsf{DS} \equiv d$*,*

$$\Pr\left[\, \mathsf{Update}(\mathsf{DS}, u) \equiv \mathsf{up}(d, u) \,\right] = 1,$$

where the probability is over the coins of Update*.*

Note that the second condition guarantees the correctness of an updated structure whether the original structure was generated by a setup operation or a previous update operation. Weaker notions of correctness (e.g., for data structures like Bloom filters) can be derived from Definition 3.3.2.

Basic data structures. We use structures for several basic data types including arrays, dictionaries and multi-maps which we recall here. An array RAM of capacity n stores n items at locations 1 through n and supports read and write operations. We write $v := \mathsf{RAM}[i]$ to denote reading the item at location i and $\mathsf{RAM}[i] := v$ the operation of storing an item at location i. A dictionary structure DX of capacity n holds a collection of n label/value pairs $\{(\ell_i, v_i)\}_{i \leq n}$ and supports get and put operations. We write $v_i := \mathsf{DX}[\ell_i]$ to denote getting the value associated with label ℓ_i and $\mathsf{DX}[\ell_i] := v_i$ to denote the operation of associating the value v_i in DX with label ℓ_i. A multi-map structure MM with capacity n is a collection of n label/tuple pairs $\{(\ell_i, \mathbf{v}_i)_i\}_{i \leq n}$ that supports get and put operations. Similarly to dictionaries, we write $\mathbf{v}_i := \mathsf{MM}[\ell_i]$ to denote getting the tuple associated with label ℓ_i and $\mathsf{MM}[\ell_i] := \mathbf{v}_i$ to denote operation of associating the tuple $\mathbf{v}_i$ to label ℓ_i. Multi-maps are the abstract data type instantiated by an inverted index. In the encrypted search literature multi-maps are sometimes referred to as indexes, databases or tuple-sets (T-sets).

Data structure logs. Given a structure DS that instantiates an object d, we will be interested in the sequence of update operations needed to create a new structure DS' that also instantiates d. We refer to this as the *query log* of DS and assume the existence of an efficient algorithm Log that takes as input DS and outputs a tuple $(u_1, \ldots, u_n)$ such that adding $u_1, \ldots, u_n$ to an empty structure results in some $\mathsf{DS}' \equiv d$.

Extensions. An important property we will need from a data structure is that it be *extendable* [50] in the sense that, given a structure DS one can create another structure $\overline{\mathsf{DS}} \neq \mathsf{DS}$ that is functionally equivalent to DS but that also supports a number of *dummy* queries. We say that a structure is efficiently extendable if there exist a query set $\overline{\mathbb{Q}} \supset \mathbb{Q}$ and a PPT algorithm $\mathsf{Ext}_{\mathbf{T}}$ that takes as input a structure DS of type $\mathbf{T}$ and a *capacity* $\lambda \geq 1$ and returns a new structure $\overline{\mathsf{DS}}$ also of type $\mathbf{T}$ [1] such that: (1) $\overline{\mathsf{DS}} \equiv d$; and (2) for all $q \in \overline{\mathbb{Q}} \setminus \mathbb{Q}$, $\mathsf{Query}(\overline{\mathsf{DS}}, q) = \perp$.We say that $\overline{\mathsf{DS}}$ is an extension of DS and that DS is a sub-structure of $\overline{\mathsf{DS}}$.

Cryptographic protocols. We denote by $(\mathsf{out}_A, \mathsf{out}_B) \leftarrow \Pi_{A,B}(X, Y)$ the execution of a two-party protocol Π between parties A and B, where X and Y are the inputs provided by A and B, respectively; and out_A and out_B are the outputs returned to A and B, respectively.

3.3.1 Structured Encryption

We recall the syntax definition of STE.

Definition 3.3.3 (Structured encryption [28]). *An interactive structured encryption scheme* $\Sigma = (\mathsf{Setup}, \mathsf{Operate})$ *consists of an algorithm and a two-party protocol that work as follows:*

- $(K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \lambda, \mathsf{DS})$*: is a probabilistic polynomial-time algorithm that takes as input a security parameter* 1^k*, a query capacity* $\lambda \geq 1$ *and a type-*$\mathbf{T}$ *structure* DS*. It outputs a secret key* K*, a state* st *and an encrypted structure* EDS*. If* $\mathsf{DS} \equiv d_0$*, it outputs an empty* EDS*.*
- $\big((st', r), \mathsf{EDS}'\big) \leftarrow \mathsf{Operate}_{\mathbf{C},\mathbf{S}}\big((K, st, \mathsf{op}), \mathsf{EDS}\big)$*: is a two-party protocol executed between a client and a server where the client inputs a secret key* K*, a state* st *and an operation* op *and the server inputs an encrypted structure* EDS*. The client receives as output a (possibly) updated state* st' *and a response* $r \in \mathbb{R} \cup \perp$ *while the server receives a (possibly updated) encrypted structure* EDS'*.*

If Σ *also has a* $\mathsf{Rebuild}$ *protocol as defined below, we say that it is rebuildable,*

- $\big((st', K'), \mathsf{EDS}'\big) \leftarrow \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}\left((K, st), \mathsf{EDS}\right)$*: is a two-party protocol executed between the client and server where the client inputs a secret key* K *and a state* st*. The server inputs an encrypted data structure* EDS*. The client receives an updated state* st' *and a new key* K' *as output while the server receives a new structure* EDS'*.*

[1]We consider that the inclusion of dummy queries in a query space does not impact the type of a structure.

Operations. Note that an STE schemes usually supports more than a single operation and the syntax above can be used (or extended) to capture this in one of two ways. The first is to notice that the $\mathsf{Operate}$ protocol can take as input an operation op that describes one of a set of operations and its operands. For example, if $\Sigma_{\mathsf{DS}} = (\mathsf{Setup}, \mathsf{Operate})$ supports both query and add operations, then op can have the form $\mathsf{op} = (\mathsf{qry}, q)$ to denote a query operation for q or $\mathsf{op} = (\mathsf{add}, a)$ to denote an add operation for a. The $\mathsf{Operate}$ protocol can then operate on EDS accordingly and output $((st, r), \mathsf{EDS}')$, where $r \neq \perp$ and $\mathsf{EDS}' = \mathsf{EDS}$ in the case of a query, and where $r = \perp$ and $\mathsf{EDS}' \neq \mathsf{EDS}$ in the case of an add. For notational convenience we will usually omit the flags qry or add and just write $\mathsf{op} = q$ or $\mathsf{op} = a$ to denote that it is a query or an add. This formulation is particularly convenient when working with schemes that hide which operation is being executed, as will be the case with our main constructions. Another approach is to include the different operations explicitly in Σ_{DS}'s syntax. For example, if it supports queries and adds, then we would write $\Sigma_{\mathsf{DS}} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Add})$, where Query is a special case of $\mathsf{Operate}$ that (usually) outputs a response $r \neq \perp$ and an $\mathsf{EDS}' = \mathsf{EDS}$ and Add is a special case that (usually) outputs $r = \perp$ and $\mathsf{EDS}' \neq \mathsf{EDS}$. This formulation is particularly convenient when working with schemes that reveal which operation is being executed, as will be the case with the constructions we use as building blocks.

Dynamism. We consider several kinds of dynamic STE schemes. The first are *fully-dynamic* schemes which support add and delete operations. We usually refer to such schemes simply as *dynamic*. Add operations insert a query/response pair (q, r) into the data structure whereas delete operations remove query/response pairs (q, r) associated with a given query q. If a scheme only handles add operations we say it is *semi-dynamic*. Finally, we consider *mutable* schemes which are schemes that support an edit operation which takes as input a query/response pair (q, r') and changes a pre-existing pair (q, r) to (q, r'). If a scheme is either semi-dynamic or mutable we say that it is *weakly dynamic*.

Security. We recall the notion of adaptive semantic security for STE.

Definition 3.3.4 (Security [29, 28]). *Let $\Sigma = (\mathsf{Setup}, \mathsf{Operate}_{\mathbf{C},\mathbf{S}}, \mathsf{Rebuild}_{\mathbf{C},\mathbf{S}})$ be a structured encryption scheme and consider the following probabilistic experiments where $\mathcal{C}$ is a stateful challenger, $\mathcal{A}$ is a stateful adversary, $\mathcal{S}$ is a stateful simulator, $\Lambda = (\mathsf{patt}_{\mathsf{S}}, \mathsf{patt}_{\mathsf{O}}, \mathsf{patt}_{\mathsf{R}})$ is a leakage profile, $\lambda \geq 1$ and $z \in \{0,1\}^*$:*

Real$_{\Sigma,\mathcal{C},\mathcal{A}}(k)$: *given z and λ the adversary $\mathcal{A}$ outputs a structure DS and receives EDS from the challenger, where $(K, st, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \lambda, \mathsf{DS})$. $\mathcal{A}$ then adaptively chooses a polynomial-size sequence of operations $(\mathsf{op}_1, \ldots \mathsf{op}_m)$. For all $1 \leq i \leq m$ the challenger and adversary do the following:*

1. *if op_i is a query or an update, they execute $\mathsf{Operate}_{\mathcal{C},\mathcal{A}}\big((K, st, \mathsf{op}_i), \mathsf{EDS}\big)$;*
2. *if op_i is a rebuild, they execute $\mathsf{Rebuild}_{\mathcal{C},\mathcal{A}}\big((K, st), \mathsf{EDS}\big)$.*

Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.

$\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k)$: *given z and λ the adversary $\mathcal{A}$ outputs a structure DS of type $\mathbf{T}$. Given $\mathsf{patt_S}(\mathsf{DS})$, the simulator returns an encrypted structure EDS to $\mathcal{A}$. $\mathcal{A}$ then adaptively chooses a polynomial-size sequence of operations $(\mathsf{op}_1, \ldots, \mathsf{op}_m)$. For all $1 \leq i \leq m$, the challenger, simulator and adversary do the following:*

1. *if op_i is either a query or an update, $\mathcal{S}$ is given $\mathsf{patt_O}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_i)$ and it executes $\mathsf{Operate}_{\mathcal{S},\mathcal{A}}$ with $\mathcal{A}$;*
2. *if op_i is a rebuild, $\mathcal{S}$ is given $\mathsf{patt_R}(\mathsf{DS})$ and it executes $\mathsf{Rebuild}_{\mathcal{S},\mathcal{A}}$ with $\mathcal{A}$;*

Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.

We say that Σ is Λ-secure if there exists a PPT *simulator $\mathcal{S}$ such that for all* PPT *adversaries $\mathcal{A}$, for all $\lambda \geq 1$ and all $z \in \{0,1\}^*$,*

$$|\Pr[\mathbf{Real}_{\Sigma,\mathcal{C},\mathcal{A}}(k) = 1] - \Pr[\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k) = 1]| \leq \mathsf{negl}(k).$$

Note that security of non-rebuildable schemes can be recovered by not allowing rebuild operations.

Leakage. We extend the leakage patterns defined in [50] to the dynamic setting. In particular [50] defined leakage patterns as functions of queries on a static data type. We will have to extend the definitions to account for general operations (queries or updates) on a dynamic data type. Let $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D})$ be a dynamic data type. We assume that updates can be written as query/response pairs, i.e., $\mathbb{U} = \mathbb{Q} \times \mathbb{R}$. Given a data structure d and a sequence of t operations $\mathsf{op}_1, \ldots, \mathsf{op}_t$, we denote by d_t the structure that results from applying the given sequence of operations to d. Consider the following leakage patterns,

- the *operation identity pattern* is the function family $\mathsf{oid} = \{\mathsf{oid}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{oid}_{k,t} : \mathbb{D}_k \times \mathbb{O}_k^t \to \{0,1\}^t$ such that $\mathsf{oid}_{k,t}(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \mathbf{m}$, where $\mathbf{m}$ is a binary t-dimensional vector such that $\mathbf{m}[i] = 0$ if $\mathsf{op}_i \in \mathbb{Q}$ and $\mathbf{m}[i] = 1$ if $\mathsf{op}_i \in \mathbb{U}$;

- the *update query equality pattern* is the function family $\mathsf{uqeq} = \{\mathsf{uqeq}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{uqeq}_{k,t} : \mathbb{D}_k \times \mathbb{U}_k^t \to \{0,1\}^{t\times t}$ such that $\mathsf{uqeq}_{k,t}(d, u_1, \ldots, u_t) = M$, where M is a binary $t \times t$ matrix such that for updates $u_i = (q_i, r_i)$ and $u_j = (q_j, r_j)$, $M[i,j] = 1$ if $q_i = q_j$ and $M[i,j] = 0$ otherwise;

- the *operation total response length pattern* is the function family $\mathsf{otrlen} = \{\mathsf{otrlen}_k\}_{k\in\mathbb{N}}$ with $\mathsf{otrlen}_k : \mathbb{D}_k \times \mathbb{O}_k^t \to \mathbb{N}$ such that $\mathsf{otrlen}_k(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \sum_{q\in\mathbb{Q}_k} |\mathsf{qu}(d_t, q)|_w$ and d_t is d after t operations.;

- the *operation data size pattern* is the function family $\mathsf{odsize} = \{\mathsf{odsize}_k\}_{k\in\mathbb{N}}$ with $\mathsf{odsize}_k : \mathbb{D}_k \times \mathbb{O}_k^t \to \mathbb{N}$ such that $\mathsf{odsize}_k(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = |d_t|_w$;

- the *operation log size pattern* is the function family $\mathsf{olsize} = \{\mathsf{olsize}_k\}_{k\in\mathbb{N}}$ with $\mathsf{olsize}_k : \mathbb{D}_k \times \mathbb{O}_k^t \to \mathbb{N}$ such that $\mathsf{olsize}_k(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \#\mathsf{Log}(\mathsf{DS})$ where DS is an instantiation of d_t such that $\mathsf{DS} \equiv d_t$;

- the *operation max log length pattern* is the function family $\mathsf{omllen} = \{\mathsf{omllen}_k\}_{k\in\mathbb{N}}$ with $\mathsf{omllen}_k : \mathbb{D}_k \times \mathbb{O}_k^t \to \mathbb{N}$ such that $\mathsf{omllen}_k(d, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \max_{\mathsf{op}\in\mathsf{Log}(d_t)} |\mathsf{op}|_w$.

Note that in the static setting, i.e., when $\mathbb{O} = \mathbb{Q}$, the leakage patterns $\mathsf{otrlen}, \mathsf{odsize}, \mathsf{olsize}, \mathsf{omllen}$ are equivalent to the patterns $\mathsf{trlen}, \mathsf{dsize}, \mathsf{lsize}, \mathsf{mllen}$ originally defined in [50].

Leakage sub-patterns. We recall the notion of leakage sub-patterns introduced in [50]. Given a leakage pattern patt, it can be decomposed into sub-patterns capturing its behavior on restricted classes of query sequences. In particular, we can decompose a leakage pattern into repeating and non-repeating sub-patterns. The non-repeating sub-pattern is pattern that results from evaluating patt on non-repeating query sequences (i.e., where all queries are unique).

Definition 3.3.5 (Non-repeating sub-patterns)**.** *Let* $\mathbf{T} = (\mathsf{qu} : \mathbb{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbb{D} \times \mathbb{U} \to \mathbb{D})$ *be a dynamic data type and* $\mathsf{patt} : \mathbb{D} \times \mathbb{Q}^t \to \mathbb{X}$ *be a query leakage pattern. The non-repeating sub-pattern of* patt *is the function* uniq *such that*

$$\mathsf{patt}(\mathsf{DS}, q_1, \ldots, q_t) = \begin{cases} \mathsf{uniq}(\mathsf{DS}, q_1, \ldots, q_t) & \text{if } q_i \neq q_j \text{ for all } i, j \in [t], \\ \mathsf{other}(\mathsf{DS}, q_1, \ldots, q_t) & \text{otherwise.} \end{cases}$$

Safe extensions. We recall and extend the notion of safe extension from [50] to support updates.

Definition 3.3.6 (Safe extensions)**.** *Let* $\Lambda = (\mathsf{patt}_\mathsf{S}, \mathsf{patt}_\mathsf{Q}, \mathsf{patt}_\mathsf{U}, \mathsf{patt}_\mathsf{R})$ *be a leakage profile. We say that an extension* Ext *is* Λ*-safe if for all* $k \in \mathbb{N}$*, for all* $d \in \mathbb{D}_k$*, for all* $\mathsf{DS} \equiv d$*, for all* $\lambda \geq 1$*, for all* $\overline{\mathsf{DS}}$ *output by* $\mathsf{Ext}(\mathsf{DS}, \lambda)$*, for all* $t \in \mathbb{N}$*, for all* $\mathbf{op} = (\mathsf{op}_1, \ldots, \mathsf{op}_t) \in \mathbb{O}_k^t$*,*

- $\mathsf{patt}_\mathsf{S}(\overline{\mathsf{DS}}) \leq \mathsf{patt}_\mathsf{S}(\mathsf{DS})$*;*

- $\mathsf{patt}_\mathsf{Q}(\overline{\mathsf{DS}}, q_1, \ldots, q_p) \leq \mathsf{patt}_\mathsf{Q}(\mathsf{DS}, q_1, \ldots, q_p)$*, where* $(q_1, \ldots, q_p)$ *is the sub-sequence of queries in* $\mathbf{op}$*;*

- $\mathsf{patt}_\mathsf{U}(\overline{\mathsf{DS}}, u_1, \ldots, u_w) \leq \mathsf{patt}_\mathsf{U}(\mathsf{DS}, u_1, \ldots, u_w)$*, where* $(u_1, \ldots, u_w)$ *is the sub-sequence of updates in* $\mathbf{op}$*;*

- $\mathsf{patt}_\mathsf{R}(\overline{\mathsf{DS}}) \leq \mathsf{patt}_\mathsf{R}(\mathsf{DS})$*,*

where $\mathsf{patt}_1 \leq \mathsf{patt}_2$ *means that* patt_1 *can be simulated from* patt_2*.*

3.4 Our Dynamic Suppression Framework

In this section, we present a dynamic variant of the query equality suppression framework proposed by [50]. Our framework transforms non-rebuildable weakly-dynamic STE schemes that leak the query equality into fully-dynamic STE schemes that do not. Recall that the static framework relies on two compilers: (1) a rebuild compiler (RBC) which transforms a semi-dynamic and non-rebuildable scheme into a static and rebuildable one; and (2) the cache-based compiler (CBC) which transforms a static and rebuildable scheme that leaks the query equality into a static scheme that does not.

Challenges. One of the challenges in designing a dynamic variant of the CBC is handling subtle correlations between various leakage patterns. For example, suppose the base STE scheme leaks the response length and the operation identity patterns and consider a sequence of operations $(\mathsf{op}_1, \ldots, \mathsf{op}_4)$ such that $\mathsf{op}_1 = q_1$, $\mathsf{op}_2 = q_2$, $\mathsf{op}_3 = u_3$ and $\mathsf{op}_4 = q_4$. Now, given the operation identities and the response lengths, suppose the adversary observes that: q_1 has the largest response length ℓ_1; that q_3 is an update operation; and that q_4 has response length $\ell_1 + 1$. From this, it can reasonably infer that q_1 might be equal to q_4 which is a "probabilistic" variant of the query equality. It is therefore not enough to suppress the exact query equality but also the patterns that can reveal partial information about it.

To address this, our compiler will have to suppress the response length and the operation identity in addition to the query equality. One can trivially suppress the former by padding responses to the maximum length but this induces a large storage cost; especially when the response lengths are skewed. A better approach would be to start with base schemes that are volume-hiding in the sense that they hide the response lengths (without naive padding). Unfortunately, all volume-hiding constructions we are aware of [47, 71] are only weakly dynamic. Our goal, therefore, will be to design a compiler that suppresses the query equality, the operation identity and the response length while upgrading the base scheme from being weakly-dynamic to fully-dynamic.

Another important challenge we must overcome is making the base scheme rebuildable. [50] already showed how to make semi-dynamic schemes rebuildable but, in our setting, we also need to handle mutable constructions which do not support add operations but only edits. To summarize, our compiler has to handle the following challenges:

- *(weak dynamism)* it must transform a weakly-dynamic (i.e., either semi-dynamic or mutable) scheme to a fully-dynamic one;
- *(operation identity)* it must suppress the operation identity; that is, queries and updates should look identical.
- *(rebuild)* it must make the base scheme rebuildable even if it is only weakly dynamic.

Overview of the dynamic CBC. The dynamic CBC is similar to the static CBC of [50] with the exception of a few steps to handle adds and edits. Let $\Sigma_{\mathsf{DS}} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Add})$ be a semi-dynamic STE scheme and let $\Sigma_{\mathsf{DX}} = (\mathsf{Setup}, \mathsf{Get}, \mathsf{Put})$ be a semi-dynamic and zero-leakage dictionary encryption scheme. The compiler produces a new scheme $\Sigma_{\mathsf{DDS}} = (\mathsf{Setup}, \mathsf{Operate})$ that works as follows. Given a structure DS and a capacity $\lambda \geq 1$, its setup algorithm outputs a structure $\mathsf{EDDS} = (\overline{\mathsf{EDS}}, \mathsf{EDX})$, where $\overline{\mathsf{EDS}}$ is the encryption of a λ-extension of DS and EDX is an encryption of a dictionary with capacity λ. Operations on EDDS are handled as follows:

- *(queries)* to make a query q, the client first executes a get on EDX for q. If this returns $\perp$ (i.e., q has never been issued before) the client queries $\overline{\mathsf{EDS}}$ for q and receives a response r. The client then does a put on EDX to add the query/response pair (q, r). If, on the other hand, the get on the cache returned a response $r \neq \perp$, the client queries $\overline{\mathsf{EDS}}$ for an unused dummy value and puts the query/response pair (q, r) in EDX;

- *(adds)* to add a query/response pair (q, r), the client executes a get on EDX for an arbitrary query and ignores the response. It then queries $\overline{\mathsf{EDS}}$ for an unused dummy and puts (q, r) in EDX;

- *(edits)* to edit the response of an existing query q (e.g., by either adding to it, deleting from it or changing it), the client first executes a get on EDX for q. If this returns $\perp$, the client queries $\overline{\mathsf{EDS}}$ for q and receives a response r. It then edits r, which results in a new response r', and puts (q, r') in EDX. If, on the other hand, the get on the cache returned a response $r \neq \perp$, the client queries $\overline{\mathsf{EDS}}$ for an unused dummy, edits r and puts the edited query/response pair (q, r') in EDX.

Note that for every operation, the dynamic CBC executes a get on EDX, then a query on $\overline{\mathsf{EDS}}$ and, finally, a put on EDX. Furthermore, $\overline{\mathsf{EDS}}$ is never queried for a query q more than once. Intuitively, the first property will guarantee that the scheme suppresses the operation identity while the second will guarantee that it suppresses the query equality.

Every operation executed on EDDS consumes a (unique) dummy item from $\overline{\mathsf{EDS}}$. And since it holds λ dummies, it needs to be rebuilt after λ operations so that it can continue to be used. We now describe how this rebuild is achieved.

Overview of the dynamic RBC. We have two main goals when rebuilding $\mathsf{EDDS} = (\overline{\mathsf{EDS}}, \mathsf{EDX})$. The first is to build a new $\overline{\mathsf{EDS}}$ structure $\overline{\mathsf{EDS}'}$ that holds the λ dummies. The second is to make sure that $\overline{\mathsf{EDS}'}$ holds the most up-to-date responses for all the queries. Note that the second goal is non-trivial because of the way adds and edits are handled. In particular, the most up-to-date response for a query q can be either in $\overline{\mathsf{EDS}}$ or in EDX depending on whether it has been added, edited or never modified. More precisely, we have that after λ operations, if a query/response pair (q, r) is in the cache then r is the most up-to-date response for q. On the other hand, if a pair (q, r) is not in the cache then the the main structure $\overline{\mathsf{EDS}}$ holds the most up-to-date response for q. In the following, we refer to a query/response pair (q, r) as *valid* if r is the most up-to-date response for q and as *invalid* if it is not. Our rebuild protocol must then extract the valid query/response pairs from EDX and $\overline{\mathsf{EDS}}$ and add them to $\overline{\mathsf{EDS}'}$ with a minimal amount of leakage. [2]

The protocol consists of five phases: (1) *initialization*, where an array RAM is initialized at the server; (2) *extract-and-tag*, where all the query/response pairs are retrieved from $\overline{\mathsf{EDS}}$ and EDX, tagged according to their validity and stored in an encrypted array at the server; (3) *sort-and-shuffle*, where the encrypted array is (obliviously) sorted to partition the invalid and valid query/response pairs so that the former can be deleted and the latter are randomly shuffled; (4) *update*, where the valid query/response pairs in the array are added to a new $\overline{\mathsf{EDS}'}$ structure; and (5) *cache setup*, where a new cache structure EDX' is created. More precisely, it works as follows:

1. *(initialization):* the server initializes an array RAM.

2. *(extract-and-tag)* the client sequentially retrieves all the query/response pairs (q, r) in $\overline{\mathsf{EDS}}$ and EDX. For all (q, r) in EDX, it adds an encryption of (q, r, f) to RAM, where f is a random non-zero k-bit value we refer to as a validity tag. If there are less than λ entries in EDX, it queries it on arbitrary values until it reaches λ queries and for each of these arbitrary queries it adds an encryption of $(\perp, \perp, 0)$ to RAM. For all query/response pairs (q, r) in $\overline{\mathsf{EDS}}$, it adds an encryption of (q, r, f) to RAM, where f is set to 0 if q was present in EDX and f is set to a random non-zero k-bit value otherwise. For each dummy in $\overline{\mathsf{EDS}}$, the client adds an encryption of $(\perp, \perp, f)$ to RAM, where f is a random non-zero k-bit value. Throughout this phase, the client also keeps count of the number of entries with 0 tags. Notice that the valid query/response pairs and the dummies are all tagged with random non-zero validity tags whereas the invalid pairs and the entries that result from the "arbitrary" queries on EDX are tagged with 0.

3. *(sort-and-shuffle)* the client obliviously sorts RAM according to the validity tags. Since the valid pairs and the dummies have random non-zero tags and the rest have 0 tags, this step will randomly shuffle the valid pairs and dummies and store the rest at the start of the array. The client then asks the server to delete the first t entries, where t is the number of entries with 0 tags. At this point, the array only holds valid query/response pairs.

4. *(update)* the client creates a new structure $\overline{\mathsf{EDS}'}$ by retrieving the query/response pairs in RAM and adding them to $\overline{\mathsf{EDS}'}$. How exactly this is done depends on the kind of dynamism Σ_{DS} supports:

    - *(semi-dynamic)* if it is semi-dynamic, the client initializes an empty structure DS_0 and encrypts it with Σ_{DS} before storing it at the server. This new encrypted structure is $\overline{\mathsf{EDS}'}$. The client sequentially retrieves the query/response pairs (q, r) from RAM and adds them to $\overline{\mathsf{EDS}'}$.

    - *(mutable)* if Σ_{DS} is mutable we can only use edit operations. The client then sets up "placeholder" structure $\widetilde{\mathsf{DS}}$ that it will encrypt and edit until it holds the necessary data.

[2]Note that invalid query/response pairs in $\overline{\mathsf{EDS}}$ result from the pair existing in $\overline{\mathsf{EDS}}$ from setup (i.e., not being added) but being edited during the last λ operations.

Note that for this to work, the placeholder must be large enough to hold the latest version of DS (i.e., the structure DS after the λ operations) and it must be "safe" in the sense that encrypting and editing the placeholder must not leak more than operating on the original structure.

5. *(cache setup)* the client generates an empty dictionary with capacity λ and encrypts it with Σ_{DX} and sets it to be EDX'.

Finally, the protocol outputs a rebuilt structure $\mathsf{EDDS}' = (\overline{\mathsf{EDS}'}, \mathsf{EDX}')$.

3.4.1 Security

We now analyze the security of our dynamic suppression framework. We state and prove two theorems of security. Theorem 3.4.1 analyzes the case when Σ_{DS} is semi-dynamic and Theorem 3.4.3 analyzes the case where Σ_{DS} is mutable. For Theorem 3.4.1, we assume Σ_{DS} has leakage profile

$$\Lambda_{\mathsf{DS}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{A}}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, (\mathsf{qeq}^{\mathsf{ds}}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}), \mathsf{patt}^{\mathsf{ds}}_{\mathsf{A}}\right),$$

and Σ_{DX} has profile

$$\Lambda_{\mathsf{DX}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{G}}, \mathcal{L}_{\mathsf{P}}) = \left(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}, \perp, \perp\right).$$

Theorem 3.4.1 (Semi-dynamic)**.** *If* Σ_{DS} *is* Λ_{DS}*-secure, if* Ext *is* $(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{uniq}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{A}})$*-safe, and if* Σ_{DX} *is* Λ_{DX}*-secure, then* Σ_{DDS} *is* Λ_{DDS}*-secure, where*

$$\Lambda_{\mathsf{DDS}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{O}}, \mathcal{L}_{\mathsf{R}}) = \left(\left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}\right), \mathsf{uniq}, \left(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}, \mathsf{patt}_1, \mathsf{patt}_2, \mathsf{patt}_3\right)\right)$$

and patt_1*,* patt_2 *and* patt_3 *are defined as,*

- $\mathsf{patt}_1(\mathsf{DS}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}_0), \mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}\right)$
- $\mathsf{patt}_2(\mathsf{DS}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q)\right)_{q \in \mathbb{Q}_{\mathsf{DS}}}$
- $\mathsf{patt}_3(\mathsf{DS}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{A}}(\mathsf{DS}_0, a)\right)_{a \in \mathsf{Log}(\mathsf{DS}_\lambda)}$,

where uniq *is the non-repeating sub-pattern of* $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}$*,* $\mathsf{DS}_0 \equiv d_0$ *and* DS_λ *is the updated* DS *after* $\lambda \geq 1$ *operations.*

Proof. Let $\mathcal{S}_{\mathsf{DS}}$ and $\mathcal{S}_{\mathsf{DX}}$ be the simulators guaranteed to exist by the adaptive security of Σ_{DS} and Σ_{DX}, respectively. To prove security, we construct a simulator $\mathcal{S}$ such that the output of $\mathbf{Ideal}_{\Sigma_{\mathsf{DDS}},\mathcal{A},\mathcal{S}}(k)$ is distributed like the output of a $\mathbf{Real}_{\Sigma_{\mathsf{DDS}},\mathcal{C},\mathcal{A}}(k)$ experiment. Our simulator $\mathcal{S}$ works as follows:

Simulating Setup: given λ and leakage $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS})$ and $\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}(\mathsf{DX})$, $\mathcal{S}$ computes

$$\overline{\mathsf{EDS}} \leftarrow \mathcal{S}_{\mathsf{DS}}\left(\mathcal{S}_1\left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS})\right)\right),$$

where $\mathcal{S}_1$ is guaranteed to exist since $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\overline{\mathsf{DS}}) \leq \mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS})$. $\mathcal{S}$ then computes

$$\mathsf{EDX} \leftarrow \mathcal{S}_{\mathsf{DX}}\Big(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}(\mathsf{DX})\Big).$$

It sets $\mathsf{cnt} := 0$ and returns $\mathsf{EDDS} = (\overline{\mathsf{EDS}}, \mathsf{EDX})$ to $\mathcal{A}$.

Simulating Operate: given the leakage $\mathsf{uniq}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_t)$ for t operations on DS, where $\mathsf{uniq}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \mathsf{uniq}(\mathsf{DS}, q_1, \ldots, q_t)$ for queries q_i and updates $u_i = (q_i, r_i)$, $\mathcal{S}$ works as follows.

If $\mathsf{cnt} > \lambda$ it aborts. If $\mathsf{cnt} \leq \lambda$, it first uses $\mathcal{S}_{\mathsf{DX}}(\perp)$ to simulate a Get query to EDX. It then uses

$$\mathcal{S}_{\mathsf{DS}}\Big(\mathcal{S}_2\Big(M_0, \mathsf{uniq}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_t)\Big)\Big)$$

to simulate a query to $\overline{\mathsf{EDS}}$, where M_0 is a $t \times t$ zero matrix. Note that it suffices to have this query leakage and $\mathsf{qeq} = M_0$ since we only ever make unique queries to EDS during the t operations. $\mathcal{S}_2$ is guaranteed to exist since $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\overline{\mathsf{DS}}, q_1, \ldots, q_t) \leq \mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q_1, \ldots, q_t)$. Recall that $\Sigma_{\mathsf{DS}}.\mathsf{Query}_{\mathbf{C},\mathbf{S}}$ is an interactive protocol so here $\mathcal{S}$ is using $\mathcal{S}_{\mathsf{DS}}$ to play the role of the client. It then uses $\mathcal{S}_{\mathsf{DX}}(\perp)$ to simulate a Put query to EDX. Finally it sets $\mathsf{cnt} := \mathsf{cnt} + 1$.

Simulating Rebuild: $\mathcal{S}$ sends $\mathsf{EDX} \leftarrow \mathcal{S}_{\mathsf{DX}}(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}(\mathsf{DX}))$ to the adversary. $\mathcal{S}$ then initializes an array RAM of capacity $\mathsf{lsize} + \lambda$ and samples a key $K_L \leftarrow \mathsf{Gen}(1^k)$.

- for $i = 1, \ldots, \lambda, \mathcal{S}$ uses $\mathcal{S}_{\mathsf{DX}}(\perp)$ to simulate a Get query to EDX, computes $\mathrm{ct}_i \leftarrow \mathsf{Enc}(K_L, 0^{\mathsf{omllen}})$ and sets $\mathsf{RAM}[i] := \mathrm{ct}_i$;
- query phase: for every $q \in \mathbb{Q}_{\overline{\mathsf{DS}}}$ use

  $$\mathcal{S}_{\mathsf{EDS}}\Big(\mathcal{S}_3\big(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q)\big)\Big)$$

  to simulate the query to EDS. $\mathcal{S}_3$ must exist due to the query safety of the extension scheme. For each such query set $i := i + 1$, compute $\mathrm{ct}_i \leftarrow \mathsf{Enc}(K_L, 0^{\mathsf{omllen}})$ and set $\mathsf{RAM}[i] := \mathrm{ct}_i$;
- sort-and-shuffle phase: for each gate $g = (i, j)$ in SN_m, after receiving ct_i and ct_j from the adversary, return $\mathrm{ct}'_i \leftarrow \mathsf{Enc}(K_L, 0^{\mathsf{omllen}})$ and $\mathrm{ct}'_j \leftarrow \mathsf{Enc}(K_L, 0^{\mathsf{omllen}})$;
- send $(\mathsf{olsize} + \lambda)$ to the adversary to truncate the array RAM;
- add phase: use $\mathcal{S}_{\mathsf{DS}}(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}_0))$ to simulate a Setup operation and for all $j \in [\mathsf{olsize} + \lambda]$, retrieve $\mathsf{RAM}[j]$ and use

  $$\mathcal{S}_{\mathsf{EDS}}\Big(\big(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{A}}(\mathsf{DS}_0, a)\big)_{a \in \mathsf{Log}(\mathsf{DS}_\lambda)}\Big)$$

  to simulate $(\mathsf{olsize} + \lambda)$ Add operations on the new EDS'. Set $\mathsf{cnt} := 0$.

It remains to show that the probability that $\mathbf{Ideal}_{\Sigma_{\mathsf{DDS}},\mathcal{A},\mathcal{S}}(k)$ outputs 1 is negligibly-close to the probability that $\mathbf{Real}_{\Sigma_{\mathsf{DDS}},\mathcal{C},\mathcal{A}}(k)$ outputs 1 for any PPT adversary $\mathcal{A}$. We do this using the following sequence of games:

Game_0: corresponds to a $\mathbf{Real}_{\Sigma_{\mathsf{DDS}},\mathcal{C},\mathcal{A}}(k)$ experiment.

Game_1: is the same as Game_0 except that during Setup; EDS is replaced with the output of $\mathcal{S}_{\mathsf{DS}}(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\overline{\mathsf{DS}}))$ and the ith execution of $\Sigma_{\mathsf{DS}}.\mathsf{Query}$ (i.e., the one for op_i) is replaced with a simulated execution between $\mathcal{S}_{\mathsf{DS}}\big(M_i, \mathsf{uniq}(\overline{\mathsf{DS}}, \mathsf{op}_1, \ldots, \mathsf{op}_i)\big)$ and the adversary, where M_i is the $i \times i$ zero matrix. Since EDS is only queried on non-repeating sequences for all operations in Game_0 it suffices to give $\mathcal{S}_{\mathsf{DS}}$ leakage $\big(M_i, \mathsf{uniq}(\overline{\mathsf{DS}}, \mathsf{op}_1, \ldots, \mathsf{op}_i)\big)$. If $\mathsf{cnt} = \lambda$; (1) in the query phase of $\mathsf{Rebuild}$, the queries are replaced with a simulated execution of $\mathcal{S}_{\mathsf{DS}}\big(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\overline{\mathsf{DS}}, q)\big)$ for all $q \in \mathbb{Q}_{\overline{\mathsf{DS}}}$ (2) in the add phase of $\mathsf{Rebuild}$, EDS is replaced with the output of $\mathcal{S}_{\mathsf{DS}}(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}_0))$ and (3) the adds to the new EDS are replaced with a simulated execution of $\mathcal{S}_{\mathsf{DS}}\big((\mathsf{patt}_{\mathsf{A}}(\mathsf{DS}_\lambda, a))_{a \in \mathsf{Log}(\mathsf{DS}_\lambda)}\big)$. The probabilities that Game_0 and Game_1 output 1 are negligibly-close, otherwise the adaptive-security of Σ_{DS} would be violated.

Game_2: is the same as Game_1 except for the following. $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\overline{\mathsf{DS}})$ is replaced with $\mathcal{S}_1(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}))$. For all operations op_i, $\mathsf{uniq}(\overline{\mathsf{DS}}, \mathsf{op}_1, \ldots, \mathsf{op}_i)$ is replaced with $\mathcal{S}_2(\mathsf{uniq}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_i))$. When $\mathsf{cnt} = \lambda$, in the query phase of the $\mathsf{Rebuild}$ protocol, $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\overline{\mathsf{DS}}, q)$ is replaced with $\mathcal{S}_3(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q))$. Since all the queries are unique, it suffices to give $\mathcal{S}_3$ the leakage $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q)$. The probabilities that Game_1 and Game_2 output 1 are negligibly-close, otherwise the $(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{uniq}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{A}})$-safety of Ext would be violated.

Game_3: is the same as Game_2 except that EDX is replaced with the output $\mathcal{S}_{\mathsf{DX}}(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}(\mathsf{DX}))$ and every execution of Get and Put are replaced with simulated executions between $\mathcal{S}_{\mathsf{DX}}(\perp)$ and the adversary. The probabilities that Game_2 and Game_3 output 1 are negligibly-close, otherwise the $(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}, \perp, \perp)$-security of Σ_{DX} would be violated.

Game_4: is the same as Game_3 except that every ciphertext ct_i is replaced with $\mathsf{Enc}(K_L, 0^{\mathsf{omllen}})$. The probabilities that Game_3 and Game_4 output 1 are negligibly-close, otherwise the CPA-security of SKE would be violated.

The Theorem follows by observing that Game_4 is exactly an $\mathbf{Ideal}_{\Sigma_{\mathsf{DDS}},\mathcal{A},\mathcal{S}}(k)$ experiment. □

Before we prove our Theorem for mutable schemes, recall that the rebuild protocol needs to setup a placeholder structure that can be edited to realize the new data object. This placeholder must be setup and edited with minimal leakage. We do this with the notion of a safe placeholder which we define below.

Definition 3.4.2 (Safe placeholder). *A placeholder structure* $\widetilde{\mathsf{DS}}$ *is* $(\mathsf{patt}_{\mathsf{S}}, \mathsf{patt}_{\mathsf{Q}}, \mathsf{patt}_{\mathsf{E}})$*-safe for a structure* DS *if, for all queries* $q_1, \ldots, q_t$*, for all edits* $e_1, \ldots, e_t$*,*

- $\mathsf{patt}_{\mathsf{S}}(\widetilde{\mathsf{DS}}) \leq \mathsf{patt}_{\mathsf{S}}(\mathsf{DS})$,

- $\mathsf{patt}_{\mathsf{Q}}(\widetilde{\mathsf{DS}}, q_1, \ldots, q_t) \leq \mathsf{patt}_{\mathsf{Q}}(\mathsf{DS}, q_1, \ldots, q_t)$,
- $\mathsf{patt}_{\mathsf{E}}(\widetilde{\mathsf{DS}}, e_1, \ldots, e_t) \leq \mathsf{patt}_{\mathsf{A}}(\mathsf{DS}, e_1, \ldots, e_t)$.

We assume that there exists an efficient algorithm $\mathsf{GenPlaceholder}$ that takes as input some state information and generates a safe placeholder. We now prove Theorem 3.4.3, assuming that Σ_{DS} has leakage profile

$$\Lambda_{\mathsf{DS}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{E}}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, (\mathsf{qeq}^{\mathsf{ds}}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}), \mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}}\right),$$

and Σ_{DX} has the same profile as above.

Theorem 3.4.3 (Mutable)**.** *If* Σ_{DS} *is* Λ_{DS}*-secure, if* Ext *is* $(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{uniq}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}})$*-safe, if* $\widetilde{\mathsf{DS}}$ *is an* $(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}})$*-safe placeholder for* DS_λ*, and if* Σ_{DX} *is* Λ_{DX}*-secure, then* Σ_{DDS} *is* Λ_{DDS}*-secure, where*

$$\Lambda_{\mathsf{DDS}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{O}}, \mathcal{L}_{\mathsf{R}}) = \left(\left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}\right), \mathsf{uniq}, \left(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}, \mathsf{patt}_1, \mathsf{patt}_2, \mathsf{patt}_3\right)\right)$$

and patt_1*,* patt_2 *and* patt_3 *are defined as,*

- $\mathsf{patt}_1(\mathsf{DS}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}_\lambda), \mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}\right)$
- $\mathsf{patt}_2(\mathsf{DS}) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q)\right)_{q \in \mathbb{Q}_{\mathsf{DS}}}$
- $\mathsf{patt}_3(\mathsf{DS}_\lambda) = \left(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}}(\mathsf{DS}_\lambda, e)\right)_{e \in \mathsf{Log}(\mathsf{DS}_\lambda)}$,

where uniq *is the non-repeating sub-pattern of* $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}$*, and* DS_λ *is the updated* DS *after* $\lambda \geq 1$ *operations.*

Proof. Let $\mathcal{S}_{\mathsf{DS}}$ and $\mathcal{S}_{\mathsf{DX}}$ be the simulators guaranteed to exist by the adaptive security of Σ_{DS} and Σ_{DX}, respectively. To prove security, we construct a simulator $\mathcal{S}$ such that the output of $\mathbf{Ideal}_{\Sigma_{\mathsf{DDS}},\mathcal{A},\mathcal{S}}(k)$ is distributed like the output of a $\mathbf{Real}_{\Sigma_{\mathsf{DDS}},\mathcal{C},\mathcal{A}}(k)$ experiment. Our simulator $\mathcal{S}$ works as follows:

Simulating $\mathsf{Setup}, \mathsf{Operate}$: $\mathcal{S}$ executes the same steps as the simulator for semi-dynamic schemes in the proof of Theorem 3.4.1. Since leakage is the same and the dynamic cache-based compiler is identical in operation for these steps, this suffices for simulation.

Simulating $\mathsf{Rebuild}$: $\mathcal{S}$ sends $\mathsf{EDX} \leftarrow \mathcal{S}_{\mathsf{DX}}(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}(\mathsf{DX}))$ to the adversary. $\mathcal{S}$ then initializes an array RAM of capacity $\mathsf{lsize} + \lambda$ and samples a key $K_L \leftarrow \mathsf{Gen}(1^k)$.

- for $i = 1, \ldots, \lambda, \mathcal{S}$ uses $\mathcal{S}_{\mathsf{DX}}(\bot)$ to simulate a Get query to EDX, computes $\mathrm{ct}_i \leftarrow \mathsf{Enc}(K_L, 0^{\mathsf{omllen}})$ and sets $\mathsf{RAM}[i] := \mathrm{ct}_i$;
- query phase: for every $q \in \mathbb{Q}_{\overline{\mathsf{DS}}}$ use

$$\mathcal{S}_{\mathsf{DS}}\Big(\mathcal{S}_3\big(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q)\big)\Big)$$

to simulate the query to EDS. $\mathcal{S}_3$ must exist due to the query safety of the extension scheme. For each such query set $i := i + 1$, compute $\mathrm{ct}_i \leftarrow \mathsf{Enc}(K_L, 0^{\mathsf{omllen}})$ and set $\mathsf{RAM}[i] := \mathrm{ct}_i$;

- sort-and-shuffle phase: for each gate $g = (i, j)$ in SN_m, after receiving ct_i and ct_j from the adversary, return $\mathrm{ct}'_i \leftarrow \mathsf{Enc}(K_L, 0^{\mathsf{omllen}})$ and $\mathrm{ct}'_j \leftarrow \mathsf{Enc}(K_L, 0^{\mathsf{omllen}})$;
- send $(\mathsf{olsize} + \lambda)$ to the adversary to truncate the array RAM;
- edit phase: use $\mathcal{S}_{\mathsf{DS}}(\mathcal{S}_4(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}_\lambda)))$ to simulate a Setup operation and for all $j \in [\mathsf{olsize}+\lambda]$, retrieve $\mathsf{RAM}[j]$ and use
$$\mathcal{S}_{\mathsf{DS}}\bigg(\mathcal{S}_4\bigg(\big(\mathsf{patt}_{\mathsf{E}}(\mathsf{DS}_\lambda, e)\big)_{e\in\mathsf{Log}(\mathsf{DS}_\lambda)}\bigg)\bigg)$$
to simulate $(\mathsf{olsize} + \lambda)$ Edit operations on the new EDS'. $\mathcal{S}_4$ must exist due to the setup and update safety of the placeholder $\widetilde{\mathsf{DS}}$. Set $\mathsf{cnt} := 0$.

It remains to show that the probability that $\mathbf{Ideal}_{\Sigma_{\mathsf{DDS}},\mathcal{A},\mathcal{S}}(k)$ outputs 1 is negligibly-close to the probability that $\mathbf{Real}_{\Sigma_{\mathsf{DDS}},\mathcal{C},\mathcal{A}}(k)$ outputs 1 for any PPT adversary $\mathcal{A}$. We do this using the following sequence of games:

Game_0: corresponds to a $\mathbf{Real}_{\Sigma_{\mathsf{DDS}},\mathcal{C},\mathcal{A}}(k)$ experiment.

Game_1: is the same as Game_0 except that during Setup; EDS is replaced with the output of $\mathcal{S}_{\mathsf{DS}}(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\overline{\mathsf{DS}}))$ and the ith execution of $\Sigma_{\mathsf{DS}}.\mathsf{Query}$ (i.e., the one for op_i) is replaced with a simulated execution between $\mathcal{S}_{\mathsf{DS}}\big(M_i, \mathsf{uniq}(\overline{\mathsf{DS}}, \mathsf{op}_1, \ldots, \mathsf{op}_i)\big)$ and the adversary, where M_i is the $i \times i$ zero matrix. Note that because EDS is only queried on non-repeating sequences for all operations in Game_0 it suffices to give $\mathcal{S}_{\mathsf{DS}}$ leakage $\big(M_i, \mathsf{uniq}(\overline{\mathsf{DS}}, \mathsf{op}_1, \ldots, \mathsf{op}_i)\big)$. If $\mathsf{cnt} = \lambda$; (1) in the query phase of $\mathsf{Rebuild}$, the queries are replaced with a simulated execution of $\mathcal{S}_{\mathsf{DS}}\big(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\overline{\mathsf{DS}}, q)\big)$ for all $q \in \mathbb{Q}_{\overline{\mathsf{DS}}}$ (2) in the edit phase of $\mathsf{Rebuild}$, EDS is replaced with the output of $\mathcal{S}_{\mathsf{DS}}(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\widetilde{\mathsf{DS}}))$ and (3) the edits to EDS are replaced with a simulated execution of $\mathcal{S}_{\mathsf{DS}}\big(\big(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}}(\widetilde{\mathsf{DS}}, e)\big)_{e\in\mathsf{Log}(\mathsf{DS}_\lambda)}\big)$. The probabilities that Game_0 and Game_1 output 1 are negligibly-close, otherwise the adaptive-security of Σ_{DS} would be violated.

Game_2: is the same as Game_1 except for the following. $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\overline{\mathsf{DS}})$ is replaced with $\mathcal{S}_1(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}))$. For all operations op_i, $\mathsf{uniq}(\overline{\mathsf{DS}}, \mathsf{op}_1, \ldots, \mathsf{op}_i)$ is replaced with $\mathcal{S}_2(\mathsf{uniq}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_i))$. When $\mathsf{cnt} = \lambda$, in the query phase of the $\mathsf{Rebuild}$ protocol, $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\overline{\mathsf{DS}}, q)$ is replaced with $\mathcal{S}_3(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q))$. Since all the queries are unique, it suffices to give $\mathcal{S}_3$ the leakage $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}(\mathsf{DS}, q)$. The probabilities that Game_1 and Game_2 output 1 are negligibly-close, otherwise the $(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{uniq}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}})$-safety of Ext would be violated.

Game_3: is the same as Game_2 except that during the edit phase of $\mathsf{Rebuild}$, $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\widetilde{\mathsf{DS}})$ is replaced with $\mathcal{S}_4(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}_\lambda))$ and for all $e \in \log(\mathsf{DS}_\lambda)$, $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}}(\widetilde{\mathsf{DS}}, e)$ is replaced with $\mathcal{S}_4(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}}(\mathsf{DS}_\lambda, e))$. The probabilities that Game_2 and Game_3 output 1 are negligibly-close otherwise the $(\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{Q}}, \mathsf{patt}^{\mathsf{ds}}_{\mathsf{E}})$-safety of the placeholder $\widetilde{\mathsf{DS}}$ would be violated.

Game_4: is the same as Game_3 except that EDX is replaced with the output $\mathcal{S}_{\mathsf{DX}}(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}(\mathsf{DX}))$ and every execution of Get and Put are replaced with simulated executions between $\mathcal{S}_{\mathsf{DX}}(\perp)$ and the adversary. The probabilities that Game_3 and Game_4 output 1 are negligibly-close, otherwise the $(\mathsf{patt}^{\mathsf{dx}}_{\mathsf{S}}, \perp, \perp)$-security of Σ_{DX} would be violated.

Game_5: is the same as Game_4 except that every ciphertext ct_i is replaced with $\mathsf{Enc}(K_L, 0^{\mathsf{omllen}})$. The probabilities that Game_4 and Game_5 output 1 are negligibly-close, otherwise the CPA-security of SKE would be violated.

The Theorem follows by observing that Game_5 is exactly an $\mathbf{Ideal}_{\Sigma_{\mathsf{DDS}},\mathcal{A},\mathcal{S}}(k)$ experiment. □

This completes the security analysis of our dynamic cache-based compiler. In the next section, we will discuss the general efficiency of the compiler.

3.4.2 Efficiency of the Dynamic Cache-Based Compiler

We now analyze the efficiency of the schemes produced by our suppression framework and compare it to using black-box ORAM simulation.

Operation complexity. The efficiency of Σ_{DDS} clearly depends on the efficiency of its building blocks Σ_{DS} and Σ_{DX}. Recall that for every operation op on EDDS, the client executes: one get operation on EDX, one query operation on $\overline{\mathsf{EDS}}$ and one put operation on EDX. This leads to an operation complexity of

$$\mathsf{time}^{\mathsf{dds}}_{\mathsf{O}} = \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} + \mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} + \mathsf{time}^{\mathsf{dx}}_{\mathsf{P}},$$

where $\mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}}$ is the query complexity of Σ_{DS}, and $\mathsf{time}^{\mathsf{dx}}_{\mathsf{G}}$ and $\mathsf{time}^{\mathsf{dx}}_{\mathsf{P}}$ are the get and put complexities of Σ_{DX}.

Rebuild complexity. Recall that the $\mathsf{Rebuild}$ protocol of Σ_{DDS} executes: (1) λ gets on EDX; (2) $\#\mathbb{Q}_{\mathsf{DS}}$ queries on EDS; (3) an oblivious sort on an array of size $\#\mathbb{Q}_{\mathsf{DS}} + 2\cdot\lambda$; and (4) $\#\mathbb{Q}_{\mathsf{DS}_\lambda}$ adds or edits on EDS. The complexity of steps (1) and (2) is

$$\lambda \cdot \mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} + \#\mathbb{Q}_{\mathsf{DS}} \cdot \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}}.$$

The complexity of steps (3) and (4) depend on the sorting network used and the storage at the client. Using Batcher's bitonic sort [17] with $O(1)$ client storage [50], steps (3) and (4) have complexity

$$O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r\in\mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda} + \#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{u\in\mathbb{U}} \mathsf{time}^{\mathsf{ds}}_{\mathsf{U}}(|u|)\right), \tag{3.1}$$

where $\mathsf{time}^{\mathsf{ds}}_{\mathsf{U}}(|u|)$ is either the add or the edit complexity of Σ_{DS}, $\mathbb{Q}_{\mathsf{DS}_\lambda}$ is the query space of DS_λ, and $\mathbb{R}_{\mathsf{DS}_\lambda}$ is the corresponding response space for the queries $q \in \mathbb{Q}_{\mathsf{DS}_\lambda}$. Note that if $\max_{u\in\mathbb{U}} \mathsf{time}^{\mathsf{ds}}_{\mathsf{U}}(|u|) = O\left(\log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right)$, then Equation (3.1) above is

$$O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r\in\mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right).$$

Adding steps (1) through (4) we have

$$\mathsf{time}^{\mathsf{dds}}_{\mathsf{R}} = \lambda \cdot \mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} + \#\mathbb{Q}_{\mathsf{DS}} \cdot \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} + O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r\in\mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right). \tag{3.2}$$

Operations & rebuild. It follows from the above that the time $\mathsf{time}^{\mathsf{ds}}_{\lambda\mathsf{O}+\mathsf{R}}$ to execute λ operations and to rebuild the structure is

$$\begin{aligned}\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} &= \lambda \cdot \mathsf{time}^{\mathsf{dds}}_{\mathsf{O}} + \mathsf{time}^{\mathsf{dds}}_{\mathsf{R}} \\ &= \lambda \cdot \left(\mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} + 2 \cdot \mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} + \mathsf{time}^{\mathsf{dx}}_{\mathsf{P}}\right) + \#\mathbb{Q}_{\mathsf{DS}} \cdot \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} \\ &\quad + O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right). \end{aligned} \tag{3.3}$$

The complexity above depends in part on the efficiency of the scheme Σ_{DX} used for the underlying cache. Several constructions can be used including the "standard" cache, square-root ORAM or the more efficient tree-based ORAM [81]. In the following, we analyze the complexity of Σ_{DDS} based on different instantiations of Σ_{DX}.

Using the standard cache. The standard (zero-leakage) cache is an array of size λ that stores encryptions of label/value pairs (ℓ, v) where the labels all have the same size and where the values are padded to the maximum value length. To execute a get for a label ℓ, the client retrieves the entire encrypted array, decrypts it and keeps the value associated with ℓ. To insert or edit a label/value pair, the client retrieves the entire encrypted array, decrypts it, inserts the new pair or modifies an existing pair, re-encrypts the array and sends it back to he server. It follows that the get and put complexities of the standard cache are

$$\mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} = \mathsf{time}^{\mathsf{dx}}_{\mathsf{P}} = O\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w\right),$$

Combining this with Equation (3.3), we have

$$\begin{aligned}\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} &= (\lambda + \#\mathbb{Q}_{\mathsf{DS}}) \cdot \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} + O\left(\lambda^2 \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w)\right) \\ &\quad + O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right).\end{aligned}$$

Using a tree-based cache. The scheme Σ_{DX} can also be instantiated with a tree-based ORAM like Path ORAM [81] which has get and put complexity

$$\mathsf{time}^{\mathsf{dx}}_{\mathsf{G}} = \mathsf{time}^{\mathsf{dx}}_{\mathsf{P}} = O\left(\max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \lambda\right),$$

where λ is the number of entries stored in the ORAM. Combining this with Equation 3.3, we have

$$\begin{aligned}\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} &= (\lambda + \#\mathbb{Q}_{\mathsf{DS}}) \cdot \mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} + O\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \lambda\right) \\ &\quad + O\left(\#\mathbb{Q}_{\mathsf{DS}_\lambda} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right).\end{aligned} \tag{3.4}$$

Comparison to black-box ORAM simulation. With the exception of the construction of [60], ORAM does not traditionally support re-sizing. So to compare our constructions with black-box

ORAM simulation based on state-of-the-art ORAMs (e.g., Path ORAM [81]) [3] we have to assume that the ORAM is initialized with some upper-bound on the size. We use an "upper-bound" data structure which we denote DS^*. More precisely, to setup the ORAM simulation for a structure DS, the ORAM is initialized to hold DS^* so that DS can expand to fill the allocated space. The ORAM simulation of one operation on DS using a tree-based ORAM then has complexity,

$$\mathsf{time}^{\mathsf{tree}}_{\mathsf{O}} = \mathrm{B}^{\mathsf{ds}}_{\mathsf{Q}} \cdot O\left(\log^2 \frac{|\mathsf{DS}^*|_2}{B}\right) \cdot \frac{B}{w},$$

where $\mathrm{B}^{\mathsf{ds}}_{\mathsf{Q}}$ is the number of blocks that need to be read to answer a query, B is the block size of the ORAM and w is the word length (in bits). Since the ORAM does not have to be rebuilt, $\mathsf{time}^{\mathsf{tree}}_{\lambda\mathsf{O}+\mathsf{R}}$ is the same as the time complexity of λ operations. Setting $B = \max_{r\in\mathbb{R}_{\mathsf{DS}^*}} |r|_2$ as an upper limit on possible response length, we have,

$$\mathsf{time}^{\mathsf{tree}}_{\lambda\mathsf{O}} = \lambda \cdot \mathrm{B}^{\mathsf{ds}}_{\mathsf{Q}} \cdot O\left(\log^2 \frac{|\mathsf{DS}^*|_2}{\max_{r\in\mathbb{R}_{\mathsf{DS}^*}} |r|_2}\right) \cdot \max_{r\in\mathbb{R}_{\mathsf{DS}^*}} |r|_w. \tag{3.5}$$

To compare the efficiency of our schemes with black-box ORAM simulation, we examine Equation (3.4). Assuming that $\lambda = O(\#\mathbb{Q}_{\mathsf{DS}})$,[4] and $\mathsf{time}^{\mathsf{ds}}_{\mathsf{Q}} = O(\log \#\mathbb{Q}_{\mathsf{DS}})$ we have that $\#\mathbb{Q}_{\mathsf{DS}_\lambda} \leq \#\mathbb{Q}_{\mathsf{DS}} + \lambda = O(\#\mathbb{Q}_{\mathsf{DS}})$. Combining the first two terms in Equation (3.4) we get,

$$\begin{aligned}\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} = O(\#\mathbb{Q}_{\mathsf{DS}} \cdot \log \#\mathbb{Q}_{\mathsf{DS}}) + O\left(\lambda \cdot \max_{r\in\mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \lambda\right)\\ + O\left(\#\mathbb{Q}_{\mathsf{DS}} \cdot \max_{r\in\mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}}\right).\end{aligned} \tag{3.6}$$

From Equation (3.6), we observe that $\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}}$ is asymptotically dominated by

$$O\left(\#\mathbb{Q}_{\mathsf{DS}} \cdot \max_{r\in\mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}}\right).$$

Comparing Equations (3.5) and (3.6), we have the following proposition.

Proposition 3.4.4. *If* $\lambda = O(\#\mathbb{Q}_{\mathsf{DS}})$, $\#\mathbb{Q}_{\mathsf{DS}} = O(\#\mathbb{Q}_{\mathsf{DS}^*})$ *and* $\mathrm{B}^{\mathsf{ds}}_{\mathsf{Q}} = \omega(1)$, *then*

$$\mathsf{time}^{\mathsf{dds}}_{\lambda\mathsf{O}+\mathsf{R}} = o\left(\mathsf{time}^{\mathsf{tree}}_{\lambda\mathsf{O}}\right).$$

For structures with constant-time queries, $\mathrm{B}^{\mathsf{ds}}_{\mathsf{Q}} = 1$ so our approach improves asymptotically over ORAM simulation whenever

$$\max_{r\in\mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w = o\left(\max_{r\in\mathbb{R}_{\mathsf{DS}^*}} |r|_w\right).$$

For a concrete efficiency comparison we refer the reader to Section 3.4.4.

[3] Note that some ORAM constructions can achieve better asymptotic query complexity [70] but we use Path ORAM for its simplicity and real-world practicality.

[4] This is a conservative assumption on λ. In practice, the selection of λ is crucial to the efficiency of the scheme. The question of selecting the optimal λ for efficiency is interesting and can be further explored.

3.4.3 Concrete Instantiations

In this section we show how to apply our framework to two concrete schemes: the piggyback scheme PBS from [50] which is a semi-dynamic construction and the advanced volume-hiding scheme AVLH^d from [47] which is mutable. The leakage profiles of the resulting schemes is minimal and only reveal information pertaining to the total size of the structure.

Our PBS-Based Constructions

PBS is a non-rebuildable semi-dynamic STE scheme. It is parameterized with a batch size α and supports query and add operations. PBS queries and adds in batches in the sense that when executing a query q_1 it only retrieves a fixed number of batches from q_1' s response and retrieves the next set of batches only when a new query q_2 occurs. In the meantime, q_2 is inserted into a queue until enough queries are made for the client to retrieve q_1's entire response. Adds are handled similarly. When a sequence of queries or adds is complete, all the remaining batches in the queue are retrieved or pushed.

PBS has two variants. The first is a perfectly correct variant which incurs some small amount of query leakage; namely, for sequences of non-repeating queries, it leaks the number of batches required to process the sequence; and for sequences with repeating queries, it reveals the query equality and the response lengths. The second variant achieves only probabilistic correctness but the non-repeating sub-pattern of its query leakage is $\perp$. The application of our framework to the first variant results in a dynamic variant of the AZL construction from [50] whereas applying it to the second variant results in a dynamic variant of the FZL construction from [50].

Leakage profile of PBS. The leakage profile of the perfectly correct variant of PBS is

$$\Lambda_{\mathsf{PBS}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{A}}) = (\mathsf{tbrlen}, \mathsf{rqeq}, \mathsf{alen}),$$

where tbrlen, rqeq and alen are defined as follows. The *total batched response length*

$$\mathsf{tbrlen}_{k,\alpha}(\mathsf{DS}) = \mathsf{trlen}(\mathsf{DS}) + \sum_{q \in \mathbb{Q}_{\mathsf{DS}}} \alpha - \big(|\mathsf{qu}(\mathsf{DS}, q)|_w \mod \alpha\big)$$

reveals the number of batches needed to store the responses in the structure. The *repeated query equality* pattern

$$\mathsf{rqeq}_{k,m}(\mathsf{DS}, q_1, \ldots, q_t) = \begin{cases} \perp & \text{if } m < t \text{ and } q_i \neq q_j \text{ for all } i, j \in [t], \\ \gamma_m & \text{if } m = t \text{ and } q_i \neq q_j \text{ for all } i, j \in [t], \\ \mathsf{qeq} \times \mathsf{rlen}(\mathsf{DS}, q_1, \ldots, q_t) & \text{otherwise,} \end{cases}$$

where

$$\gamma_m \overset{def}{=} \Bigg(\sum_{i \in [m]} |\mathsf{qu}(\mathsf{DS}, q_i)|_w + \alpha - \big(|\mathsf{qu}(\mathsf{DS}, q_i)|_w \mod \alpha\big)\Bigg) \cdot \alpha^{-1} - (m - 1).$$

Note that the non-repeating sub-pattern of rqeq is uniq where

$$\mathsf{uniq}_{k,m}(\mathsf{DS}, q_1, \ldots, q_t) = \begin{cases} \perp & \text{if } m < t \text{ and } q_i \neq q_j \text{ for all } i, j \in [t], \\ \gamma_m & \text{if } m = t \text{ and } q_i \neq q_j \text{ for all } i, j \in [t]. \end{cases}$$

The *add length* pattern

$$\mathsf{alen}_{k,m}(\mathsf{DS}, u_1, \ldots, u_t) = \begin{cases} \perp & \text{if } m < t, \\ \gamma_m & \text{if } m = t, \end{cases}$$

reveals nothing until the last add of the sequence, and then reveals the number of batches required to finish the add sequence.

When PBS is modified to support only probabilistic correctness for queries, the non-repeating sub-pattern of its query leakage is $\perp$. The leakage profile of the probabilistic variant of PBS is therefore $(\mathcal{L}_\mathsf{S}^\mathsf{pbs}, \mathcal{L}_\mathsf{Q}^\mathsf{pbs}, \mathcal{L}_\mathsf{U}^\mathsf{pbs}) = (\mathsf{tbrlen}, \mathsf{patt_Q}, \mathsf{alen})$ where

$$\mathsf{patt_Q}(\mathsf{DS}, q_1, \ldots, q_t) = \begin{cases} \perp & \text{if } q_i \neq q_j \text{ for all } i, j \in [t], \\ \mathsf{qeq} \times \mathsf{rlen}(\mathsf{DS}, q_1, \ldots, q_t) & \text{otherwise.} \end{cases}$$

Safe extension for PBS. Let $(\widetilde{q}_1, \cdots, \widetilde{q}_\lambda)$ be dummy queries. For all $1 \le i \le \lambda$, compute $\overline{\mathsf{DS}} \leftarrow \mathsf{Add}(\overline{\mathsf{DS}}, (\widetilde{q}_i, \mathbf{0}))$, where $|\mathbf{0}|_w = \max_{r \in \mathbb{R}_\mathsf{DS}} |r|_w$.

Theorem 3.4.5. *If λ and α are publicly-known parameters and if all queries in $\mathbb{Q}_\mathsf{DS}$ have the same bit length, the extension scheme described above is* $(\mathsf{tbrlen}, \mathsf{uniq}, \mathsf{alen})$*-safe.*

Proof. In order to show that $\mathsf{patt}_\mathsf{S}^\mathsf{ds}(\overline{\mathsf{DS}}) \le \mathsf{patt}_\mathsf{S}^\mathsf{ds}(\mathsf{DS})$, note that

$$\begin{aligned} \mathsf{tbrlen}(\overline{\mathsf{DS}}) &= \sum_{r \in \mathbb{R}_\mathsf{DS}} \Big(|r|_w + \alpha - (|r|_w \mod \alpha) \Big) + \lambda \cdot \Big(\mu + \alpha - (\mu \mod \alpha) \Big) \\ &= \mathsf{tbrlen}(\mathsf{DS}) + \lambda \cdot \Big(\mu + \alpha - (\mu \mod \alpha) \Big) \end{aligned} \tag{3.7}$$

where $\mu \overset{def}{=} \max_{r \in \mathbb{R}_\mathsf{DS}} |r|_w$. Then given leakage $\mathsf{patt}_\mathsf{S}^\mathsf{ds}(\mathsf{DS}) = \mathsf{tbrlen}(\mathsf{DS})$ and public parameters λ and α, we can simulate $\mathsf{patt}_\mathsf{S}^\mathsf{ds}(\overline{\mathsf{DS}})$ using Eq.(3.7) with μ set to $\mathsf{mllen}(\mathsf{DS}) - |q|$. For query leakage, to show that $\mathsf{uniq}(\overline{\mathsf{DS}}, q_1, \ldots, q_t) \le \mathsf{uniq}(\mathsf{DS}, q_1, \ldots, q_t)$, observe that the output of uniq is the same over DS or $\overline{\mathsf{DS}}$ and therefore simulation is trivial. Similarly, for the add leakage, the output of alen over DS and $\overline{\mathsf{DS}}$ is the same and therefore simulation is trivial. $\square$

Dynamic AZL. Let dynamic AZL be the perfectly-correct fully-dynamic rebuildable scheme that results from applying our framework to the perfectly-correct variant of PBS. Its security is proved in the following Theorem.

Theorem 3.4.6. *If Σ_DX is Λ_DX-secure where $\Lambda_\mathsf{DX} = (\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{G}, \mathcal{L}_\mathsf{P}) = (\mathsf{mllen}, \perp, \perp)$, then dynamic* AZL *is Λ_AZL-secure where*

$$\begin{aligned} \Lambda_\mathsf{AZL} &= (\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{O}, \mathcal{L}_\mathsf{R}) \\ &= \big((\mathsf{tbrlen}, \mathsf{mllen}), \mathsf{uniq}', (\mathsf{lsize}, \mathsf{tbrlen}, \mathsf{olsize}, \mathsf{omllen}, \mathsf{otbrlen}) \big) \end{aligned}$$

where $\mathsf{otbrlen}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_\lambda) = \mathsf{tbrlen}_{k,\alpha}(\mathsf{DS}_\lambda)$ *and*

$$\mathsf{uniq}'_{k,m}(\mathsf{DS}, \mathsf{op}_1, \ldots, \mathsf{op}_t) = \mathsf{uniq}_{k,m}(\mathsf{DS}, q_1, \ldots, q_t),$$

where op_i *is either a query* q_i *or an update* $u_i = (q_i, r_i)$*.*

Proof. To complete the proof we note that the following leakages can be expressed in terms of leakage patterns: (1) any sequence of operations for PBS is compiled into a sequence of non-repeating queries and therefore the operation leakage is uniq' – the total number of batches to retrieve the responses; and (2) from Theorem 3.4.1, the rebuild leakage consists of the following leakage patterns: the setup leakage for the empty data structure DS_0 is $\perp$. Then $\mathsf{patt}_1(\mathsf{DS}) = \big(\mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}\big)$; the query leakage for all unique queries is the non-repeating sub-pattern of rqeq which is the total number of batches to retrieve responses for all the queries, tbrlen. Then, $\mathsf{patt}_2(\mathsf{DS}) = \mathsf{tbrlen}$; and, the update leakage for adding all the responses to the data structure DS_0 is alen applied to all the responses, which is the total batch response length $\mathsf{otbrlen}$ and therefore $\mathsf{patt}_3(\mathsf{DS}) = \mathsf{otbrlen}$. ☐

Efficiency of dynamic AZL. It follows from Equation (3.4) that the complexity of dynamic AZL when Σ_{DX} is initialized with a tree-based ORAM is

$$\mathsf{time}^{\mathsf{azl}}_{\lambda\mathsf{O+R}} = (\lambda + \#\mathbb{Q}_{\mathsf{DS}}) \cdot \mathsf{time}^{\mathsf{pbs}}_{\mathsf{Q}} + O\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda}} |r|_w \cdot \log^2 \lambda\right)$$
$$+ O\left(\#\mathbb{Q}_{\mathsf{DS}} \cdot \max_{r \in \mathbb{R}_{\mathsf{DS}_\lambda} |r|_w} \cdot \log^2 \#\mathbb{Q}_{\mathsf{DS}_\lambda}\right),$$

where $\mathsf{time}^{\mathsf{pbs}}_{\mathsf{Q}}$ is the query complexity of PBS which is equal to the query complexity of is underlying multi-map encryption scheme. The storage complexity of dynamic AZL is the sum of the storage required for the cache and the storage required for the PBS structure. This results in storage complexity

$$O\bigg(\lambda \cdot (\alpha + \max_{a \in \mathsf{Log}(\mathsf{DS}_\lambda)} |a|_w) + \#\mathbb{Q}_{\mathsf{DS}} \cdot (\alpha + \max_{r \in \mathbb{R}_{\mathsf{DS}}} |r|_w)\bigg).$$

Dynamic FZL. Dynamic FZL is the probabilistically-correct fully-dynamic scheme that results from applying our framework to the probabilistically-correct variant of PBS. Its security is analyzed in the following Theorem.

Theorem 3.4.7. *If* Σ_{DX} *is* Λ_{DX} *where* $\Lambda_{\mathsf{DX}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{G}}, \mathcal{L}_{\mathsf{P}}) = (\mathsf{mllen}, \perp, \perp)$*, then dynamic* FZL *is* Λ_{FZL}*-secure where*

$$\Lambda_{\mathsf{FZL}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{O}}, \mathcal{L}_{\mathsf{R}}) = \left((\mathsf{tbrlen}, \mathsf{mllen}), \perp, (\mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}, \mathsf{otbrlen})\right).$$

Proof. To complete the proof we note that the following leakages can be expressed in terms of leakage patterns as follows: (1) the operation leakage uniq' is $\perp$ applied to any operation, which is also $\perp$ (2) from Theorem 3.4.3, the rebuild leakage consists of the following: the setup leakage for the data structure DS_λ is $\perp$. Then, $\mathsf{patt}_1(\mathsf{DS}) = \big(\mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}\big)$; the query leakage for all unique queries is $\perp$. Then, $\mathsf{patt}_2(\mathsf{DS}) = \perp$; and, the edit leakage for editing all the responses for the data structure DS_λ is the total batch response length $\mathsf{otbrlen}$ and therefore $\mathsf{patt}_3(\mathsf{DS}_\lambda) = \mathsf{otbrlen}$. ☐

Efficiency of dynamic FZL. The efficiency of dynamic FZL is the same as that of dynamic AZL.

Our AVLH-Based Construction

We now apply our framework to the mutable variant of the advanced volume-hiding multi-map encryption scheme AVLH^d from [47]. Note that here we do not consider the variant that exploits concentrated components for storage improvements.

Overview of AVLH. At a high level, the scheme uses n bins to store a multi-map of size N, where N is the sum over all labels of the labels' tuple lengths. The scheme uses a random bipartite graph to map labels to bins. More precisely, each label ℓ is mapped at random to t out of n bins, where t is the maximum tuple length. The elements of the tuple corresponding to a label ℓ are placed in each bin mapped to ℓ. If there are more bins mapped than the length of the tuple, some bins are left empty. The bins are then padded to the size of the maximum bin, encrypted and stored on the server. To query for a label ℓ, the client retrieves all the bins mapped to ℓ. The scheme hides the tuple lengths, i.e., the response length rlen. It also supports restricted edits in the sense that one can edit/change the values in a tuple but not add values to it.

The leakage profile of AVLH^d is

$$\Lambda_{\mathsf{AVLH}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{E}}) = (\mathsf{trlen}, \mathsf{qeq}, (\mathsf{oid}, \mathsf{uqeq})).$$

Extension. Let $(\widetilde{q}_1, \cdots, \widetilde{q}_\lambda)$ be dummy queries and $(\widetilde{r}_1, \cdots, \widetilde{r}_\lambda)$ be the corresponding dummy responses such that $|\widetilde{r}_i| = 1$. For all $i \in [\lambda]$, compute $\overline{\mathsf{MM}} \leftarrow \mathsf{Add}(\overline{\mathsf{MM}}, (\widetilde{q}_i, \widetilde{r}_i))$. We prove the security of this extension in the Theorem below.

Theorem 3.4.8. *If λ is a publicly-known parameter and that all queries in the query space $\mathbb{Q}_{\mathsf{DS}}$ have the same bit length, the above extension scheme is $(\mathsf{trlen}, \perp, (\mathsf{oid}, \mathsf{uqeq}))$-safe.*

Proof. For setup leakage, to show that $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\overline{\mathsf{DS}}) \leq \mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS})$ we note that $\mathsf{trlen}(\overline{\mathsf{DS}}) = \mathsf{trlen}(\mathsf{DS}) + \lambda \cdot 1 = \mathsf{trlen}(\mathsf{DS}) + \lambda$. Given the public parameter λ and the $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\mathsf{DS}) = \mathsf{trlen}$ we can then simulate $\mathsf{patt}^{\mathsf{ds}}_{\mathsf{S}}(\overline{\mathsf{DS}})$. For query leakage, $\perp$ is trivially the same over DS and $\overline{\mathsf{DS}}$. For edit leakage, the output of $(\mathsf{oid}, \mathsf{uqeq})$ is the same over DS and $\overline{\mathsf{DS}}$ and therefore the simulation is trivial. $\square$

Safe placeholder. Since AVLH^d is mutable we define a safe placeholder multi-map $\widetilde{\mathsf{MM}}$. Note that the placeholder must have the following properties:

1. $\widetilde{\mathsf{MM}}$ must have enough space to hold the tuples of all the labels $\ell \in \mathbb{L}_{\mathsf{MM}_\lambda}$[5];

2. the setup, query and edit leakages on $\widetilde{\mathsf{MM}}$ must be at most the setup, query and edit leakages on MM.

[5]For any multi-map data structure MM, the query space $\mathbb{Q}_{\mathsf{DS}}$ is the label space $\mathbb{L}_{\mathsf{MM}}$.

The placeholder structure is created as follows during rebuilds. During the extract-and-tag phase, the client learns which labels are valid and their tuple lengths. During the update phase it creates, for every valid label ℓ a dummy tuple $\mathbf{t}$ of the same length and inserts $(\ell, \mathbf{t})$ in $\widetilde{\mathsf{MM}}$. We prove the security of the placeholder in the Theorem below.

Theorem 3.4.9. *The placeholder above is* $(\mathsf{trlen}, \mathsf{qeq}, (\mathsf{oid}, \mathsf{uqeq}))$*-safe.*

Proof. To simulate the setup leakage trlen for MM_λ note that $\mathsf{trlen}(\widetilde{\mathsf{MM}}) = \sum_i \#\mathsf{MM}_\lambda[\ell_i] = \mathsf{trlen}(\mathsf{MM}_\lambda)$. The query and edit leakages are the same over both $\widetilde{\mathsf{MM}}$ and MM_λ and they have the same label space $\mathbb{L}_{\mathsf{MM}_\lambda}$. Then simulation is trivial and the placeholder is $(\mathsf{trlen}, \mathsf{qeq}, (\mathsf{oid}, \mathsf{uqeq})$-safe. □

Zero-leakage advanced volume-hiding. Let ZAVLH be the dynamic rebuildable multi-map encryption scheme that results from applying our framework to AVLH^d with the above placeholder structure and a dictionary encryption scheme Σ_{DX} with leakage profile $\Lambda_{\mathsf{DX}} = (\mathcal{L}_{\mathsf{S}}^{\mathsf{dx}}, \mathcal{L}_{\mathsf{G}}^{\mathsf{dx}}, \mathcal{L}_{\mathsf{P}}^{\mathsf{dx}}) = (\mathsf{mllen}, \perp, \perp)$. Theorem 3.4.10 below proves the security of ZAVLH.

Theorem 3.4.10. *If* Σ_{DX} *is* Λ_{DX}*-secure, then* ZAVLH *is* Λ_{ZAVLH}*-secure where*

$$\Lambda_{\mathsf{ZAVLH}} = (\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{O}}, \mathcal{L}_{\mathsf{R}}) = ((\mathsf{trlen}, \mathsf{mllen}), \perp, (\mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}, \mathsf{otrlen}))\,.$$

Proof. To complete the proof we note that the following leakages can be expressed in terms of leakage patterns as follows: (1) for the operation leakage, uniq' is $\perp$ applied to any operation, which is also $\perp$; and (2) the rebuild leakage consists of the following: the setup leakage for the data structure DS_λ is the total response length after λ operations, otrlen. Then from Theorem 3.4.3, $\mathsf{patt}_1(\mathsf{DS}) = \big(\mathsf{lsize}, \mathsf{olsize}, \mathsf{omllen}, \mathsf{otrlen}\big)$; the query leakage for all unique queries is $\perp$. Then, $\mathsf{patt}_2(\mathsf{DS}) = \perp$; and, since there are olsize unique edit operations for the data structure DS_λ the leakage is then $\mathsf{patt}_3(\mathsf{DS}_\lambda) = \mathsf{olsize}$. Note that the operation leakage oid can be simulated from olsize and is therefore not part of patt_3. □

Efficiency of ZAVLH. We now analyze the efficiency of our dynamic cache-based compiler with a tree-based cache and the AVLH^d scheme. The query complexity for ZAVLH is

$$\mathsf{time}_{\mathsf{Q}}^{\mathsf{zavlh}} = O(t \cdot N/n)$$

If $t = O(1)$ and $n = O(N/\log N)$ where t is the maximum tuple length and n is the number of bins, the query complexity is $O(\log N)$ for zero-leakage operations. From Equation (3.4) we have,

$$\mathsf{time}_{\lambda\mathsf{O}+\mathsf{R}}^{\mathsf{zavlh}} = O\left(\#\mathbb{L}_{\mathsf{MM}} \cdot \log N\right) + O\left(\lambda \cdot \max_{r \in \mathbb{R}_{\mathsf{MM}_\lambda}} |r|_w \cdot \log^2 \lambda\right) \tag{3.8}$$

$$+ O\left(\#\mathbb{L}_{\mathsf{MM}} \cdot \max_{r \in \mathbb{R}_{\mathsf{MM}_\lambda}} |r|_w \cdot \log^2 \#\mathbb{L}_{\mathsf{MM}_\lambda}\right) \tag{3.9}$$

3.4.4 Concrete Comparisons

In Section 3.4.2, we showed that our framework can asymptotically outperform black-box ORAM simulation under natural assumptions on the data and queries. In this section, we are interested in gaining a better understanding of the practical gains in different settings. Specifically, we compare the concrete efficiency of our ZAVLH scheme to an oblivious multi-map constructed via black-box ORAM simulation and to a standard dynamic encrypted multi-map called $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$ [26]. Since the latter has optimal storage and query complexities, this comparison highlights the cost of leakage suppression.

Parameters and notation. For our comparison, we consider a multi-map MM with t labels and $N = \sum_{\ell \in \mathbb{L}_{\mathsf{MM}}} \#\mathsf{MM}[\ell]$ total values and maximum tuple length l. After λ Add operations on MM, the resulting multi-map is denoted MM_λ. We denote the number of labels in MM_λ as t_λ and the total values in MM_λ as N_λ. The maximum tuple size in MM_λ is denoted by l_λ. All PRF keys and outputs are of length $k = 256$ bits, all values in the multi-maps are 64 bits and N is set to 2^{16}.

Parameters	**Setting 1**	**Setting 2**	**Setting 3**	**Setting 4**
General:				
length of PRF output (bits)	256	256	256	256
length of MM value (bits)	64	64	64	64
cache size (λ)	64	64	64	64
MM:				
max. tuple length (l)	512	512	512	512
total # of labels (t)	256	256	256	256
total # of values (N)	2^{16}	2^{16}	2^{16}	2^{16}
total # of AVLH bins (n)	8192	8192	8192	8192
Updated MM_λ:				
max. tuple length (l_λ)	512	512	512	512
total # of labels (t_λ)	256	256	256	256
total # of values (N_λ)	65600	65600	65600	65600
total # of AVLH bins (n_λ)	8199	8199	8199	8199
Upper-bound MM_*:				
factor of growth	25	50	150	1000
max. tuple length (l_*)	1.28×10^4	2.56×10^4	7.68×10^4	51.2×10^4
total # of labels (t_*)	0.64×10^4	1.28×10^4	3.84×10^4	25.6×10^4
total # of values (N_*)	163.84×10^4	327.68×10^4	983.04×10^4	6553.6×10^4

Table 3.1: Parameters for the efficiency comparison of dynamic CBC, black-box ORAM simulation, and $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$, given a multi-map MM and a sequence of λ add operations.

Parameters for ZAVLH. The number of bins in AVLH are chosen such that each bin contains $(\log N)/2$ values on average. The tree-based cache used in the dynamic CBC is instantiated with Path ORAM with λ leaf nodes; one for each tuple in the cache. Each block is initialized to hold one tuple and therefore $(l + \lambda)$ values at most. Each node/bucket in the binary tree holds $Z = 5$ blocks.

Efficiency Measure	ZAVLH (OPS)	ZAVLH (E&T)	ZAVLH (S&S)	ZAVLH (UP)	ZAVLH (Total)	Path ORAM (MM_*)	Std EMM ($\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$)
Client State (Mbits)	0.401	0.084	-	0.401	0.486	4.78 10.058 32.539 244.137	0.066
Server Storage (Mbits)	29.704	14.352	-	29.71	44.062	52424.704 209707.008 1887412.224 83885916.16	20.992
Communication (Mbits)	166.739	211.042	1181.008	268.294	1827.084	1995.534 4306.721 14421.059 113419.012	10.485
Leakage	l, N	t	t_λ	l_λ, N_λ	l, N, t $l_\lambda, N_\lambda, t_\lambda$	l_*, t_*	vol, qeq

Table 3.2: Concrete efficiency comparison. The efficiency numbers shown for ORAM correspond to each of the 4 settings for the ORAM upper-bound data structure.

The position map maps every label to a leaf node in the ORAM and has size $\lambda(k+\log\lambda)$. The stash stores at most $\log\lambda$ blocks and therefore $\log\lambda(l+\lambda)$ values. A query to the cache reads and writes a path of $\log\lambda$ buckets in the tree. The multi-map MM stores $t+\lambda$ labels and $N+\lambda$ total values. We summarize the cost of ZAVLH in Table 3.2 breaking it down into the cost to execute λ operations (OPS) and the costs of the different rebuild phases: extract-and-tag (E&T), sort-and-shuffle (S&S) and update (UP).

Black-box ORAM simulation. To manage the dynamic multi-map MM with Path ORAM, we initialize an *upper-bound* structure MM_* with t_* labels and N_* values.[6] Specifically, we use upper-bound structures that are $25, 50, 150$, and 1000 times larger than the multi-map's original size (Table 3.1). The maximum length of a tuple in MM_* is l_*. The Path ORAM that manages MM_* has t_* leaf nodes, one for each label in MM_*. Each block is initialized to hold l_* values and each node/bucket in the binary tree holds $Z = 5$ blocks. This ORAM has a position map of size $t_*(q+\log t_*)$ and a stash that holds at most $\log t_*$ blocks at any given time.

Comparison. Table 3.2 shows the costs in Mbits for each of the 4 settings for ZAVLH, black-box ORAM simulation, and $\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$. We can see that ZAVLH out-performs black-box ORAM simulation in both space and communication for our chosen parameters. In particular, the storage cost of ZAVLH is 3 to 7 orders of magnitude smaller than black-box ORAM simulation and only a factor of 2 larger than $\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$. We also observe that the communication cost of ZAVLH is up to 60 times smaller

[6]This is due to Path ORAM's inability to resize.

than black-box ORAM simulation, but 180 times larger than $\Pi^{\mathsf{dyn}}_{\mathsf{bas}}$ which is optimal but incurs more leakage.

Efficiency Measure	ZAVLH (OPS)	ZAVLH (E&T)	ZAVLH (S&S)	ZAVLH (UP)	Path ORAM (MM_*)	Std EMM ($\Pi^{\text{dyn}}_{\text{bas}}$)
Client State	$2k(t+\lambda)+$ $(k+\log\lambda)\lambda$ $+(l+\lambda)v\log\lambda$	$(k+\log l_\lambda)\cdot$ $(t_\lambda+\lambda)$	—	$2k(t_\lambda+\lambda)+$ $(k+\log\lambda)\lambda$ $+(l_\lambda+\lambda)v\log\lambda$	$(k+\log t_*)t_*$ $+l_* v\log t_*$	$(k+\log t_*)t_\lambda$
Server Storage (ϕ)	$((N+\lambda)v+kn)$ $+5v(2\lambda-1)$ $(l+\lambda)$	$(t+2\lambda)\cdot$ $(2k+(l+\lambda)v)$	—	$((N_\lambda+\lambda)v+kn_\lambda)$ $+5v(2\lambda-1)$ $(l_\lambda+\lambda)$	$5l_*v(2t_*-1)$	$N_\lambda(k+v)$
Communication (ψ)	$\lambda\big(lk+$ $l((N+\lambda)v/n)$ $+10v(l+\lambda)\log\lambda\big)$	$l(t+\lambda)\cdot$ $(k+$ $(N+\lambda)v/n)$ $+5v(l+\lambda)\cdot$ $\lambda\log\lambda$ $+\phi^{(\text{E\&T})}$	$4(2k+$ $(l+\lambda)v)$ $\cdot S^{t+2\lambda}$	kn_λ $+\log((N_\lambda+\lambda)/n_\lambda)$ $+2l_\lambda(t_\lambda+\lambda)\cdot$ $\big(k+(N_\lambda+\lambda)v/n_\lambda\big)$ $+\phi^{(\text{E\&T})}$	$\lambda(10vl_*\log t_*)$	$\lambda l_\lambda(k+v)$
Leakage	l, N	t	t_λ	l_λ, N_λ	l_*, t_*	vol,qeq

Table 3.3: Calculations for efficiency comparison using a sequence of λ add operations.

In Table 3.3 we show the expressions used to calculate the results in Table 3.2. We recall that our scheme ZAVLH has two phases for any sequence of λ operations: (1) the operations themselves (OPS); and (2) the different rebuild phases : (2a) extract-and-tag (E&T), sort-and-shuffle (S&S), and (2c) update (UP). We now present a brief explanation of the expressions used to compute the efficiency costs:

1. **OPS.** During the λ operations, the client maintains state for the multi-map as well as the state for the tree-ORAM based cache with λ entries:

   - The stash for the cache ORAM has at most $\log\lambda$ blocks. Each block has $(l+\lambda)$ values of bit length v. Then the ORAM stash is of size $(l+\lambda)v\log\lambda$ bits.
   - The position map for the ORAM cache maps each leaf in the tree to a label. Then there are λ entries consisting of a label and a leaf in the tree. A label is k bits long and a leaf can be identified with $\log\lambda$ bits. Then the position map is of size $(k+\log\lambda)\lambda$.
   - The client also maintains $2k$ state per label in the multi-map, which can have at most $(t+\lambda)$ values.

   The server stores all the entries for both the multi-map and the cache.

   - The multimap has at most $(N+\lambda)$ values of length v and n labels of length k each. Then the total size of the multimap in bits will be $(N+\lambda)v+kn$.
   - The cache consists of a tree with $2\lambda-1$ nodes. Each node contains 5 blocks, and each block contains at most $(l+\lambda)$ values of length v bits each. Then the total size of the cache in bits will be $5v(2\lambda-1)(l+\lambda)$.

   Communication per operation consists of reading and writing a path in the cache, as well as querying for a label and receiving the corresponding values.

- For the cache, each path is $\log \lambda$ nodes long, each node has 5 buckets and each bucket has $(l + \lambda)$ values of length v bits each. Reading reads a path, and writing writes a path in the ORAM. Then the total cache communication will be $10v(l + \lambda) \log \lambda$.
- Since the underlying multi-map scheme is AVLH, a query to the multi-map consists of l bin labels and a response of l bin contents. Each bin has at most $(N + \lambda)/n$ values where n is the number of bins. Then the total multi-map communication is $lk + lv((N + \lambda)/n)$ bits.
- Each of the above steps must be repeated for every operation and therefore the cost is multiplied by λ to get the final communication in bits.

This phase leaks both the maximum number of values corresponding to one label in the multi-map l and the total number of values in the multi-map N. Note that λ is a public parameter and therefore known to the adversary.

2. **E&T.** In this phase, the client queries all the labels in the multi-map. Then for each label the client creates a RAM entry at the server side.

- For each of the labels the client has to store the (updated) number of values. There are $(t_\lambda + \lambda)$ labels in the updated multi-map (including dummies) and each of them could have up to l_λ values.
- For each label, the client needs to store the label, and the new length, which would be $k + \log l_\lambda$ bits for each i.e. $(k + \log l_\lambda) \cdot (t_\lambda + \lambda)$ bits total.

The server stores the RAM. Each entry of the RAM contains: a label, a freshness value, and is padded to $(l + \lambda)$ values. Then the total server storage is $(2k + (l + \lambda)v)$ for each label (including dummies). The RAM is also padded to have $(t + 2\lambda)$ labels. Then the total server storage is $(t + 2\lambda)(2k + (l + \lambda)v)$ bits. The communication of constructing the RAM is that of (1) querying all the labels in the cache followed by all the labels in the multi-map, (2) sending each of the entries of the RAM to the server. The cost of step (2) is the same as the server storage $\phi^{(\text{E\&T})}$. For step (1):

- Since we query all the labels in the cache, we need not read and write to cache for every operation. We can merely read every path in the tree. Then each path is of size $5v(l+\lambda) \log \lambda$ bits, as shown for previous operations. There are λ such paths, so the total communication cost is $5v(l + \lambda)\lambda \log \lambda$ bits.
- Querying the multi-map consists of sending the $t + \lambda$ original labels (including dummies) and receiving the results for each. As shown earlier for AVLH, each of these queries costs $l(k + (N + \lambda)v/n)$ bits and there are a total of $(t + \lambda)$ queries.
- Combining the above two expressions, we have the resulting communication cost in bits.

The only additional leakage in this phase is the number of labels in the multi-map, t.

3. **S&S.** There is no persistent client state, or extra server storage required in this phase. For each gate of the oblivious sort circuit, two RAM entries are sent back and forth between the client and the server. Then the total communication cost is 4 times the size of one RAM entry, multiplied by the number of gates in a circuit to obliviously sort $t + 2\lambda$ elements. We have seen previously that one RAM entry consists of $(2k + (l + \lambda)v)$ bits. Then the total communication cost is $4(2k + (l + \lambda)v) \cdot S^{t+2\lambda}$ where $S^{t+2\lambda}$ is the number of gates in the oblivious sorting circuit for $(t + 2\lambda)$ elements. Only the new number of labels t_λ is leaked in this phase, since the RAM is truncated at $t_\lambda + \lambda$.

4. **UP.** Since the update phase consists of queries to the new multi-map, just like the (OPS) stage, the client and server state are exactly the same except with the parameters of the new multi-map (subscripted with λ). The communication costs have to account for (1) setting up a placeholder structure at the server, (2) reading every entry in the RAM and running the corresponding update on the placeholder structure.

    - Every entry in the RAM is read once by the client. Then this cost is the same as the size of the RAM, or $\phi^{(\text{E\&T})}$.
    - To set up a placeholder structure in AVLH, the server need only receive the labels, and the size of the bins. There are n_λ labels of size k bits each. And the bins have size less than $(N_\lambda + \lambda)/n_\lambda$. Then the communication required is $kn_\lambda + \log((N_\lambda + \lambda)/n_\lambda)$ bits.
    - For each update, a label must be sent and l_λ bins must be read and written. Given the size of the bins, the total communication would be $2l_\lambda(t_\lambda + \lambda) \cdot (k + (N_\lambda + \lambda)v/n_\lambda)$.

    This phase leaks the updated parameters t_λ, N_λ of the new multi-map.

5. **Black-box ORAM simulation.** The ORAM simulation using Path ORAM with the upper-bound structure MM_* has the same client state, server state and communication cost as the tree-ORAM based cache in (OPS), except that the parameters l, t are replaced by l_*, t_* which are the parameters of MM_*. The ORAM construction would leak only the upper-bound parameters.

6. **Standard EMM ($\Pi^{\mathbf{dyn}}_{\mathbf{bas}}$).** The standard dynamic EMM has the following costs:

    - The client state consists of one counter per label that keeps track of the length of the tuple. Therefore for a dynamic EMM that holds t_λ labels the client state would be $k + \log t_*$ bits per label where t_* is the upper-bound on the length of a tuple.
    - The server storage would consist of N_λ label-value pairs, each pair consisting of $(k + v)$ bits.
    - The communication for each operation would be l_λ label-value pairs, and the total communication across λ operations would therefore be $\lambda l_\lambda(k + v)$ bits.

3.5 Our Techniques for Efficient Leakage Suppression

In the previous sections, we have seen that our dynamic query equality suppression framework results in zero-leakage STE schemes that are asymptotically more efficient than black-box ORAM simulation. However, our schemes are still significantly inefficient when compared to standard (but leaky) STE schemes. In this section, we present new techniques for leakage suppression which are practically efficient and much closer in efficiency to standard constructions. We initiate a line of work in leakage suppression that will increase practical efficiency significantly. In the rest of this section, we present a static dictionary transform that suppresses the query equality leakage, and use the transform to design an encrypted dictionary scheme. Our dictionary transform QRT uses *replicated* label-value pairs in order to suppress the query equality leakage of a dictionary. The transform adds replicated label-value pairs to the input dictionary and transforms every repeating query to the dictionary to a query for a unique replica, thereby suppressing the query equality. However, the transform has a limited query capacity, and it can only support up to some fixed number of queries. We show how to overcome this limitation using a rebuild protocol for encrypted dictionaries and design a static dictionary scheme RPL that supports optimal query complexity while also suppressing the query equality.

3.5.1 Query Replication

We introduce the technique of *query replication*, or making copies of the queries, in order to suppress the query equality pattern. As a first attempt at using replication to hide query equality, a replication transform could simply create replica label-value pairs in the dictionary, and later query the replica labels. For example, consider a label ℓ with its corresponding value v replicated as two replica label-value pairs (ℓ_1, v_1) and (ℓ_2, v_2), where $v_1 = v_2 = v$. Both the label-value replicas will be added to the plaintext dictionary during setup. At query time, when ℓ is queried for the first time, the query is transformed to ℓ_1 and retrieves v_1. Later, when ℓ is queried a second time, the query is now transformed to ℓ_2 and retrieves v_2. The two queries are equal, but they are transformed to two unique queries for different (equivalent) labels and each of them retrieves a different encrypted value. Therefore, the query equality leakage between the two queries is suppressed by the transform.

Size constraints. However, what if ℓ is queried a third, or even a fourth time? In order to continue suppressing the query equality, the transform would need to add more replicas in the dictionary. As the number of queries grows, the number of replicas required to suppress the query equality would also therefore grow. This would increase the size of the transformed dictionary output by the replication transform. If the dictionary could be unbounded, the transform could, in theory, create a very large number of replicas. Every query would then retrieve a previously unused replica, and the query equality would always be suppressed. However, in practice, the size of the output dictionary is bounded, and therefore the number of replicas that can be added is also bounded. Since these replicas are used to hide the query equality, the transform can then only support a limited number of queries before it runs out of replicas. Our transform captures this constraint by requiring two public parameters γ and δ, which control both the size of the transformed dictionary, and the number of queries that can be supported with query equality suppression.

Replication distribution. Given the constraints on the size of the replicated dictionary and the number of supported queries, how best can the transform utilize the replicas in order to support the most number of queries correctly? In other words, how must each label be replicated in order to support the most number of queries? Intuitively, the queries that are most popular will be repeated the most, and therefore will require the most query equality suppression. It is therefore reasonable to replicate the most popular queries the most number of times. In other words, given the client's potential query distribution, the transform can use this information to replicate each label-value pair proportional to the probability that it will be queried. Therefore the transform also takes as additional input a probability distribution over the label space of the dictionary, which we refer to as the *replication distribution.* This distribution is then used to determine the number of replicas that will be created for each label-value pair in the transformed dictionary. At query time, each time a label is queried, the query will be transformed to query an unused replicated label-value pair, thereby suppressing the query equality.

Dummy labels. However, the replication distribution need not be the same as the eventual distribution that is used to query the transformed dictionary. This situation can arise due to incomplete or incorrect information about the query distribution. For instance, the transform may have created very few replicas for some label ℓ that is frequently queried. Whenever the label ℓ is queried, a replica will be used, and the transform will soon run out of unused replices to query. However, we require the transform to provide security in the worst case, and suppress the query equality, *regardless of the query distribution.* In order to maintain security as required, the transform introduces *dummy* label-value pairs into the dictionary. These pairs contain no valid information, but they are retrieved whenever a queried label does not have any valid replicas remaining. In this manner, every query can still be transformed into a unique query on the transformed dictionary, regardless of the query distribution. Since the dummy values do not contain any information, the response to the query will be incorrect, and hence the transform trades off correctness to preserve security.

3.5.2 The Query Replication Transform

We now formally describe our query replication transform QRT which transforms a dictionary according to a replication distribution. Our transformation effectively transforms the query equality leakage pattern of the dictionary to the identity matrix, i.e., every query to the dictionary is transformed into a unique query on the transformed dictionary. The pseudocode for QRT is described in Figure 3.1.

Overview. QRT is a data structure transformation that takes as input the following: (1) a dictionary DX, (2) a *replication distribution* $\vec{p}$ on the query space of the dictionary, and (3) public parameters γ, δ. The transformation outputs a *replicated dictionary* where each label is replicated proportional to its probability from the replication distribution. The replicated dictionary is generated as follows: first, create an empty dictionary DX' of size $M = (2+\delta)\cdot\gamma$. Now, let p_ℓ be the probability corresponding to label ℓ in the query space $\mathbb{Q}_k$. Create $p_\ell \cdot (1+\delta)\cdot\gamma$ replicas, or copies, of label ℓ and its corresponding value v. Append a per-label counter value to each replicated label. After replicating all the labels present in the input dictionary, create a *dummy* label $\ell_\perp$ and assign to it a corresponding dummy value 0^k. Add γ replicas of $\ell_\perp$ to the replicated dictionary DX'. The transformed dictionary then contains a total of M label-value pairs after replication.

Query transformation. The QRT also transforms queries to the underlying dictionary structure. Every query for a label is transformed into a query for an unused replicated label in the transformed dictionary. At a high level, this also transforms the query equality pattern because an unused replicated value is retrieved for every query, transforming any repeating queries into queries for unique labels in the replicated dictionary. The QRT must maintain some state to support this transformed get operation, namely, both the number of queried replicas for each label, and the total number of queries performed. Therefore the QRT is a *stateful* transform.

Query limit. The QRT must also account for two special cases, due to the fact that the replicated dictionary is bounded in size: (1) when all the replicas of a label in the transformed dictionary have been queried, the transformed query will return a dummy label. Due to this modification, the get operation may not always return the correct value. However, we will show that we can guarantee correctness with high probability if the query distribution is the same as the replication distribution; (2) when the replicated dictionary runs out of replicas, the transformation does not support any further queries. We set the *query limit* (or *epoch*), to γ for the transformed data structure. After γ queries, the transformed dictionary will not support any further get operations.

Efficiency. The *storage overhead* for the replicated dictionary is $M = (2+\delta)\cdot\gamma$, $M \geq (N+\gamma)$, where N is the number of labels in the original dictionary. The transformation maintains a counter of $\log M$ bits for each label in the dictionary, as well as a query counter, and therefore total *state* of $(m+1)\cdot\log M$ bits. For each get on the transformed dictionary, exactly one encrypted value is retrieved and therefore both the *communication complexity* and the *round complexity* of the transformed queries are optimal.

Correctness of QRT

In this section we prove the correctness guarantees of QRT. We show that correctness holds with high probability if the replication distribution and the query distribution over the label space are the same.

Let γ and δ be public parameters, and let $\vec{p}$ be a distribution on the query space of a dictionary DX, with probability p_ℓ corresponding to a label ℓ. Consider the transform QRT defined as follows:

```
• $\mathsf{QRT}(1^k, \gamma, \delta, \vec{p})$:
    – $\mathsf{QRT.Transform}(\mathsf{DX})$
        1. initialize empty dictionary $\overline{\mathsf{DX}}$;
        2. for each label $\ell$:
            (a) compute the number of replicas $r_\ell = p_\ell \cdot (1+\delta) \cdot \gamma$;
            (b) if $\ell$ is present in $\mathsf{DX}$, let $v = \mathsf{DX}[\ell]$;
            (c) else, let $v = 0^k$;
            (d) for $i = 1, 2, \ldots, r_\ell$:
                ∗ set $\overline{\mathsf{DX}}[\ell \,\|\, i] = v$;
        3. select a dummy label $\ell_\perp$;
        4. for $j = 1, 2, \ldots, \gamma$:
            (a) set $\overline{\mathsf{DX}}[\ell_\perp \,\|\, j] = 0^k$;
        5. initialize a dictionary $\mathsf{DX}^c$ and set $\mathsf{DX}^c[\ell] = r_\ell$ for every label $\ell$, and counter $\mathsf{gcnt} = 0$;
        6. output the transformed dictionary $\overline{\mathsf{DX}}$ and the state $st = (\mathsf{DX}^c, \mathsf{gcnt})$.
    – $\mathsf{QRT.Transform}(\ell, st)$
        1. if $\mathsf{gcnt} > \gamma$, return $\perp$;
        2. let $\mathsf{cnt} = \mathsf{DX}^c[\ell]$:
            (a) if $\mathsf{cnt} > 0$, decrement $\mathsf{DX}^c[\ell]$;
            (b) if $\mathsf{cnt} = 0$, set $\ell = \ell_\perp$ and $\mathsf{cnt} = \mathsf{gcnt}$;
        3. increment $\mathsf{gcnt}$;
        4. output $\ell \,\|\, \mathsf{cnt}$.
```

Figure 3.1: The query replication transform: QRT

Theorem 3.5.1. *If both the query distribution and the replication distribution are given by $\{p_\ell\}_{\ell \in \mathbb{Q}_k}$, then* QRT *is ε-correct for up to γ queries where $\varepsilon = m \cdot e^{-\frac{\delta^2 \gamma \cdot \min_\ell p_\ell}{2+\delta}}$.*

Proof. Let ℓ be some label with query probability p_ℓ. Then from the replication distribution, we know that there exist $r_\ell = p_\ell \cdot (1+\delta) \cdot \gamma$ replicas for this label. Then we can compute the probability that the total number of queries in an epoch for this label exceed r_ℓ. Let the queries in the epoch be $q_1, q_2, \ldots, q_\gamma$, and $X_{i,\ell}$ be an indicator variable that is defined as follows:

$$X_{i,\ell} = \begin{cases} 1, & \text{if } q_i = \ell, 1 \leq i \leq \gamma \\ 0, & \text{otherwise} \end{cases}$$

Let $X_\ell = \sum_{i=1}^{\gamma} X_{i,\ell}$. Then X_ℓ is the number of times that ℓ is queried during the epoch, given that the query probability is p_ℓ. Given that there are γ queries in the epoch, the expected number of queries is $p_\ell \cdot \gamma$. Using a Chernoff bound,

$$\Pr[X_\ell \geq p_\ell \cdot (1+\delta) \cdot \gamma] \leq \exp\left(\frac{-\delta^2 \cdot \gamma \cdot p_\ell}{2+\delta}\right)$$

Since $\min_\ell p_\ell \leq p_\ell$, and given that $\exp\left(-\frac{\delta^2 \gamma \cdot \min_\ell p_\ell}{2+\delta}\right) = \varepsilon/m$ it follows:

$$\Pr[X_\ell \geq p_\ell \cdot (1+\delta) \cdot \gamma] \leq \frac{\varepsilon}{m},$$

Finally, using the union bound over all m labels, we see that QRT is ε-correct (for any label). This guarantee holds for the first γ queried labels, after which the QRT will only return $\perp$. $\square$

Security of QRT

When QRT is used in conjunction with a dictionary encryption scheme, we must ensure that the transformation leaks only the public parameters γ, δ of QRT. Then given a leakage profile Λ of a static rebuildable encrypted dictionary scheme, we can define the Λ-safety of QRT as follows:

Definition 3.5.2 (Λ-safe QRT)**.** *Let $\Lambda = (\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q}, \mathcal{L}_\mathsf{R})$ be a static rebuildable dictionary leakage profile. We say that* QRT *is Λ-safe if for all $k \in \mathbb{N}$, for all distributions $\vec{p}$, for all $\gamma > 0$, $\delta \geq 0$, for all $d \in \mathbb{D}_k$, for all $\mathsf{DX} \equiv d$, for all $(\overline{\mathsf{DX}}, st)$ output by* $\mathsf{QRT.Transform}(\mathsf{DX})$*, for all $t \in \mathbb{N}$, for all $(q_1, \ldots, q_t) \in \mathbb{Q}_k^t$, for all $(\overline{q}_1, \ldots, \overline{q}_t)$ where $\overline{q}_i$ is output by* $\mathsf{QRT.Transform}(q_i, st)$*,*

$$\mathcal{L}_\mathsf{S}(\overline{\mathsf{DX}}) \leq \{\mathcal{L}_\mathsf{S}(\mathsf{DX}), \mathsf{params}\} \quad \text{and} \quad \mathcal{L}_\mathsf{Q}(\overline{\mathsf{DX}}, \overline{q}_1, \ldots, \overline{q}_t) \leq \mathcal{L}_\mathsf{Q}(\mathsf{DX}, q_1, \ldots, q_t),$$

$$\text{and} \quad \mathcal{L}_\mathsf{R}(\overline{\mathsf{DX}}) \leq \mathcal{L}_\mathsf{R}(\mathsf{DX}),$$

where $\mathsf{params} = \{\gamma, \delta\}$.

3.5.3 A Query Equality-Suppressed Dictionary Scheme

In this section we use the QRT to design a dictionary encryption scheme RPL with optimal non-amortized query complexity that also suppresses the query equality pattern. In the previous section we observed that the QRT can only support a fixed number of queries, γ. RPL must then somehow increase the number of queries supported by the QRT in order to work around this limitation.

```
• $\mathsf{QRT}(1^k, \gamma, \delta, \vec{p})$:
    – $\mathsf{QRT.Reset}()$
        1. for each label $\ell$:
            (a) compute $r_\ell = p_\ell \cdot (1+\delta) \cdot \gamma$;
            (b) set $\mathsf{DX}^c[\ell] = r_\ell$;
        2. set $\mathsf{gcnt} = 0$.
```

Figure 3.2: Resetting client state for the QRT

Refreshing replicas. In order to be able to support more queries, RPL periodically "refreshes" the encrypted replicas. Once all the existing replicas have been refreshed, the scheme can support the next γ queries from the client. In order to refresh the replicas, we introduce a Rebuild protocol, similar to previous work in query equality suppression [50]. The Rebuild protocol must re-encrypt the encrypted dictionary such that the server cannot correlate the replicas before and after re-encryption. Then given such a static rebuildable dictionary encryption scheme, we can apply the QRT to construct our scheme which suppresses the query equality.

We now describe our dictionary encryption scheme, RPL, which efficiently suppresses the query equality pattern using the QRT. Our scheme uses a blackbox static rebuildable dictionary scheme Σ_{DX} as a building block. At a high level, the client uses the QRT to transform the input dictionary before encrypting it using Σ_{DX} and uploading the encrypted dictionary to the server. At query time, the client's queries are transformed using the QRT before being sent to the server. Finally, after the client issues γ queries, the client and the server execute a protocol to rebuild the encrypted dictionary. Since the QRT supports only a limited number of queries, the rebuild protocol is required to re-initialize the QRT and continue supporting queries on the encrypted structure. For security, we require that the rebuild protocol not leak any correlations between the labels of the dictionary before and after the rebuild protocol is executed. Any correlations leaked during the rebuild process would allow the server to (partially) infer the query equality pattern. The reset procedure for the QRT is given in Figure 3.2. The pseudocode for our scheme is given in Figure 3.3. We now analyse the security and efficiency of RPL.

Security of RPL

Our first theorem shows that our scheme RPL does indeed hide the query equality leakage, when instantiated with a static rebuildable dictionary encryption scheme with no rebuild leakage. We then show how to instantiate RPL with such a dictionary encryption scheme in order to suppress the query equality.

Theorem 3.5.3. *If Σ_{DX} is a response-hiding rebuildable static dictionary encryption scheme which is $(\mathcal{L}_{\mathsf{S}}^{\mathsf{DX}}, (\mathsf{qeq}, \mathsf{patt}), \mathcal{L}_{\mathsf{R}}^{\mathsf{DX}})$-secure, and QRT is $(\mathcal{L}_{\mathsf{S}}^{\mathsf{DX}}, \mathsf{uniq}, \mathcal{L}_{\mathsf{R}}^{\mathsf{DX}})$-safe, where $\mathcal{L}_{\mathsf{R}}^{\mathsf{DX}} = \perp$, then RPL is a static, $(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{G}}, \mathcal{L}_{\mathsf{R}})$-secure dictionary encryption scheme, where $\mathcal{L}_{\mathsf{S}} = \{\mathcal{L}_{\mathsf{S}}^{\mathsf{DX}}, \mathsf{params}\}$, $\mathcal{L}_{\mathsf{G}} = \mathsf{uniq}$, and $\mathcal{L}_{\mathsf{R}} = \perp$ where uniq is the non-repeating sub-pattern of patt and $\mathsf{params} = (\gamma, \delta)$ are the public parameters of the QRT.*

Let $\Sigma_{\mathsf{DX}} = (\mathsf{Setup}, \mathsf{Get}, \mathsf{Rebuild})$ be a response-hiding rebuildable static dictionary encryption scheme, let γ, δ be the public parameters of the QRT, let the total number of labels be m, and $\vec{p}$ be the replication distribution. Then every label ℓ has an associated replication probability p_ℓ. Consider the static rebuildable dictionary encryption scheme $\mathsf{RPL} = (\mathsf{Setup}, \mathsf{Get}, \mathsf{Rebuild})$ defined as follows:

- $\mathsf{Setup}(1^k, \gamma, \delta, \vec{p}, \mathsf{DX})$:
  1. generate a QRT transform of DX by computing
  $$(\mathsf{DX}', st_{\mathsf{QRT}}) \leftarrow \mathsf{QRT}.\mathsf{Transform}(\mathsf{DX});$$
  2. encrypt the transformed dictionary DX' by computing
  $$(K, st_{\mathsf{DX}}, \mathsf{EDX}) \leftarrow \Sigma_{\mathsf{DX}}.\mathsf{Setup}(1^k, \mathsf{DX}');$$
  3. initialize a counter $\mathsf{gcnt} = 0$;
  4. output the key K, the client state $st = (st_{\mathsf{DX}_1}, st_{\mathsf{QRT}}, \mathsf{gcnt})$, and the encrypted dictionary EDX.
- $\mathsf{Get}_{\mathbf{C},\mathbf{S}}((K, st, \ell), \mathsf{EDX})$:
  1. $\mathbf{C}$ checks if $\mathsf{gcnt} > \gamma$ and if true aborts;
  2. $\mathbf{C}$ transforms the query ℓ by computing $\ell' \leftarrow \mathsf{QRT}.\mathsf{Transform}(\ell, st_{\mathsf{QRT}})$;
  3. $\mathbf{C}$ and $\mathbf{S}$ execute $(v, \perp) \leftarrow \Sigma_{\mathsf{DX}}.\mathsf{Get}_{\mathbf{C},\mathbf{S}}\big((K, st_{\mathsf{DX}}, \ell'), \mathsf{EDX}\big)$;
  4. $\mathbf{C}$ increments gcnt.
- $\mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}\left((K, st), \mathsf{EDX}\right)$:
  1. $\mathbf{C}$ and $\mathbf{S}$ generate the rebuilt dictionary by executing
  $$((K', st'), \mathsf{EDX}') \leftarrow \Sigma_{\mathsf{DX}}.\mathsf{Rebuild}_{\mathbf{C},\mathbf{S}}((K, st), \mathsf{EDX});$$
  2. $\mathbf{C}$ resets the client state st_{QRT} by running $\mathsf{QRT}.\mathsf{Reset}()$;
  3. $\mathbf{C}$ sets $\mathsf{gcnt} = 0$.

Figure 3.3: RPL: A dictionary encryption scheme with query equality suppression.

Proof. Let $\mathcal{S}_{\mathsf{DX}}$ be the simulator guaranteed to exist by the adaptive security of Σ_{DX} and let $\mathcal{S}_{\mathsf{QRT}}$ be the simulator guaranteed to exist by the $(\mathcal{L}_{\mathsf{S}}^{\mathsf{DX}}, \mathsf{uniq}, \mathcal{L}_{\mathsf{R}}^{\mathsf{DX}})$-safety of QRT. To prove security, we construct a simulator $\mathcal{S}$ such that the output of $\mathbf{Ideal}_{\mathsf{RPL},\mathcal{A},\mathcal{S}}(k)$ is distributed like the output of a $\mathbf{Real}_{\mathsf{RPL},\mathcal{C},\mathcal{A}}(k)$ experiment. Our simulator $\mathcal{S}$ works as follows:

Simulating Setup: given γ, δ, and leakage $\mathcal{L}_{\mathsf{S}}^{\mathsf{DX}}(\mathsf{DX})$, $\mathcal{S}$ computes

$$\mathsf{EDX} \leftarrow \mathcal{S}_{\mathsf{DX}}\bigg(\mathcal{S}_{\mathsf{QRT}}\bigg(\mathcal{L}_{\mathsf{S}}^{\mathsf{DX}}(\mathsf{DX}), \mathsf{params}\bigg)\bigg),$$

It sets $\mathsf{gcnt} = 0$ and returns EDX to $\mathcal{A}$. It suffices to use the leakage on the input dictionary DX, because the leakage of the replicated dictionary can be simulated using the leakage computed on DX and params.

Simulating Get: given the leakage $\mathsf{uniq}(\mathsf{DX}, g_1, \ldots, g_t)$ for t gets on DX_1, $\mathcal{S}$ works as follows.

If $\mathsf{gcnt} > \gamma$ it aborts. If $\mathsf{gcnt} \leq \gamma$, it uses

$$\mathcal{S}_{\mathsf{DX}}\bigg(\mathcal{S}_{\mathsf{QRT}}\bigg(M_0, \mathsf{uniq}(\mathsf{DX}, g_1, \ldots, g_t)\bigg)\bigg)$$

to simulate a Get to EDX, where M_0 is a $t \times t$ zero matrix. Using M_0 and the non-repeating sub-pattern suffices because all the gets to EDX are unique. After each Get, $\mathcal{S}$ sets $\mathsf{gcnt} := \mathsf{gcnt} + 1$.

Simulating $\mathsf{Rebuild}$: given the rebuild leakage $\mathcal{L}_{\mathsf{R}}^{\mathsf{DX}} = \perp$, $\mathcal{S}$ runs the simulator

$$\mathsf{EDX}' \leftarrow \mathcal{S}_{\mathsf{DX}}\bigg(\mathcal{S}_{\mathsf{QRT}}\bigg(\perp\bigg)\bigg),$$

in order to simulate the rebuild protocol.

It remains to show that the probability that $\mathbf{Ideal}_{\mathsf{RPL},\mathcal{A},\mathcal{S}}(k)$ outputs 1 is negligibly-close to the probability that $\mathbf{Real}_{\mathsf{RPL},\mathcal{C},\mathcal{A}}(k)$ outputs 1 for any PPT adversary $\mathcal{A}$. We do this using the following sequence of games:

Game_0: corresponds to a $\mathbf{Real}_{\mathsf{RPL},\mathcal{C},\mathcal{A}}(k)$ experiment.

Game_1: is the same as Game_0 except that during Setup; EDX is replaced with the output of $\mathcal{S}_{\mathsf{DX}}(\mathcal{L}_{\mathsf{S}}^{\mathsf{DX}}(\overline{\mathsf{DX}}))$ and the ith execution of $\Sigma_{\mathsf{DX}}.\mathsf{Get}$ is replaced with a simulated execution between $\mathcal{S}_{\mathsf{DX}}\big(M_i, \mathsf{uniq}(\overline{\mathsf{DX}}, \overline{g}_1, \ldots, \overline{g}_i)\big)$ and the adversary, where M_i is the $i \times i$ zero matrix, $\overline{\mathsf{DX}}$ is the transformed dictionary and $\overline{g_i}$ the i^{th} transformed get output by the QRT. Since all the gets to EDX are unique in Game_0 it suffices to give $\mathcal{S}_{\mathsf{DX}}$ the leakage $\big(M_i, \mathsf{uniq}(\overline{\mathsf{DX}}, \overline{g}_1, \ldots, \overline{g}_i)\big)$. If $\mathsf{gcnt} = \gamma$; the rebuild protocol is replaced by a simulated interaction of the simulator $\mathcal{S}_{\mathsf{DX}}$ with the rebuild leakage $\mathcal{L}_{\mathsf{R}}$, in order to generate the rebuilt structure EDX'. The probabilities that Game_0 and Game_1 output 1 are negligibly-close, otherwise the adaptive-security of Σ_{DX} would be violated.

Game_2: is the same as Game_1, except the $(\mathcal{L}_{\mathsf{S}}^{\mathsf{DX}}, \mathsf{uniq}, \mathcal{L}_{\mathsf{R}}^{\mathsf{DX}})$ leakage computed on the transformed dictionary $\overline{\mathsf{DX}}$ are replaced by running $\mathcal{S}_{\mathsf{QRT}}$ on the respective leakage of DX and the public parameters params of the QRT. The probabilities that Game_1 and Game_2 output 1 are negligibly-close, otherwise the $(\mathcal{L}_{\mathsf{S}}^{\mathsf{DX}}, \mathsf{uniq}, \mathcal{L}_{\mathsf{R}}^{\mathsf{DX}})$-safety of QRT would be violated.

The Theorem follows by observing that Game_2 is exactly an $\mathbf{Ideal}_{\mathsf{RPL},\mathcal{A},\mathcal{S}}(k)$ experiment. □

In the above theorem, we show that RPL suppresses the query equality leakage when the underlying scheme has no rebuild leakage. We note here that this constraint is slightly stricter than required, since we only need that the correlations between old and new replicas not be leaked during rebuild. We now show how to instantiate such a dictionary encryption scheme using existing constructions from the literature.

Concrete instantiation. In order to instantiate RPL, we compile a semi-dynamic dictionary encryption scheme using the static RBC compiler [50], which results in a static rebuildable dictionary encryption scheme. We use the semi-dynamic (only Add operations are supported) version of the standard dictionary encryption scheme $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$ with the leakage profile $(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{A}}) = (N, \mathsf{qeq}, \perp)$, where N is the total number of labels in the dictionary and qeq is the query equality. We note that since the semi-dynamic scheme does not support deletes on the underlying dictionary, every Add operation must introduce a new label-value pair into the encrypted dictionary, and therefore the Add operation has no leakage. Then, applying the static RBC, we have a resulting static rebuildable scheme Σ_{DX} with the leakage profile $(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{R}}) = (N, \mathsf{qeq}, \perp)$ [50, Theorem 2]. We now show that the QRT is $(N, \mathsf{uniq}, \perp)$-safe and prove the security of the concrete instantiation.

Lemma 3.5.4. *The* QRT *is* $(N, \mathsf{uniq}, \perp)$*-safe.*

Proof. We first note that the setup leakage is $\mathcal{L}_{\mathsf{S}}(\overline{\mathsf{DX}}) = (2 + \delta) \cdot \gamma$, which can be simulated from params. The query leakage is $\mathcal{L}_{\mathsf{Q}}(\overline{\mathsf{DX}}, \overline{q}_1, \ldots, \overline{q}_t) = \mathsf{uniq}$, because the QRT transforms every query to unique queries and therefore the leakage is the same, and query can be simulated from uniq. Finally the rebuild leakage, $\mathcal{L}_{\mathsf{R}} = \perp$, and therefore rebuild can be simulated with no leakage. □

Combining Lemma 3.5.4 with our Theorem 3.5.3, we have that RPL has leakage $(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{G}}, \mathcal{L}_{\mathsf{R}}) = (N, \mathsf{uniq}, \perp)$, and therefore the query equality leakage is suppressed.

Concrete Efficiency of RPL

In this section we analyze the concrete efficiency and correctness of our scheme RPL. We first construct a scheme using $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$ [26] compiled using the static RBC [50]. The resulting scheme, Σ_{DX}, is a static rebuildable dictionary encryption scheme. We can now use this scheme to build RPL, resulting in a static rebuildable dictionary scheme with no query equality leakage.

Server storage. We first note that the server storage depends on the size of the replicated dictionary, which depends only on the public parameters γ and δ. In particular, the server must store $(2 + \delta) \cdot \gamma$ encrypted label-value pairs.

Client state. The client state consists of the state for the QRT transform: one dictionary with m labels, and one counter; and a query counter. Since we use the $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$ scheme to implement an encrypted dictionary, the scheme itself does not require any separate client state. The client must also store two encrypted entries of the array RAM during the oblivious sort phase of the static RBC.

Non-amortized communication complexity. The get complexity for our scheme is the Get complexity of $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$, which is optimal. However, the $\mathsf{Rebuild}$ protocol is initiated after every γ Get operations. The total time complexity of the rebuild protocol depends on the underlying rebuild of the encrypted dictionary scheme. Since we assume the static RBC instantiated with $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$, the rebuild incurs a logarithmic communication overhead. However, the costs of the rebuild can be amortized over a sequence of γ get operations. In the following, we analyze the amortized time complexity for a Get operation in our scheme.

Amortized communication complexity. A sequence of γ Get operations in our scheme, consists of γ Get operations to the underlying encrypted dictionary scheme, as well as the setup, sorting and put operations performed during the rebuild protocol. Then:

$$\mathrm{T}_{\mathsf{G}}^{\mathsf{RPL}}(g_1, \ldots, g_\gamma) = \sum_{i=1}^{\gamma} \mathrm{T}_{\mathsf{G}}^{\mathsf{EDX}}(g_i) + \mathrm{T}_{\mathsf{R}}^{\mathsf{EDX}}(\gamma),$$

where $\mathrm{T}_{\mathsf{G}}(g_i)$ is the time taken to perform the i^{th} Get operation and $\mathrm{T}_{\mathsf{R}}^{\mathsf{EDX}}(\gamma)$ is the time taken to perform the rebuild operation for EDX.

Given that the RBC rebuild uses an oblivious sort protocol, followed by $M = (2+\delta)\cdot\gamma$ Put operations, and using the Ajtai-Komlos-Szemeredi sorting network [11],

$$\mathrm{T}_{\mathsf{G}}^{\mathsf{RPL}}(g_1, \ldots, g_\gamma) = \sum_{i=1}^{\gamma} \mathrm{T}_{\mathsf{G}}^{\mathsf{EDX}}(g_i) + O(|v|_w \cdot M \cdot \log M) + \sum_{j=1}^{M} \mathrm{T}_{\mathsf{P}}^{\mathsf{EDX}}(p_j),$$

where $\mathrm{T}_{\mathsf{G}}(g_i)$ and $\mathrm{T}_{\mathsf{P}}(p_j)$ are the times taken to perform the i^{th} Get and the j^{th} Put operations, respectively, and $v = \mathsf{qu}(g)$.

Concretely, for $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$, the Get and Put complexities are both $O(1)$, and assuming that $|v|_w$ and δ are $O(1)$, our scheme would have the following complexity:

$$\begin{aligned}
\gamma \cdot \mathrm{T}_{\mathsf{G}}^{\mathsf{RPL}}(g) &= \gamma \cdot O(1) + O(M \cdot \log M) + M \cdot O(1), \\
\mathrm{T}_{\mathsf{G}}^{\mathsf{RPL}}(g) &= O(1) + O((2+\delta)\cdot \log((2+\delta)\cdot\gamma)) + (2+\delta)\cdot O(1), \\
\mathrm{T}_{\mathsf{G}}^{\mathsf{RPL}}(g) &= O(\log((2+\delta)\cdot\gamma)).
\end{aligned}$$

Then our get complexity is logarithmic in the total size of replicated dictionary.

Round complexity. Similarly, we inherit the round complexity of the underlying construction. Since $\Pi_{\mathsf{bas}}^{\mathsf{dyn}}$ executes get operations in one round, our non-amortized get complexity is also one round, and therefore optimal. However, the rebuild introduces $O(M \cdot \log M)$ rounds, if the client has limited state. However, this round complexity can be amortized to $O(\log M)$ rounds per get operation. Further, if the client has more state available, the oblivious sorting gates can be parallelized to reduce the round complexity of the rebuild protocol.

Correctness. We inherit the correctness properties of the QRT. Given that the underlying encrypted dictionary scheme has only a negligible probability of being incorrect, the resulting scheme from RPL will have the correctness properties described in Section 3.5.2, shifted by an additional negligible factor.

Concrete parameters. We now show the concrete correctness properties of RPL depending on the distribution used for the QRT. Since RPL trades off correctness for security, given the fixed public parameters γ and δ, the scheme's correctness properties will vary depending on the replication/query distribution $\vec{p}$. For example, let the number of labels in the dictionary $m = 100$, and the public parameters be $\gamma = 2500, \delta = 3$. Then if both the replication and query distributions are *uniform*, i.e., each ℓ has probability $p_\ell = 1/m$, then from Theorem 3.5.1 we have:

$$\begin{aligned} \varepsilon &= m \cdot e^{-\frac{\delta^2 \gamma \cdot \min_\ell p_\ell}{2+\delta}} \\ &= 100 \cdot e^{-\frac{9 \cdot 2500}{5 \cdot 100}} \approx 2.86 \times 10^{-18} \end{aligned}$$

On the other hand, if the replication and query distribution are both Zipf, i.e., ℓ_i has probability $p_i = 1/(i \cdot H_m)$, $i \in [m]$ where H_m is the m^{th} harmonic number we have instead:

$$\varepsilon = 100 \cdot e^{-\frac{9 \cdot 2500}{5 \cdot 100 \cdot H_{100}}} \approx 0.017.$$

Efficient rebuilding. From the analysis of communication complexity, we see that the largest overhead is incurred due to the rebuilding of the dictionary. The static RBC incurs a logarithmic overhead in communication cost due to the use of an oblivious sorting protocol. We outline here a new rebuild compiler called the *fixed rebuild compiler* (FRC) that will reduce the asymptotic query complexity of RPL. At a high level, similar to the RBC, the FRC takes as input a dynamic dictionary encryption scheme and compiles it into a static rebuildable scheme. The FRC will use two *fixed* but random orderings of the labels in order to rebuild the encrypted dictionary. These orderings are referred to as *schedules*. One of the schedules will be used to download label-value pairs from the original dictionary and store them in the client stash and the other will be used to add label-value pairs from the client stash to the rebuilt dictionary. If an add is scheduled to the rebuilt dictionary, it will be executed *regardless* of if the query is already present in the client stash. Therefore the server sees a query and an add operation at every round of the rebuild protocol. Since the query and add operations follow fixed but random schedules, correlations between them will not be revealed to the server, which is what we require for the security of RPL. The FRC then trades off client state in order to reduce the asymptotic communication complexity of the rebuild protocol. Our proposed compiler is an adaptation of the injection-secure dynamic multi-map encryption scheme, FIX [13][7].

[7]Manuscript received as private communication from the authors.

3.6 Conclusions

We studied whether it was possible to construct a dynamic framework to suppress the query equality leakage of an STE scheme, and if leakage suppression techniques could be practically efficient.

At a high level, our dynamic query equality suppression compiler uses a small server-side cache to suppress the query equality leakage, and rebuilds the entire encrypted structure when the cache is full. We faced the following challenges: (1) in the dynamic setting, the volume leakage is correlated with query equality and the operation identity leakage and therefore our compiler had to suppress all three patterns; (2) we started with a base volume-hiding STE scheme for our compiler. At the time, volume-hiding schemes were only weakly-dynamic, and so our compiler had to upgrade the dynamism of the base construction to make it fully-dynamic; (3) we had to use the base scheme weakly-dynamic operations in order to enable the 'rebuilding' phase of the construction.

Our construction resolved all these challenges and was proved to be secure. It was also shown to be asymptotically more efficient than black-box ORAM simulation, which was the only other viable option for dynamic query equality suppression. During our study, we realized that suppressing the query equality leakage is an expensive operation, and that our techniques were still far from optimally efficient.

This led us to explore practically efficient leakage suppression. We presented our replica-based leakage suppression transform and constructed the first dictionary encryption scheme that has optimal non-amortized query complexity with query equality suppression. We explore the security, efficiency, and correctness trade-offs of the resulting scheme. With this scheme, we initiate a line of work in practical leakage suppression that is crucial to the real-world adoption of these techniques.

Bibliography

[1] The Hacker News.

[2] moz-sql-parser - SQL query parser.

[3] Reddit.

[4] Ruby on Rails.

[5] The TPC-H decision support benchmark.

[6] The TPC-H decision support benchmark documentation.

[7] The Web framework for perfectionists with deadlines | Django.

[8] WordPress.com: Create a Free Website or Blog.

[9] 18 Biggest GDPR Fines of 2020 & 2021 (So Far) | Updated 2021, May 2021.

[10] aermin. ghchat (react version). GitHub.

[11] M. Ajtai, J. Komlós, and E. Szemerédi. An o(n log n) sorting network. In *ACM Symposium on Theory of Computing (STOC '83)*, pages 1–9, 1983.

[12] Ghous Amjad, Seny Kamara, and Tarik Moataz. Breach-resistant structured encryption. In *Proceedings on Privacy Enhancing Technologies (Po/PETS '19)*, 2019.

[13] Ghous Amjad, Seny Kamara, and Tarik Moataz. Injection-secure structured and searchable symmetric encryption. Private manuscript, 2022.

[14] Gilad Asharov, Moni Naor, Gil Segev, and Ido Shahaf. Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 1101–1114. ACM, 2016.

[15] Gilad Asharov, Gil Segev, and Ido Shahaf. Tight tradeoffs in searchable symmetric encryption. In *Annual International Cryptology Conference*, pages 407–436. Springer, 2018.

[16] Automattic. WooCommerce.

[17] K. Batcher. Sorting networks and their applications. In *Proceedings of the Joint Computer Conference*, pages 307–314, 1968.

[18] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *Network and Distributed System Security Symposium (NDSS '20)*, 2020.

[19] R. Bost. Sophos - forward secure searchable encryption. In *ACM Conference on Computer and Communications Security (CCS '16)*, 20016.

[20] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *ACM Conference on Computer and Communications Security (CCS '17)*, 2017.

[21] Raphael Bost and Pierre-Alain Fouque. Thwarting leakage abuse attacks against searchable encryption – a formal approach and applicaitons to database padding. Technical Report 2017/1060, IACR Cryptology ePrint Archive, 2017.

[22] California Legislature. The California Consumer Privacy Act of 2018, June 2018.

[23] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM Conference on Communications and Computer Security (CCS '15)*, pages 668–679. ACM, 2015.

[24] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO '13*. Springer, 2013.

[25] D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.

[26] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[27] Adhityaa Chandrasekar. Commento. GitHub.

[28] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[29] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[30] Data443. The GDPR Framework By Data443.

[31] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 371–406. Springer, 2018.

[32] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, L119:1–88, May 2016.

[33] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *European Symposium on Research in Computer Security (ESORICS '15). Lecture Notes in Computer Science*, volume 9327, pages 123–145, 2015.

[34] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Advances in Cryptology - CRYPTO 2016*, pages 563–592, 2016.

[35] Thailand Government Gazette. Personal data protection act. Unofficial English translation.

[36] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[37] M. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *ACM Workshop on Cloud Computing Security Workshop (CCSW '11)*, pages 95–100, 2011.

[38] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 315–331. ACM, 2018.

[39] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1067–1083. IEEE, 2019.

[40] Gustavo Bordoni. FakerPress.

[41] Arne Holst. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. See `https://www.statista.com/statistics/871513/worldwide-data-created/`. Last accessed: 04-Oct-2021.

[42] PRS Legislative Research India. The personal data protection bill, 2019.

[43] M. Saiful Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS '12)*, 2012.

[44] Zsolt István, Soujanya Ponnapalli, and Vijay Chidambaram. Software-defined data protection: Low overhead policy compliance at the storage layer is within reach! *Proceedings of the VLDB Endowment*, 14(7):1167–1174, March 2021.

[45] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Advances in Cryptology - EUROCRYPT '17*, 2017.

[46] S. Kamara and T. Moataz. SQL on Structurally-Encrypted Data. In *Asiacrypt*, 2018.

[47] S. Kamara and T. Moataz. Computationally volume-hiding structured encryption. In *Advances in Cryptology - Eurocrypt' 19*, 2019.

[48] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC '13)*, 2013.

[49] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.

[50] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. Structured encryption and leakage suppression. In *Advances in Cryptology - CRYPTO '18*, 2018.

[51] Seny Kamara, Tarik Moataz, Stan Zdonik, and Zheguang Zhao. An optimal relational database encryption scheme. *Cryptology ePrint Archive*, 2020.

[52] G. Kellaris, G. Kollios, K. Nissim, and A. O' Neill. Generic attacks on secure outsourced databases. In *ACM Conference on Computer and Communications Security (CCS '16)*, 2016.

[53] Eddie Kohler. Hotcrp conference review software.

[54] Tim Kraska, Michael Stonebraker, Michael Brodie, Sacha Servan-Schreiber, and Daniel Weitzner. Schengendb: A data protection database proposal. In Vijay Gadepally, Timothy Mattson, Michael Stonebraker, Fusheng Wang, Gang Luo, Yanhui Laing, and Alevtina Dubovitskaya, editors, *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, pages 24–38, 2019.

[55] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *ACM-SIAM Symposium on Discrete Algorithms (SODA '12)*, pages 143–156, 2012.

[56] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 297–314. IEEE Computer Society, 2018.

[57] Lobsters Developers. Lobsters news aggregator, March 2018.

[58] Connor Luckett, Andrew Crotty, Alex Galakatos, and Ugur Cetintemel. Odlaw: A tool for retroactive gdpr compliance.

[59] Dan Swinhoe Michael Hill. The 15 biggest data breaches of the 21st century. See https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-cen. Last accessed: 04-Oct-2021.

[60] Tarik Moataz, Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Resizable tree-based oblivious RAM. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, volume 8975 of *Lecture Notes in Computer Science*, pages 147–167. Springer, 2015.

[61] Sudharsanan Muralidharan. Socify: open source social network using ruby on rails. GitHub.

[62] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '15, pages 644–655. ACM, 2015.

[63] M. Naveed, M. Prabhakaran, and C. Gunter. Dynamic searchable encryption via blind storage. In *IEEE Symposium on Security and Privacy (S&P '14)*, 2014.

[64] National Congress of Brazil. Lei geral de proteção de dados [brazilian general data protection law]. English translation by Ronaldo Lemos, Daniel Douek, Sofia Lima Franco, Ramon Alberto dos Santos and Natalia Langenegger.

[65] Van Ons. WP GDPR Compliance.

[66] OpenCart: open source e-commerce platform. Opencart. GitHub.

[67] R. Ostrovsky and V. Shoup. Private information storage. In *ACM Symposium on Theory of Computing (STOC '97)*, pages 294–303, 1997.

[68] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 127–142. USENIX Association, August 2021.

[69] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S.-G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

[70] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious ram with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882. IEEE, 2018.

[71] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: volume-hiding for multi-maps via hashing. In *Conference on Computer and Communications Security (CCS '19)*, pages 79–93, 2019.

[72] PrestaShop SA. Prestashop: open-source e-commerce. GitHub.

[73] schnack! schnack.js. GitHub.

[74] Malte Schwarzkopf, Eddie Kohler, M Frans Kaashoek, and Robert Morris. Gdpr compliance by construction. In *Proceedings of the 2019 Workshop on Heterogeneous Data Management, Polystores, and Analytics for Healthcare (Poly)*, pages 39–53. Springer, August 2019.

[75] Aashaka Shah, Vinay Banakar, Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Analyzing the impact of gdpr on storage systems. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'19, page 4, USA, 2019. USENIX Association.

[76] Faiyaz Shaikh. React-instagram-clone-2.0. GitHub.

[77] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. Understanding and benchmarking the impact of gdpr on database systems. *Proceedings of the VLDB Endowment*, 13(7):1064–1077, March 2020.

[78] E. Shi, T.-H. Chan, E. Stefanov, and M. Li. Oblivious ram with o((logn)¡sup¿3¡/sup¿) worst-case cost. In *Advances in Cryptology - ASIACRYPT '11*, pages 197–214. Springer-Verlag, 2011.

[79] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.

[80] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[81] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *ACM Conference on Computer and Communications Security (CCS '13)*, 2013.

[82] StylemixThemes. GDPR Compliance & Cookie Consent.

[83] Griffin Thorne. Gdpr meets its match ... in china. China Law Blog, July 2019.

[84] Virgina Legislative Information System. 2021 special session, h2307: Consumer data protection act.

[85] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226, 2014.

[86] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security (CCS '08)*, pages 139–148, 2008.

[87] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, 2016.