BREAKING AND BUILDING ENCRYPTED DATABASES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Paul Allen Grubbs

August 2020

BREAKING AND BUILDING ENCRYPTED DATABASES

Paul Allen Grubbs, Ph.D.

Cornell University 2020

The subject of this thesis is *encrypted databases*: systems that use novel cryptographic techniques to store and efficiently query encrypted data. Motivated by the increasing frequency and severity of harmful data breaches, encrypted databases keep data encrypted at all times, ensuring that it is unavailable even to an attacker that compromises the database system's security. To keep queries efficient, encrypted databases must leak some information about the underlying plaintext data and queries. The leakage and its impact on security differs depending on the way the system is compromised.

In this thesis, I investigate the performance-security tradeoffs made by encrypted databases. First, I study current encrypted databases to understand the leakage that would be available to an attacker in likely compromise scenarios. I conclude that many of the security claims made of encrypted databases are incorrect. Then, I examine the security impact of a concrete leakage shared by most encrypted databases. In the process I develop new technical tools based on statistical learning theory. Finally, informed by an understanding of existing databases, I propose a novel performance-security tradeoff for encrypted key-value stores. I instantiate that new tradeoff with frequency smoothing, analyze it using new theory, and build a system.

BIOGRAPHICAL SKETCH

Paul Allen Grubbs was born and raised in Indiana. He did his undergraduate studies in Mathematics and Computer Science at Indiana University. While there, he dipped his toe in the cryptography research waters as an undergrad RA. Finding the temperature to his liking but not wanting to dive in immediately, he opted to move to the Bay area before continuing to graduate school. He worked for nearly two and a half years as a cryptography engineer at Skyhigh Networks (now part of McAfee), where he implemented and deployed cryptographic techniques for managing encrypted data. He left in 2015, packed his car and moved across the country (again) to start his graduate studies in Computer Science at Cornell. After a year in idyllic Ithaca, he packed his car and moved to New York City and Cornell's new Tech campus. In Spring 2018, he lived in England while visiting Royal Holloway, University of London. After many happy years at Cornell, he will begin a yearlong postdoc at NYU in Fall 2020. Then, in summer 2021 he will pack his car and move (again, though perhaps for the last time) to Ann Arbor to join the faculty of the University of Michigan[1].

[1] Go blue.

I dedicate this thesis to my mom, dad, sister, extended family, and all my friends. They all heroically listened to me complain about grad life for five years; I hope they can stomach many more years of me complaining about faculty life.

To Marie-Sarah, *ma pieuvre*:

If I should find a twinkling star

One half so wondrous as you are,

That star would be, like this thesis and me,

Dedicated to you.

ACKNOWLEDGEMENTS

The work in this thesis would not be possible without all of my incredible collaborators and coauthors: including ongoing projects there are now over 30 of you, so I can't thank you all individually even though you all deserve it. I also must thank everyone (students, staff, postdocs, faculty) in the Cornell security and cryptography group for creating an amazing research environment filled with discussion, debate, and brilliance. You all taught me so much. Special thanks to Becky Stewart, one of the most jaw-droppingly competent people I've ever met.

I also thank all the more senior people who've mentored me, given me advice, and written letters for me: Steve Myers, Sasha Boldyreva, David Cash, Vitaly Shmatikov, Kenny Paterson, Rachit Agarwal, and Lorenzo Alvisi. Thanks to Ramin Zabih for being on my committee.

Finally, this thesis wouldn't even exist without my advisor, Tom Ristenpart. Thank you so, so much—for everything.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

It is, perhaps, a truism that the amount of data being generated and stored in computer systems is exploding. As regrettable as it is to begin with bromides, no serious discussion of data security can start other than by acknowledging that nearly every aspect of a human's life is now catalogued and stored in a computer: their hospital birth records, school transcripts, credit reports, employment history, pension disbursements, and death certificate. So, too, are businesses and governments storing more, and more sensitive, data, especially in the age of SARS-CoV-2-mandated remote work. All this means that more (and more complex) data management systems are necessary.

As these systems proliferate and become more complex, they are frequently compromised by hackers, who steal and disclose sensitive information for nefarious purposes. In recent years, medical data [90], financial records [104], classified government secrets [103], and other kinds of data have all been stolen from hacked data management systems. Such attacks are increasing in both frequency and severity.

Against this backdrop, a fundamental shift in the architecture of computer systems is taking place: various components of the software stack are being replaced by services offered by third-party providers. For example, instead of owning and maintaining their own dedicated data management infrastructure, businesses and governments are choosing to *rent* infrastructure and expertise from data management service providers [94, 35, 131] like Amazon or MongoDB. Entire software stacks can even be rented from third-party "software-as-a-service" providers like Salesforce [125].

To summarize: the amount and sensitivity of data stored in data management systems is increasing, these systems are increasingly targeted by attacks, and the rise of outsourcing means these systems' attacks surfaces are getting much larger. These three trends combine to make securing data management systems one of the most urgent research questions in all of computer security, and makes our continued inability to do so one of the most conspicuous and damning failures of our field. [1]

This dissertation is about securing data in data management systems. My research on this topic has focused both on cryptographic techniques that enable some classes of queries to be executed on encrypted data, as well as the systems security questions that arise when incorporating these novel techniques into real systems.

The rest of the dissertation is organized as follows: first, in the remainder of the introduction I will explain the area in more detail, differentiate it from other approaches to securing data management systems, and give a personal account of how I came to work in the area. Then, I will give a brief survey of all my published and ongoing research on this topic, totalling nine distinct projects over the last five years. Finally, I will introduce in more detail the three constituent papers of my dissertation.

In the body (Chapters 2, 3, and 4) I will reproduce all or part of these three papers. These papers were chosen both because they form a representative cross-section of theoretical and practical challenges, and because they are some of the work I am proudest of.

Finally, in the conclusion (Chapter 5) I will give some high-level takeaways of

[1] This is a pretty tough list to top — there are many strong contenders.

my research and outline what I think are the most pressing open problems for future work.

1.1 Introduction to "Encrypted Databases"

We must first understand what an "encrypted database" is. I find it useful to break the term into its component words ("encrypted" and "database") and explain each separately. First, an encrypted database is a *database*: a system designed to facilitate persistent storage and management of data. This sounds tautological, but in fact makes an important distinction between what we do and do not care about: we will *not* generally be concerned with hard disks or file systems, even though databases rely on both. Following convention in this area, we take a somewhat expansive and unorthodox view of "database" to include key-value stores (e.g., Redis), streaming analytics platforms (e.g., Hadoop, Spark), search indexes (e.g., Elasticsearch), graph and scientific databases (e.g., Neo4J) as well as SQL-compliant databases (e.g., MySQL, PostgreSQL) and NoSQL data stores (e.g., Cassandra, MongoDB).[2]

Second, an encrypted database is *encrypted*: more specifically, the data stored in the database is encrypted. The purpose of encrypting the data is to prevent it from being disclosed if the database system is compromised by external hackers, or if a malicious insider (e.g., a database administrator) tries to steal it. Crucially, the data is *end-to-end* encrypted: the key is not stored in the database, nor is the key required to perform queries. This seemingly small detail is actually the crux of the entire idea of an encrypted database, and a major point of departure from related

[2]Our (mis)use of "database" in this area is widely reviled by database researchers: a colleague once told me any self-respecting SIGMOD regular would be moved to physical violence if I used "database" this way in front of them.

solutions for cryptographically protecting stored data, such as full-disk encryption.

Listing these related solutions and explaining why they are insufficient is important enough to spend a few more sentences on. (A persistent, well-founded criticism of work on encrypted databases is that it's never exactly clear what attacks they prevent that other solutions do not.)

An important related solution is full-disk encryption (FDE), which keeps encrypted all data that is stored on some non-volatile storage medium like a hard disk or tape drive. FDE can be done either in hardware, with the encryption key in a tamper-resistant element, or software, with the key in plaintext in the computer's memory. Examples of hardware FDE include Samsung EVO [87]. Examples of software FDE include LUKS, FileVault, and BitLocker [85, 42, 16]. Whether in software or hardware, FDE is designed to protect data in the event of hard disk theft or disclosure of a backup. It does not protect data from being stolen from the computer's memory. Thus, enabling full-disk encryption on an otherwise unmodified database server does not turn it into an encrypted database.

Another related solution is "page-level" or "transparent" encryption for databases [96]. This is akin to FDE, except encryption/decryption is performed by the database application with an in-memory key instead of the OS kernel or in hardware. Usually the database encrypts and decrypts at the granularity of a database page, and like FDE all data is decrypted as soon as it is read into memory. Because an administrator (or any process with root on the server) can still access plaintext, and queries are executed on plaintext, page-level encryption is not an encrypted database per our criteria.

In short, we will concern ourselves only with "end-to-end" encryption that is

applied before the data enters the database system and is not removed until after the data leaves the database. This is not the only important detail of the cryptography in encrypted databases: another aspect that separates encrypted databases is that they use cryptography that allows some kinds of queries to be executed *efficiently*: namely, with required time and bandwidth sublinear in the database size. This restriction serves to rule out "trivial" uninteresting constructions that involve downloading, decrypting, and re-encrypting the entire database on every query. One widely-used example of this kind of cryptography is deterministic encryption, in which every plaintext maps to only one ciphertext. Deterministic encryption allows index lookups and point queries to be done efficiently. We will see a related (deterministic) primitive called a pseudorandom function (PRF) used to enable efficient key-value operations in Chapter 4.

To enable efficient queries, the cryptographic primitives used in encrypted databases all leak some information about the underlying data and queries. For example, deterministic encryption leaks the equality pattern of the underlying plaintexts — it reveals whether any two ciphertexts' underlying plaintexts are equal. Even expensive tools like oblivious RAM (ORAM) and private information retrieval (PIR) leak some information. In general, there is a direct tradeoff between how efficient and functional queries on encrypted data are and how much the cryptographic primitive leaks — more leakage, faster and more expressive queries, and vice versa.

Next, we will state and discuss the most important overarching research questions in encrypted databases. For each, I've stated the broadest version of it in bold, followed by many related sub-questions. There are four, arranged in no particular order.

Q1: How do we build cryptography that support efficient, expressive queries? More specifically, what kinds of algorithms can we use to encrypt data but still allow the kinds of query functionality needed by applications? How do we take advantage of pre-built indexes to accelerate queries? How do we actually perform a query? Do queries require multiple rounds, or just one? Do we need to store data on multiple non-colluding servers? How can we do all this while minimizing leakage?

Q2: How do we evaluate the security of cryptography built in the previous step? What kinds of security definitions should we use? How do we capture the inherent leakage of a primitive in the definition? How do we capture different threat models in definitions? What algorithms can we use to extract information from leakage, and under what assumptions? How do we evaluate these attacks? How do we ensure this evaluation is relevant for real use cases of this cryptography?

Q3: How do we build systems that use this cryptography? Do we build new databases? Do we modify existing ones? Do we force ourselves to use legacy database systems with no modifications? How do we build other system components, like encryption proxies? What properties do those systems need to have? How do we minimize the assumptions needed on other components? How do we preserve the database's query semantics (ACID, linearizability, etc.) using cryptographic query algorithms? How do we compose encryption schemes to handle different queries on different data? On the same data?

Q4: How do we evaluate security of those systems? What kinds of things might an attacker even want to learn? How will this system be compromised in

practice? How have similar systems been compromised in the past? What information will an attacker have at their disposal in a particular compromise model? Do cryptographic security definitions cleanly represent real attacker models? How do we evaluate the composition of leakages from different encryption schemes applied to different data? Can an attacker learn information about data that isn't even in the database?

The "better than plaintext" argument is flawed. For unclear reasons, attacks on encrypted database systems (and/or the cryptography used therein) often lead to tedious public arguments about the validity of the attack, how realistic it is, whether or not the system was used correctly, etc. One of the most reliable tropes in these arguments is that, even if X primitive/system can be attacked, it still improves security because it's better than what everybody does today: storing data in plaintext. For brevity, I'll call this a "better than plaintext" argument.

This is a clean and simple argument with a sheen of admirable "not letting perfect be the enemy of good" pragmatism. In my opinion, though, it is deeply flawed. Since I've come across many forms of this argument, I'd like to spend a few sentences cleanly refuting it so in the future I can refer people to my dissertation instead of rehashing the same debate over and over.

The basic problem is that it's based on a flawed premise, namely that any encryption, even if weak, prevents *some* attacks that would be possible if the data was in plaintext. This premise is flawed because it implicitly assumes everything else about the system would stay the same, and only the presence of encryption would change.

My experience in industry helped me see two reasons why this assumption

is false. First, deploying even so-called "legacy-compatible" encryption schemes into legacy systems necessitates complex new infrastructure like encryption proxies and key management servers. This additional complexity only creates new attack surface. Second, the (perceived) protection offered by encryption often justifies adoption of cloud computing or SaaS. Especially in regulated industries, the question organizations ask often isn't "Should I encrypt my outsourced plaintext data?", but rather "Can I safely outsource my plaintext data to a potentially untrusted third party?". If the answer is "Yes, because we can keep it encrypted" but the encryption is weak, security overall has actually *decreased*.

Aside from the flawed premise, the other major issue with "better than plaintext" is that it doesn't account for the fact that the data owner isn't the only stakeholder, and what's best for the data owner isn't necessarily the best for all stakeholders. For example, many organizations are legally required to report data breaches to customers (the HITECH health care law in the US requires this of medical practices and hospitals [45]) *unless* the data is encrypted. If weak encryption justifies an organization not reporting a large breach of customer data, the organization may be better off, but the customers evidently aren't.

All this being said, I do think that to keep making progress in this area, we eventually need to accept that there is no silver bullet that will prevent all attacks. Anything we come up with will probably be attackable in some settings, so we really can't let perfect be the enemy of good. We simply need to understand (a) in what settings can we build things that provably can't be attacked, and (b) for all other settings, how we can enable informed decision-making about the risks of attacks.

1.2 A Survey of My Papers

Next I'll give a brief survey of all my papers on the subject of encrypted databases, even those not appearing in my thesis. For each paper, I'll follow the summary with at least one of Q1, Q2, Q3, or Q4, to indicate which of the four broad research questions from Section 1.1 that paper addresses.

Leakage-Abuse Attacks Against Searchable Encryption [24] This work was motivated by the questions SHN's customers were asking about the security of SE, as well as the general lack of clarity surrounding SE's leakage. In it, we laid out the "leakage-abuse" methodology of attacking common leakages instead of specific schemes. We also categorized the different kinds of attacks on searchable encryption, as well as the threat models of those attacks. (Q2)

Breaking web applications built on top of encrypted data [57] This paper does for encrypted database *systems* what the CCS'15 paper did for SE: lays out a clean hierarchy of threat models, leakages, and attacks. This paper introduced the snapshot/persistent passive/active trichotomy as the three main threat models for systems, and studies the Mylar [116] system's security in each threat model. It also stirred some controversy in the research community. (Q4)

Why Your Encrypted Database Is Not Secure [58] This short workshop paper tries to understand the "snapshot" threat model used in both crypto and systems papers. In a "snapshot" attack, the adversary steals a one-time copy of the system state. Several works claimed that in this threat model, they achieved very strong security and had no leakage because they used cryptography that only

leaked on queries, and no queries were present in the system states compromised in a snapshot attack. Taking MySQL as a case study, this paper sets out a hierarchy of snapshot attacks that could occur in real deployments and for each, looks at what would be contained in MySQL's state in that attack. In each attack in the hierarchy, we found that queries would be present, demonstrating the snapshot model is unrealistic. (Q4)

Leakage-Abuse Attacks against Order-Revealing Encryption [59] This paper looks at the security of order-revealing encryption for large, sparse domains. The only prior work, by Naveed et al., developed attacks on just one kind of ORE leakage, and their attacks only worked well if the domain was small and almost every possible value occurred many times. Our attacks work on large, sparse domains like names, and can incorporate additional scheme-specific leakage. We also widened the scope of prior work, considering for the first time known- and chosen-plaintext attacks and attacks on schemes without frequency leakage. (Q2)

The Tao of Inference in Privacy-Protected Databases [12] This paper addresses two important questions not answered by prior work: (1) how can leakage from multiple correlated database columns be used in an attack, and (2) can attacks infer plaintexts from columns encrypted with semantically-secure encryption, or even columns that have been segmented out (i.e., removed) from the database? We developed a *multinomial* attack framework based on maximum-likelihood estimation, and used it to answer both of these questions. (Q2)

Pump Up The Volume: Practical Database Reconstruction from Volume Leakage on Range Queries [55] In this paper we studied volume leakage

from range queries. Our goal was understanding the practical security impact of that leakage for real databases, specifically those containing *dense* columns, where each value appears at least once. We developed a database reconstruction attack based on a novel reduction to clique-finding, and showed it would be feasible to execute against many real databases of hospital data. (Q2)

Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks [56] This paper studies the security impact of access pattern leakage, focusing mainly on range queries but also presenting some novel generic attacks. For space reasons, I have not included the attacks on range queries in this thesis. The interested reader should see the full version of the S&P 2019 paper. (Q2)

Pancake: Frequency Smoothing for Encrypted Data Stores [54] In this paper, we work towards efficiently preventing attacks on access pattern leakage. We focus on key-value stores [35, 1, 70], a common building block for applications. To circumvent ORAM lower bounds, we propose a new security model for encrypted databases, that allows the adversary to specify a distribution over queries but not the queries themselves. In this model we show that efficient frequency smoothing can eliminate access pattern leakage with small overhead. (Q1, Q3)

CHAPTER 2

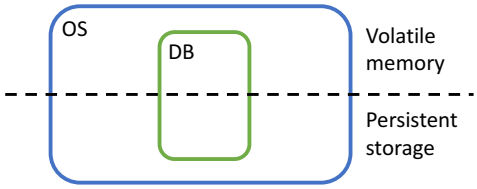
WHY YOUR ENCRYPTED DATABASE IS NOT SECURE

Encrypted databases, a popular approach to protecting data from compromised database management systems (DBMS's), use abstract threat models that capture neither realistic databases, nor realistic attack scenarios. In particular, the "snapshot attacker" model used to support the security claims for many encrypted databases does not reflect the information about past queries available in any snapshot attack on an actual DBMS. We demonstrate how this gap between theory and reality causes encrypted databases to fail to achieve their "provable security" guarantees.

Personal contributions. I suggested that the "snapshot" threat model was unrealistic and proposed this project, though I'm certain the basic idea was folklore before our paper. I also did the survey of MySQL's internals, and connected the dots between what I found and the attacks in Section 2.6.

2.1 Introduction

In this section, we take a system-centric view of encrypted databases and investigate what an attacker would learn in a realistic scenario: stealing a disk, performing SQL injection, or rootkitting the OS. We demonstrate that a "snapshot" attacker, which is the main security model of most encrypted databases, is largely a myth. Modern DBMS's keep logs, caches, and data structures that, in any realistic snapshot attack, reveal information about past queries. This leakage is inherent in today's production environments because a DBMS must maintain caches and other metadata to adapt the system to the workload and help manage its performance.



	Logs	Diagnostic Tables	Data Structures
Disk theft	✓		
SQL injection	✓	✓	
VM snapshot leak	✓	✓	✓
Full-system compromise	✓	✓	✓

Figure 2.1: We use a simple abstraction of DB-hosting systems (left) to explain the discrepancies between the academic threat models and the information revealed in actual attacks (right).

We then concretely demonstrate how an attacker can exploit this system-level information to break the claimed security guarantees of encrypted databases. We conclude with guidelines for future research.

2.2 Attacks on Databases

We use a simple abstraction to explain (1) academic threat models in the encrypted database (EDB) literature and (2) concrete attacks that DBMS's face in reality.

Academic threat models. The strongest threat model (e.g., mentioned in [116, 112]) is an *active attacker* who fully compromises the DBMS server and performs arbitrary malicious operations. Such attacks are difficult to defend against (viz. [57]), and recent designs for efficient EDBs no longer claim security against active attacks.

Instead, the latest security models focus on passive attacks that do not interfere with DBMS functionality. The weaker version is a **snapshot attacker** [83, 57, 112] who obtains "a snapshot of the database (tables and indices included)" [112]. The stronger version is a **persistent passive attacker** [57] who compromises the DBMS server and passively observes all its operations. The latter includes

observing the queries issued to the database and how they access the encrypted data.

Observations of query evaluation are particularly damaging in EDBs that rely on property-revealing encryption (PRE), such as order-revealing encryption [83], deterministic encryption [9], and searchable encryption [25]. All PRE schemes leak some information about plaintexts in order to support certain computations by the DBMS. Some PRE ciphertexts always leak [9, 18], enabling powerful snapshot attacks that recover plaintexts [100, 24, 59]. Other PRE schemes [116, 83, 25] leak only if the attacker observes accesses to the ciphertexts (e.g., search queries). By definition, a persistent attacker can exploit the leakage from queries and accesses to recover plaintexts [57, 24, 69, 65].

Since PRE schemes are always vulnerable to persistent attacks, many EDBs claim security against snapshot attacks only. Examples include the latest claims [117, 115] for CryptDB [114] and Mylar [116] (revised after the original claims were shown false by, respectively, [100] and [57]), new systems Arx [112] and Seabed [106], and new cryptographic schemes such as Lewi-Wu order-revealing encryption [83]. A common implicit assumption for these systems is that snapshot attackers will *not* obtain past queries. We will show that this assumption is false in commodity DBMS's under realistic snapshot attacks.

A simple system abstraction. We will treat a DBMS as if it consists of (a) the DB software running as one or more user-level processes and (b) the rest of the system, including the OS and other applications (OS, for brevity). Therefore, the state of the system has four parts: volatile DB state (data in RAM and CPU registers), persistent DB state (data on disk), volatile OS state, and persistent OS

state—see Figure 2.1. For simplicity, we assume the database is not sharded across multiple machines, i.e., even if the database is replicated, every machine has a full copy of the data.

We use MySQL as our running example, but similar caches, logs, and data structures exist in all practical DBMS's and can be recovered via forensic analysis (e.g., see [20] for MongoDB).

Concrete attacks. An oft-cited threat to DB security is ***disk theft***, i.e., theft of persistent storage [114, 117, 48, 4, 83, 96, 105, 41]. Full-disk encryption (FDE) can mitigate this threat, but EDBs aim to protect data even in the absence of FDE. Without FDE, this attack yields the persistent OS and DB state, but not any volatile state.

SQL injection is an old but still prevalent [142] attack. It also enables arbitrary code injection and full control of the memory space of the DB process [60, 38], thus yielding the persistent and volatile DB state.

DBMS's increasingly run on virtual machines (VMs), exposing them to the threat of ***VM image leaks*** [121, 47, 22, 8]. Some VM snapshots only contain the persistent storage, whereas full-state snapshots also include the VM's memory and CPU registers. We focus on the latter. This attack yields the persistent and volatile OS and DB state.

Finally, a ***full-system compromise*** involves rooting the DBMS and gaining full access to the persistent and volatile OS and DB state. This enables persistent passive and active attacks, but "smash-and-grab" attacks that simply grab available data and leave are prevalent [142].

The table on the right of Figure 2.1 summarizes DBMS-specific data yielded by the attacks of each type. In the rest of the paper, we explain how this data reveals information about past queries, thus breaking the security models of EDBs and enabling recovery of plaintext data.

2.3 Logs on Disk

We start by investigating what information about past queries can be gleaned from the log files on disk that are required for high availability and transactional semantics in a production DBMS.

Inferring writes. Industrial databases must support transactions with ACID properties (atomicity, consistency, isolation, durability). We'll use MySQL as an example. Other DBMS's such as SQL Server, Postgres, and MongoDB use similar techniques.

MySQL's default storage engine, InnoDB, uses circular *undo* and *redo* logs to give the database layer multi-version concurrency control. Both logs record changes to the individual database records at the byte level. Using standard forensic techniques for reconstructing insert, update, and delete transactions from these logs [43, 44], an attacker who compromised the disk can reconstruct queries that modified the database. The number of reconstructed queries depends on insertion size and volume. For example, with 1 write modifying a 20-byte field per second, the undo and redo logs of default size (50 Mb) store 16 days' worth of inserts.

Recent work [57] showed that the timing of queries reveals sensitive data in certain applications of encrypted databases. In MySQL, timing can be extracted

from a separate *binary log* (binlog) used to support replicated transactions and point-in-time recovery [14]. Binlog stores the text of every transaction that modifies any row of the database, along with its UNIX timestamp. It is not enabled upon installation but must be turned on for high availability and therefore will be present on the disk of production MySQL servers. This log is so important that a utility for reading it (`mysqlbinlog`) comes pre-installed with MySQL [15]. Its contents are never purged unless the administrator executes a special command. A similar mechanism for replicated transactions in MongoDB also records transaction timestamps [95]. Even without this log, the default primary key of each MongoDB document contains its creation time [20].

MySQL's binlog also enables the attacker to compute the correlation between the timestamps and the rate of change in the log sequence numbers (LSN). The attacker can thus infer the approximate timestamps for the transactions in the undo and redo logs that are no longer present in the binlog.

This leakage is inherent in ACID databases. Transactional guarantees require the ability to roll back recent transactions (perhaps even across database crashes), thus information about recent database modifications must persist on the disk.

Inferring reads. The easiest way for information about reads to end up on disk is through too-verbose logging. In MySQL, the general query log records every query, including SELECT, but few systems enable it because it takes huge amounts of disk space. Instead, on many production MySQL systems, the "slow query" log [132] records transactions that take an unusually long time.

A more subtle way to extract information about read-only queries is from the buffer pool file. On shutdown and at other points during normal server opera-

tion, MySQL creates a file in the data directory containing the current pages in the buffer pool in LRU order. This is done to avoid a "warm-up" period of slow responses after a restart. This file reveals information about several previous SELECT queries, such as the paths through the B+ tree that MySQL took when evaluating them.

2.4 Diagnostic Tables

SQL injection is still a common way to compromise databases [142]. Designers of encrypted databases often assume that a SQL injection attack reveals only the database's view of the data itself. But modern DBMS's include tables—extractable via SQL injection—that store a great deal of performance statistics, intended to help tune specific databases to their workloads and diagnose problems and performance bottlenecks [118, 88, 93, 136].

The **information_schema** database in MySQL [64] aggregates information about the internal state of the DBMS, including contents of caches and how many connections are active. It also includes a `processlist` table with the timestamped list of all currently executing queries. By injecting a SELECT query on this table, an attacker can obtain queries made by other users.

The **performance_schema** database [109] aggregates statistics about query execution, such as the number of queries per second and the amount of contention for synchronization objects. It also contains a `threads` table with the current statements being executed by all threads, enabling a SQL-injection attacker to monitor queries.

This database also keeps information about all past queries. The `events_statements_current` table stores the current statement being executed by any thread. The `events_statements_history` table stores the most recent queries made by any thread (essentially, the most recent queries appearing in `events_statements_current` for all threads). The number of queries stored per thread is configurable (10 by default). In addition to the text of the query, it also stores the number of rows examined by the query and returned to the client.

MySQL does not store historical information about every individual query, but `performance_schema` stores statistics about all query "types" made since the database was last restarted. The "type" is determined by a simple canonicalization algorithm which removes the arguments but preserves the select-from-where structure of the query and the attributes it uses. So, for example, the queries `SELECT * FROM CUSTOMERS WHERE STATE='IN'` and `SELECT * FROM CUSTOMERS WHERE STATE='AZ'` have the same canonical form, which is different from the canonical form of the queries `SELECT * FROM CUSTOMERS WHERE AGE >=25` and `SELECT * FROM CUSTOMERS WHERE STATE='IN' AND AGE >=25` (the `WHERE` clause has multiple constraints in the latter).

Even if the DBMS has internal access controls, SQL injection can be leveraged into arbitrary code execution [60, 38] that bypasses all access restrictions within the DBMS process.

2.5 In-memory Data Structures

The strongest snapshot attack scenario involves the attacker obtaining an image of the virtual machine executing the DBMS or, alternatively, rootkitting the OS

(but only making a single observation of the system). This snapshot reveals a point-in-time state of the entire persistent and volatile memory. We focus on the internal data structures of the database process because they reveal information about past queries, especially accesses to the individual pages in its cache.

To adaptively improve performance and support (amortized) constant-time retrieval for frequently accessed database pages, InnoDB keeps per-page metadata and access counters. If a page is accessed often, InnoDB indexes its contents in an **adaptive hash index** [2]. Postgres has a similar mechanism for tracking accesses to individual pages to handle eviction from its buffer cache.

Further, the **query cache** in MySQL is an internal key-value map that can be configured to keep the results of certain SELECT queries [119] so that answering them is essentially free. Unlike the buffer pool, this cache is strictly internal to MySQL and cannot be exposed via `information_schema` (see Section 2.4), but will be visible to a whole-system snapshot attacker. Other commodity DBMS's, too, implement some form of query caching—e.g., Microsoft SQL Server caches queries and their execution plans but not the full result set [89].

Even if the query cache is disabled, queries persist in MySQL's internal heap long after they've been executed. We performed a simple experiment with MySQL in the default configuration. First, we issued a SELECT query with a random string as the column name. This random string appears nowhere in the database, thus the query does not match any rows. Then, we issued 100 SELECT queries which matched some rows and 900 that did not. Then, we inserted 500 random rows and made 1,000 more SELECT queries, waited around twenty minutes and made 100,000 more SELECT queries. After this, we dumped the memory of the MySQL process.

The full text of the original query appeared in three distinct locations in memory, and the random string appeared in three additional locations by itself. We verified that this is not a peculiarity of how MySQL handles column names by repeating the experiment with a random string parameter in a WHERE clause. This leak is not surprising since MySQL is not designed for security-critical operations and does not implement secure deletion. In Section 2.6, we show that in the context of encrypted databases this otherwise minor oversight has dramatic implications for the (lack of) security.

2.6 How Systems Fail

We now explain how the confidentiality of data in encrypted databases can be broken by snapshot attackers using techniques described above. We focus on the encrypted databases that have been designed to work on top of commodity DBMS's.

At-rest encryption. This protection works the same way in most DBMS's: a key, stored in memory but not on disk, is used to encrypt the database files on disk. An attacker who compromises only the disk will therefore learn nothing useful (except via side channels such as relative sizes of encrypted objects), but any higher level of access will reveal the entire data.

Token-based systems. Many encrypted databases are based on schemes that delegate a query-specific trapdoor to the server. The server uses it to reveal information about the plaintexts necessary to answer the query. For example, CryptDB

and Mylar [114, 116] use variants of the scheme of Song et al. [135]. More advanced examples include the ORE scheme of Lewi and Wu [83] and the searchable encryption scheme of Cash et al. [25].

For any such scheme, semantic security [51] cannot be achieved if the attacker obtains even a single token value. Intuitively, semantic security requires that even if the attacker knows the original query, he cannot tell the difference between an encrypted record that matched the query and one that didn't. If the attacker observes the query token, he can apply it to the encrypted database and recognize which records match and which don't, thus breaking the definition of semantic security.

As we explained in Section 2.5, the text of queries (and, therefore, the search token) is stored in several locations in MySQL. Queries also appear in two log files, the query cache, `performance_schema` and `information_schema`. They can even be found after the fact in the internal heap of the DBMS. Tokens will thus be available to any realistic snapshot attacker.

The consequences depend on the system. For CryptDB, Mylar, and any other system using variants of searchable encryption [114, 116, 25, 77], a snapshot attacker can use the leakage-abuse attacks such as [24] to infer the query and the plaintext of any record it matches.

These attacks exploit the observation that the number of results that match a query is often unique across a corpus, e.g., 63% of the 500 most frequent words in the Enron email corpus have a unique result count. With partial knowledge of the encrypted documents, unique counts immediately reveal the value of the corresponding encrypted keyword. Since the search functionality also reveals which

documents contain the keyword, this attack also recovers partial content of the encrypted documents.

Lewi-Wu ORE. In the Lewi-Wu scheme [83], query tokens reveal ordering information and, in some parameter regimes, individual plaintext bits. A damaging attack against an ORE with similar leakage was demonstrated in [59], but it does not directly apply to the Lewi-Wu ORE because the Lewi-Wu scheme is not deterministic.

Nevertheless, the Lewi-Wu scheme reveals equality of plaintexts when a value in the database is queried. This leaks a partial histogram to a snapshot attacker, who can combine it with the bit leakage from the query tokens and the "binomial attack" of [59], to which the Lewi-Wu scheme is vulnerable even in the absence of tokens.

To show the consequences of this leakage, we simulated an attack on the Lewi-Wu scheme (with block size of 1 bit). We sampled a database of 32-bit integers and several range queries (both an upper and lower bound), all uniformly at random. We then computed the leakage resulting from each set of queries if executed against a given database, aggregating the results over 1,000 trials.

For a database of size 10,000 and only *five* simulated range queries, the average fraction of bits leaked (out of possible 320,000) is surprisingly high, around 12%, i.e., 4 bits of each 32-bit value are leaked on average. For twenty-five range queries, the fraction is 19%. If fifty range queries are found in the memory snapshot (this is not inconceivable in practice; MySQL can create dozens of threads for query processing and network I/O), the snapshot attacker recovers 25% of the bits (on average, 8 bits of each 32-bit value). We did not attempt to estimate whether the

leakage would be equally severe for non-uniform distributions of range queries.

In summary, query tokens found in system snapshots enable a snapshot adversary to recover large amounts of protected data in all existing encrypted databases.

Seabed. Seabed's ORE scheme [106] is known to be insecure [59]. The attack of [59] uses the known plaintext distribution (auxiliary model), which is publicly available for many types of data. It starts by computing all possible comparisons between the ciphertexts, as permitted by the ORE scheme, to learn some bits of the underlying plaintexts. Then, it creates a bipartite graph in which each ciphertext is a node on the left-hand side and each possible plaintext is a node on the right-hand side, and draws an edge between a left-hand node and a right-hand node only if the bits it learned about the left-hand ciphertext match the bits of the right-hand plaintext. Each edge in the graph is weighted using frequency information. Finally, the attack recovers the most likely plaintext for each ciphertext by finding a matching in the graph.

For data in the columns that need to support joins, Seabed uses basic deterministic encryption (DET). This data is therefore vulnerable to the frequency analysis attack described below. For data in the columns used as filters in count or aggregation queries, Seabed attempts to prevent frequency analysis using the SPLASHE scheme, which creates a different column for each possible plaintext. Analytics queries such as aggregations are rewritten and encrypted so they are evaluated on the correct column. For example, if the plaintext 10 corresponds to the `c3` column of `table`, the query `SELECT count(*) FROM table WHERE a = 10` is converted into an equivalent of `SELECT ashe(c3) FROM table`, where "ashe()" is a custom summation over ciphertexts [106, Table 2].

If two queries have different values in the WHERE clause, they will operate on different columns (after rewriting). If SPLASHE were to run on MySQL, the `events_statements_summary_by_digest` table in the `performance_schema` database will canonicalize them to different forms. This table will thus count the number of queries made *for each plaintext*. This reveals the exact histogram of queries for each plaintext value to any attacker with a snapshot of the DBMS memory. If SPLASHE runs on Spark, the attacker can simply obtain queries from the event history server [136] or from the heap of the worker nodes.

Characterizing the exact leakage of query distributions (as opposed to plaintext distributions) is an open problem in general, but if the attacker has a sufficiently good model of the query distribution, then basic inference attacks like frequency analysis can be used.

Frequency analysis is a very simple cryptanalytic technique which would work here in two steps. In the first step, the observed histogram of the ciphertexts and the histogram of the query distribution model would both be sorted in decreasing order. So, for example, the most frequently occurring ciphertext query would be the first element in the list of queries, and the most frequently occurring query according to the model would be the first element on its list. In the second step, the elements of the lists are matched by rank: the first elements are matched with each other, then the second elements, and so on. Lacharité and Paterson [79] proved that this simple process is a maximum-likelihood estimator for the encryption function, meaning that this attack is most likely to correctly recover the underlying plaintexts.

While frequency analysis can recover the plaintext corresponding to a given column in Seabed, it will not recover the value of that column for a particular

row. However, to save space, an enhanced version of SPLASHE uses deterministic encryption with padding for infrequent plaintext values, rather than creating a dedicated column in the schema. The `performance_schema` will leak a query histogram for the frequently occurring values (as described above) in this scheme, but will not leak a histogram of the infrequently occurring values. Nevertheless, a partial histogram could be reconstructed from the logs or in-memory data structures. This leakage is even more damaging against enhanced SPLASHE because the frequency analysis attack described above can reveal the value in the enhanced SPLASHE column for a particular row. Combined with other leakage about frequent values from query patterns and any DET- or ORE-encrypted columns, this enables even more damaging cross-column inference attacks.

Arx. Arx [112] uses a treap-based data structure to evaluate range queries on encrypted data in a single network round-trip using chained garbled circuits. Because index values are encrypted using standard encryption, the authors claim semantic security against snapshot attacks.

An important property of the Arx scheme is that after each range query, the nodes of the treap become "consumed" and must be repaired; essentially the client must supply a new encryption of the node's value which overwrites the old value. Reads and writes are thus perfectly correlated because a read of any node is immediately followed by a write to the same node. If Arx runs on MySQL, MongoDB, or a similar DBMS, a snapshot of the system's persistent state will contain a transcript of every range query made on the index because the write corresponding to each read will be recorded in the transaction logs. This breaks semantic security and gives a snapshot attacker much of the information a persistent attacker would have. For example, this snapshot attacker will have ordering information about

the upper and lower bounds of encrypted range queries, as well as the frequency of visits to each node in the tree.

Arx uses a two-round protocol to hide the relationship between the node values of the range query index and the database rows holding that value, thus a snapshot attack on the range query data structure does not immediately reveal the plaintext in a given row. Nevertheless, the leakage is sufficient to recover the values in the index using a variant of the bipartite matching attack from [59] described above. The index does not leak the frequencies of individual values, but transaction logs do leak the frequencies of visits to each value in the index. These frequencies can be used in combination with auxiliary data about the distribution of queries to recover these values.

Moreover, transaction logs leak the *rank* of the queries, i.e., the number of values in the index less than the query. Prior work has shown how to exploit this leakage [59], and we conjecture that a similar approach can be used to recover the plaintext of the encrypted queries. We leave full development of this attack to future work.

2.7 Discussion

Deploying encrypted databases on commodity DBMS's can have unexpectedly bad consequences for security. Logs, caches, and data structures kept by DBMS's leak information that is not accounted for in the threat models used by the designers of encrypted databases. Critically, today there is no such thing as a "snapshot" attacker who cannot observe past queries, workloads, and access patterns—because any realistic snapshot of the system contains this information. We demonstrated

how this leads to confidentiality breaches.

We focused on commodity DBMS's, but similar issues arise in schemes using custom or non-standard databases [107, 66, 28, 101]. There is a trend in database design towards adaptively changing the structure of the database based on the workload [129, 29, 63]. We expect that the snapshots of such databases leak even more information about past queries.

There appears to be an inherent conflict between security and transparency: if the internal information about workloads is available to the developers and administrators, it is also available in some form to a snapshot attacker. The tension between effective caching and security was noted in the early research on history-independent data structures [98], but whether history independence can be achieved for practical encrypted databases remains an open question. Solving it requires new research into designing and implementing databases that efficiently hide queries and access patterns.

We conclude with guidelines and recommendations for the different research communities working in the area of encrypted databases.

Cryptographers: A desirable property of new schemes is that the encrypted data by itself leaks nothing about the plaintext. This does <u>not</u> imply security against "snapshot" or "offline" attacks when the scheme is actually deployed in a real DBMS.

Systems researchers: Any system that uses any kind of property-revealing encryption must be assessed using state-of-the-art leakage-abuse attacks as part of the standard evaluation, and the results of the assessment must be presented in the evaluation section of the paper.

PC members of systems conferences: Any paper that proposes an encrypted database but does not assess its security under known attacks should be treated with suspicion. Be very skeptical of claims of "provable confidentiality," especially if not supported by a thorough security evaluation, and solicit external reviews from cryptographers and security researchers to validate these claims.

CHAPTER 3

LEARNING TO RECONSTRUCT: STATISTICAL LEARNING THEORY AND ENCRYPTED DATABASE ATTACKS

We show that the problem of reconstructing encrypted databases from access pattern leakage is closely related to statistical learning theory. This new viewpoint enables us to develop broader attacks that are supported by streamlined performance analyses. As an introduction to this viewpoint, we first present a general reduction from reconstruction with known queries to PAC learning. Then we consider what learning theory tells us about the impact of access pattern leakage for other classes of queries, focusing on prefix and suffix queries. We illustrate this with both concrete attacks for prefix queries and with a general lower bound for all query classes.

Personal contributions. I suggested the use of statistical learning theory to understand access pattern leakage and developed the correspondence between learning and reconstruction attacks. I also contributed the PAC learning reduction and proof (though not the tight analysis in Section 3.3.1), the attack on prefix queries, and the general lower bound.

3.1 Introduction

In this work, we show how statistical learning theory effectively addresses the problem of database reconstruction, yielding new results across a range of settings. A common thread through all of our results is the analysis of *concept spaces* over the set of all queries. The results we apply from learning theory rely on the *VC*

dimension of the concept space, intuitively a measure of how complex it is. (See Section 3.2 for a short primer on statistical learning theory.)

PAC learning and known-query attacks. In Section 3.3, we show that database reconstruction given a set of *known* queries can be recast as an instance of Probably Approximately Correct (PAC) learning, and standard results from that field can predict how many queries are needed to achieve reconstruction. We present this reduction to PAC learning as an introduction to how we view database reconstruction as a learning problem, as well as an illustration of the power of this viewpoint. While the attack model here is rather powerful, it is considered realistic in some recent literature [146, 59, 57]; our analysis largely resolves the question of how damaging such attacks can be. In the remainder, we no longer assume queries are known, aligning our setting with most prior work.

Beyond range queries. As illustration of the power of the viewpoint we have taken, in Section 3.4, we generalize approximate reconstruction to other query classes and analyze the resulting attacks using tools from learning theory. Using *generalization error* as a metric γ on the values in the database, we show that all query classes with finite VC dimension reveal the distance (in γ) between the underlying values of records, which allows an attacker to group records whose values are close. Further, we show how to use an ϵ-net to precisely analyze how many queries are needed to guarantee all groups of records have small diameter according to γ. We construct, analyze, and evaluate the first reconstruction attack on prefix queries. We conclude the section with a general lower bound, via a reduction to PAC learning, relating the query class's VC dimension, attack accuracy, and number of queries needed for any reconstruction attack using access pattern

leakage. In addition to being of theoretical interest, this suggests VC dimension or similar concepts from learning theory could be a useful way to compare different techniques which leak access patterns.

Notation and assumptions. Throughout $[n]$ denotes the set of integers $\{1, \ldots, n\}$; $[a, b]$ denotes the set of integers within the given interval; and open brackets such as $[a, b[$ denote that the corresponding endpoint is excluded. (If $b \leq a$, $[a, b[$ is empty.) We model a database as a set of R records where each record has a single attribute that takes an integer value in $[N]$. We let $\mathsf{val}(r) \in [N]$ denote the value of the record r.

We assume the adversary knows the number of possible values N, and the set of all possible queries. We do not assume that the adversary knows the set of all records in advance, or even their number. We do not assume that every value appears in at least one record (no density assumption).

3.1.1 Related Work

Kellaris *et al.* (KKNO) [69] described the first exact reconstruction attack on range queries with access pattern leakage; Lacharité *et al.* (LMP) [78] improved the results of KKNO in the dense setting, obtaining an $\mathcal{O}(N \log N)$ exact reconstruction attack. Kornaropoulos et al. [76] gave an approximate reconstruction attack for access pattern leakage from k-nearest-neighbor queries. Other papers attacking encrypted databases include [59, 100, 12, 57, 24]; these mostly analyze so-called "property-revealing encryption" schemes, which leak strictly more than what we assume.

3.2 Statistical Learning Theory Primer

We begin with a brief introduction to some elements of statistical learning theory that will play a central role in our work. We use terminology from a recent textbook [91].

3.2.1 Concept Spaces, ϵ-Nets, ϵ-Samples

Let X be some set of (base) elements. In this work, X is always finite (although our scale-free reconstruction bounds extend to infinite sets, e.g. continuous ranges, as is). A *concept* (also called *event* or *range*) C is a subset of X. Given a probability distribution $\mathcal{D}$ on the set X, let $f_{\mathcal{D}}$ represent the probability mass function. The probability $\Pr_{\mathcal{D}}(C)$ of a concept C is equal to the probability that a single element of X sampled according to $\mathcal{D}$ is in C, i.e., $\Pr_{\mathcal{D}}(C) = \sum_{c \in C} f_{\mathcal{D}}(c)$.

A *concept space* (or *set system*) is a pair $(\mathsf{X}, \mathbb{C})$ where $\mathbb{C}$ is a set of concepts (subsets) of X. Any concept C can also be viewed as a function $\mathsf{X} \to \{0, 1\}$ (its characteristic function): the function's output on input $x \in \mathsf{X}$ is 1 if $x \in C$ and 0 otherwise. Given a concept space $(\mathsf{X}, \mathbb{C})$ and a sample S of elements drawn from X according to $\mathcal{D}$, we may ask the following questions:

- Does every concept in $\mathbb{C}$ with some not-too-small probability occur in the sample S?
- Is the relative occurrence of every concept of $\mathbb{C}$ in the sample S close to its expectation?

Answering these questions involves analyzing objects called ϵ-nets [61] and ϵ-

samples.

Definition 1. *A subset $S \subseteq \mathsf{X}$ is an ϵ-net for the concept space $(\mathsf{X}, \mathbb{C})$ with respect to the distribution $\mathcal{D}$ if for every event $C \in \mathbb{C}$ with $\Pr_{\mathcal{D}}(C) \geq \epsilon$, the intersection $S \cap C$ is non-empty.*

Definition 2. *A subset $S \subseteq \mathsf{X}$ is an ϵ-sample (also called ϵ-*approximation*) for the concept space $(\mathsf{X}, \mathbb{C})$ with respect to the distribution $\mathcal{D}$ if for every concept $C \in \mathbb{C}$,*

$$|\frac{|S \cap C|}{|S|} - Pr_{\mathcal{D}}(C)| \leq \epsilon.$$

Informally, a sample S is an ϵ-sample when every concept's relative frequency in S is within ϵ of its true probability. It is an ϵ-net iff every concept of probability at least ϵ occurs in the sample.

One way to analyze when a set S is an ϵ-net or an ϵ-sample is to characterize the complexity of the concept space. We turn to this next.

3.2.2 Shattering, VC Dimension, Growth Functions

The critical measures in determining the complexity of a concept space are the growth function $m_{\mathbb{C}}(n)$ and the Vapnik-Chervonenkis (VC) dimension d, which are related. Given a concept space $(\mathsf{X}, \mathbb{C})$ and a finite sample $S \subseteq \mathsf{X}$, an important object is the *set of subsamples of S induced by* $\mathbb{C}$ (also called *the projection of* $\mathbb{C}$ *on S*): $\mathbb{C}_S := \{C \cap S\}_{C \in \mathbb{C}}$. The size of this set is the *index of* $\mathbb{C}$ *with respect to S*:

$$\Delta_{\mathbb{C}}(S) := |\mathbb{C}_S| = |\{C \cap S : C \in \mathbb{C}\}|.$$

Clearly, the index of a concept space relative to a set S is at most $2^{|S|}$, and, when $\mathbb{C}$ is finite, it is at most $|\mathbb{C}|$.

The concept space $(\mathsf{X}, \mathbb{C})$ *shatters* the sample $S \subseteq \mathsf{X}$ if $\mathbb{C}$ induces all possible subsamples of S, i.e., $\Delta_{\mathbb{C}}(S) = 2^{|S|}$.

The *VC dimension* (also called *density* [126] or *capacity* and denoted by d) of a concept space $(\mathsf{X}, \mathbb{C})$ is the largest cardinality (possibly infinite) of a set $S \subseteq \mathsf{X}$ that can be shattered by $\mathbb{C}$. (It is sufficient for only one set of this size to exist; not all sets of this size need to be shattered by $\mathbb{C}$.) VC dimension is an indicator of the complexity of a concept space. Related to VC dimension is the *growth function* of a concept space $(\mathsf{X}, \mathbb{C})$, which is the maximum index of $\mathbb{C}$ over all samples $S \subseteq \mathsf{X}$ of size n: $m_{\mathbb{C}}(n) := \max_{S \subseteq \mathsf{X}: |S|=n} \Delta_{\mathbb{C}}(S)$.

The VC dimension, then, is the largest value of n for which the growth function equals 2^n. Knowing the VC dimension of a concept space is sufficient to determine an upper bound on the growth function: either d is infinite or the growth function is bounded by $\sum_{i=0}^{d} \binom{n}{i}$.

Lemma 1 (Sauer's Lemma [126])**.** *Let* $(\mathsf{X}, \mathbb{C})$ *be a concept space having finite VC dimension* d*. Then, the growth function satisfies* $m_{\mathbb{C}}(n) \leq \sum_{i=0}^{d} \binom{n}{i}$*.*

Moreover, the partial sum of binomial coefficients $\sum_{i=0}^{d} \binom{n}{i}$ is upper-bounded by $(ne/d)^d \leq n^d$ for all $d \geq 2$.

3.2.3 Sufficient Conditions for ϵ-Nets and ϵ-Samples

In their groundbreaking paper, Vapnik and Chervonenkis established a lower bound [141, Thm. 2] on the probability that a sample S is an ϵ-sample, i.e., that the relative frequencies of events in $\mathbb{C}$ are all within ϵ of their true probabilities.

Theorem 1 (Sufficient conditions for ϵ-sample [141])**.** *Let* $(\mathsf{X}, \mathbb{C})$ *be a concept*

space with growth function $m_{\mathbb{C}}(n)$ and VC dimension d. Let $\mathcal{D}$ be a probability distribution on X and let S be a set of size n drawn from X according to $\mathcal{D}$. Then, for any $\epsilon > 0$, the probability that S is an ϵ-sample is at least $1-4\cdot m_{\mathbb{C}}(2n)\cdot e^{-\epsilon^2 n/8}$. In particular, there is an n that is

$$\mathcal{O}\left(\frac{d}{\epsilon^2}\log\frac{d}{\epsilon}+\frac{1}{\epsilon^2}\log\frac{1}{\delta}\right)$$

such that a sample S of size at least n is an ϵ-sample with probability at least $1-\delta$.

More precisely, [91, Thm. 14.15] establishes that a sample of size at least $\frac{32d}{\epsilon^2}\log\frac{64d}{\epsilon^2}+\frac{16}{\epsilon^2}\log\frac{2}{\delta}$ is an ϵ-sample with probability at least $1-\delta$.

Inspired by Vapnik and Chervonenkis's work on ϵ-samples, Haussler and Welzel introduced ϵ-nets and derived a lower bound in the case where the distribution over X is uniform [61, Thm. 3.7]. Later work extended this bound to arbitrary distributions:

Theorem 2 (Sufficient conditions for ϵ-net [91])**.** *Let $(\mathsf{X},\mathbb{C})$ be a concept space with growth function $m_{\mathbb{C}}(n)$ and VC dimension d. Let $\mathcal{D}$ be a probability distribution on X and let S be a set of size n drawn from X according to $\mathcal{D}$. Then, for any $\epsilon > 0$, the probability that S is an ϵ-net is at least $1-2\cdot m_{\mathbb{C}}(2n)\cdot e^{-\epsilon n/2}$. In particular, there is an n that is*

$$\mathcal{O}\left(\frac{d}{\epsilon}\log\frac{d}{\epsilon}+\frac{1}{\epsilon}\log\frac{1}{\delta}\right)$$

such that a sample of at least this size is an ϵ-net with probability at least $1-\delta$. Specifically, a random sample of size at least $\max\{\frac{8d}{\epsilon}\log\frac{16d}{\epsilon},\frac{4}{\epsilon}\log\frac{2}{\delta}\}$ is an ϵ-net with probability at least $1-\delta$.

Ehrenfeucht *et al.* prove a lower bound [39, Cor. 5] on the number of samples needed to obtain an ϵ-net with probability at least $1-\delta$. Since every ϵ-sample is an ϵ-net, this lower bound also applies to ϵ-samples.

Theorem 3 (Necessary conditions for ϵ-net [39, 91])**.** *Let* $(\mathsf{X}, \mathbb{C})$ *be a concept space of VC dimension* d*. Let* $\mathcal{D}$ *be a probability distribution on* X *and let* S *be sample drawn from* X *according to* $\mathcal{D}$*. Let* $\epsilon > 0$ *and* $\delta > 0$*. Suppose* S *is an* ϵ*-net with probability at least* $1 - \delta$*. Then* $|S| = \Omega\left(\frac{d}{\epsilon} + \frac{1}{\epsilon}\log\frac{1}{\delta}\right)$*.*

3.2.4 PAC Learning

Introduced by Valiant [140], PAC learning is concerned with algorithms that learn from labelled examples. (We restrict our attention to *realizable* and *consistent* PAC learning; for a more general treatment see [68].) Using the terminology above, a learner $\mathcal{L}$ is an algorithm which takes as input a transcript $\{(s_i, C(s_i))\}_{i=1}^m$ of elements from X (where each $s_i \leftarrow\$ \pi$ for some distribution π on X) along with their labels according to the unknown concept C. A learner outputs a *hypothesis* $H \in \mathbb{C}$ representing its guess for C.

The learner $\mathcal{L}$ is a *PAC learner* for $\mathbb{C}$ if for any $C \in \mathbb{C}$, distribution π on X, and $0 < \epsilon, \delta < 1/2$, for any sample (or transcript) of size m in $\mathcal{O}(\mathrm{poly}(\frac{1}{\epsilon}, \frac{1}{\delta}))$ drawn according to π, the *generalization error* $\Pr_\pi[\,\{x \in \mathsf{X} \mid H(x) \neq C(x)\}\,]$ is less than ϵ with probability at least $1-\delta$. In words, the generalization error is the probability under π that an element is labelled differently by H and C. A central result [17] in learning theory states that if $\mathcal{C}$ has finite VC dimension, then there exists a (not necessarily efficient) PAC learner for $\mathcal{C}$. In particular, the following holds [91].

Theorem 4. *Let* $(\mathsf{X}, \mathbb{C})$ *be a concept space with finite VC dimension* d*. Then for any* $0 < \epsilon, \delta < 1/2$*, there exists a PAC learner* $\mathcal{L}$ *for* $(\mathsf{X}, \mathbb{C})$ *that uses a sample of size*

$$m = \mathcal{O}\left(\frac{d}{\epsilon}\log\frac{d}{\epsilon} + \frac{1}{\epsilon}\log\frac{1}{\delta}\right).$$

That is, for any concept $C \in \mathbb{C}$, with probability at least $1 - \delta$, $\mathcal{L}$ achieves generalization error less than ϵ using a sample of size m.

3.3 PAC Learning and Database Reconstruction Attacks

In this section, we begin exploring the connection between learning theory and database reconstruction attacks. Concretely, we demonstrate a connection between approximate database reconstruction and "Probably Approximately Correct" (PAC) learning [140] in the setting where the attacker has access pattern leakage from some known queries. For a brief introduction to PAC learning, see Section 3.2.4.

Reconstruction via PAC learning. The attack setting we consider here is one in which an attacker has observed the access pattern leakage from a number of *known* queries drawn i.i.d. from a fixed query distribution (which the adversary does not need to know). The assumption of known queries is somewhat stronger than has been considered previously in this literature; however, some recent works have argued that it is realistic [59, 57] on the grounds that the adversary is able to make some queries or has compromised an honest user.

A crucial question is the relationship between the *number* of known queries and the amount of information the adversary can learn about the database itself, cf. Section 1.1. We will see that this question is largely resolved via a simple reduction to PAC learning in the known query setting.

We can think of a database DB with R records having values in $[N]$ as being a vector of length R with values in $[N]$; the value of record j is $DB[j]$. We construct

a concept space $\mathcal{C} = (\mathcal{Q}, \mathbb{C})$ as follows. The points in the ground set are the possible queries $q \in \mathcal{Q}$. We write $q(i) = 1$ when value $i \in [N]$ matches query q. We set $C_i = \{q \in \mathcal{Q} \,|\, q(i) = 1\}$. We then define $\mathbb{C} = \{C_i : i \in [N]\}$. With this set-up, we have the following result.

Theorem 5. *Let $\mathcal{Q}$ be a class of queries and $\mathcal{C} = (\mathcal{Q}, \mathbb{C})$ be the concept space constructed as above. Let π_q be any distribution over $\mathcal{Q}$. Let d be the VC dimension of $\mathcal{C}$, and assume d is finite. Then, there is an adversary such that for any database DB, given as input $m \in \mathcal{O}(\frac{d}{\epsilon} \log \frac{d}{\epsilon\delta})$ queries sampled from π_q and their access pattern leakage on DB, the adversary outputs a database DB' such that $\Pr_{\pi_q}[\, q(DB[j]) \neq q(DB'[j]) \,] \leq \epsilon$ holds simultaneously for all $j \in [R]$, with probability at least $1 - R\delta$.*

This theorem requires some explanation. In the statement, we chose to use the generalization error $\Pr_{\pi_q}[\, q(DB[j]) \neq q(DB'[j]) \,]$ as the accuracy measure. This is intended to surface the core points without adding unnecessary detail, but it may also make the result hard to interpret. Section 3.4 studies in more detail how generalization error relates to traditional notions of attack accuracy.

The proof proceeds via a natural reduction to PAC learning. The adversary gets as input m known queries along with their access pattern leakage (i.e. which records match the query) for each of the R records in the database. The core observation is that the access pattern is a binary classification of each database element; further, each database value is a concept in $\mathbb{C}$. This means that the task of reconstructing each database element can be seen as R PAC learning experiments for the concept space $\mathcal{C}$ defined above. The adversary simply runs the PAC learner from Theorem 4 R times, invoking it once for each record j. For each invocation, the adversary gives the learner as input the m queries and their access patterns (i.e. the 0/1 labellings)

for record j. Each time the learner is run, it outputs a hypothesis $H_{\mathsf{ind}} \in \mathbb{C}$ corresponding to an element of $[N]$. The adversary's complete output is then of the form $[H_1, H_2, \ldots, H_R]$, which we denote by DB'. Each independent invocation of the learner outputs a hypothesis H_j such that $\Pr_{\pi_q}[\, q(H_j) \neq q(DB[j]) \,] > \epsilon$ with probability at most δ, and a union bound over the R elements completes the proof.

Remark. Here, we obtain a probability bound that depends on R. While this may look discouraging, the sample complexity of PAC learning is only logarithmic in δ, so the resulting loss in tightness small. Further, in Section 3.3.1, we show that the dependency on R can be removed. The proof in Section 3.3.1 does not use a generic reduction to PAC learning; instead it uses a fixed learner and applies the ϵ-net theorem. The approach we chose in this section is simpler and directly highlights the connection with PAC learning.

Extensions. The above result can be extended in several ways. First, a symmetric and nearly identical result can be proven about *query* reconstruction attacks in the presence of known records. Such a result would proceed as above except flipping the role of the queries and values in the concept space. An interesting extension of the above result (and its twin for known database elements) is a setting where the adversary has both known and unknown queries (or database elements). One question which our PAC learning approach can address is how much information the adversary learns about the *unknown* information elements by PAC learning with its known information.

Some authors have argued recently that the even stronger setting of *chosen* query attacks is a realistic threat model for encrypted databases. With chosen queries, the corresponding learning setting is not PAC learning, but active learn-

ing [34]. Active learning is similar to PAC learning except the learner can adaptively query an oracle which labels points in X according to the unknown concept. Interestingly, both the folklore binary-search attack on order-revealing encryption and the Zhang *et al.* [146] document-injection attack can be viewed as active learning algorithms.

We note that lower bounds on the sample complexity of PAC learning can be used to prove that certain kinds of security guarantees hold even in the presence of some known or chosen queries (or database elements). In Section 3.4 we state and prove one such result which does not assume any records or queries are known to the adversary.

3.3.1 Reducing Query Complexity for Reconstruction with Known Queries

Next we prove that the dependency on the number of records R can be removed from Theorem 5. This is formalized in Theorem 6.

Recall the following notation from Section 3.3. We can think of a database DB with R records having values in $[N]$ as being a vector of length R with values in $[N]$; the value of record j is $DB[j]$. We construct a concept space $\mathcal{C} = (\mathcal{Q}, \mathbb{C})$ as follows. The points in the ground set are the possible queries $q \in \mathcal{Q}$. We write $q(i) = 1$ when value $i \in [N]$ matches query q. We set $C_i = \{q \in \mathcal{Q} \mid q(i) = 1\}$. We then define $\mathbb{C} = \{C_i : i \in [N]\}$.

Theorem 6. *Let $\mathcal{Q}$ be a class of queries and $\mathcal{C} = (\mathcal{Q}, \mathbb{C})$ be the concept space constructed as above. Let π_q be any distribution over $\mathcal{Q}$. Let d be the VC di-*

mension of $\mathcal{C}$, and assume d is finite. Then, there is an adversary such that for any database DB, given as input $m \in \mathcal{O}(\frac{d}{\epsilon}\log\frac{d}{\epsilon\delta})$ queries sampled from π_q and their access pattern leakage on DB, the adversary outputs a database DB' such that $\Pr_{\pi_q}[\, q(DB[j]) \neq q(DB'[j])\,] \leq \epsilon$ holds simultaneously for all $j \in [R]$, with probability at least $1-\delta$.

Remark. The statement above is identical to Theorem 5, except the probability of success is $1-\delta$ instead of $1-R\delta$. In reality, there is small price to pay: the required number of queries is increased by a constant factor, per Lemma 2 below. This potential increase is not reflected in the statement, since the constant factor disappears into the $\mathcal{O}()$ notation.

Before proving Theorem 6, we start with a short lemma. Given a concept class $(\mathsf{X}, \mathbb{C})$, $\mathbb{C}^{\Delta}$ denotes the set of symmetric differences of elements of $\mathbb{C}$; that is:

$$\mathbb{C}^{\Delta} := \{\Delta(A, B) : A, B \in \mathbb{C}\}$$

where $\Delta(\cdot,\cdot)$ denotes the symmetric difference of the input sets.

Lemma 2. *Let $(\mathsf{X}, \mathbb{C})$ be an arbitrary concept class with finite VC dimension d. Then the VC dimension of $(\mathsf{X}, \mathbb{C}^{\Delta})$ is also finite. Moreover it is $\mathcal{O}(d)$.*

Proof. The proof of the lemma is identical to standard proofs of the same result for e.g. unions (see [91, Theorem 14.5] for a generic argument). For the sake of completeness, we give a proof here. Let d denote the VC dimension of $(\mathsf{X}, \mathbb{C})$. Then the growth function of $(\mathsf{X}, \mathbb{C})$ is upper-bounded by $\sum_{i=0}^{d}\binom{n}{i} \leq (ne/d)^d$ as a function of the number of points n by Lemma 1. Hence, given n points in X, $\mathbb{C}$ induces at most $(ne/d)^d$ subsamples, hence symmetric differences of two elements in $\mathbb{C}$ can induce at most $(ne/d)^{2d}$ subsamples. To show that $n = \alpha d$ points cannot

be shattered by $\mathbb{C}^{\Delta}$ for some α, it is thus enough to show that $(\alpha e)^{2d} < 2^{\alpha d}$, which is equivalent to $\alpha e < 2^{\alpha/2}$, which is clearly true for some fixed α (e.g. $\alpha = 10$ suffices). $\square$

We now turn to the proof of Theorem 6. The proof is no longer a "generic" reduction to PAC learning. Instead, the proof uses the ϵ-net theorem (Theorem 2) directly.

Proof of Theorem 6. We choose as adversary any adversary that outputs any database DB' that is consistent with the observed leakage. Such a database exists, since DB must be consistent with its own leakage. (As with general PAC learning, there is no guarantee that the adversary is efficient.) Next we pick a sample size m large enough to ensure that the sample is an ϵ-net for the concept space $(\mathsf{X}, \mathbb{C}^{\Delta})$. By Lemma 2, the VC dimension of that concept space is $O(d)$. By the ϵ-net theorem, it follows that $m \in \mathcal{O}(\frac{d}{\epsilon} \log \frac{d}{\epsilon\delta})$ suffices to ensure that the sample forms an ϵ-net with probability at least $1 - \delta$. We claim that this choice of adversary and m satisfies the conclusion of the theorem.

To see this, assume the sample is an ϵ-net, which holds with probability at least $1 - \delta$. Let DB' be the database output by the adversary. Assume towards contradiction that there exists $j \in [R]$ such that $\Pr_{\pi_q}[\, q(DB[j]) \neq q(DB'[j]) \,] > \epsilon$. This last expression is equivalent to saying that the measure of $\Delta(C_{DB[j]}, C_{DB'[j]})$ according to π_q is greater than ϵ. Since the sample is an ϵ-net for $\mathbb{C}^{\Delta}$, it must contain a point in $\Delta(C_{DB[j]}, C_{DB'[j]})$. This point is a (known) query that matches record j in DB but not in DB', or conversely. Hence the database DB' output by the adversary is not consistent with the sample, a contradiction. $\square$

3.4 Generalizing Approximate Reconstruction

We have seen how ϵ-nets and ϵ-samples can be used to build and analyze approximate reconstruction attacks on range queries. In this section, we abstract a core technical idea from those attacks – that records accessed the same way by most queries must be "close" – and show how it extends beyond range queries. We explore this in three ways: (1) by using learning theory to define a natural and general notion of distance relevant to access pattern attacks, (2) by showing how ϵ-nets are the right technical tool for analyzing the meaning of this distance for particular query classes, and (3) by using this distance notion to prove a general lower bound on the query complexity of any attack with access pattern leakage. To the best of our knowledge, our general lower bound is the first such proof ever given for this setting and illustrates a core finding of this work: the security impact of access pattern leakage for any class of queries is related to its VC dimension.

Distances induced by range queries. Let C_i be the set of range queries matching value i, and C_j, j. Then, the set of queries matching i XOR j (exactly one of i and j) is $\Delta(C_i, C_j)$, where Δ is the symmetric difference operator on sets, and the number of such queries is $\gamma(i, j) \coloneqq |\Delta(C_i, C_j)|$. We can make three interesting observations about $\gamma(i, j)$. First, it is related to the numerical distance metric $|i - j|$ (though, importantly, they are not identical). Second, $\gamma(\cdot, \cdot)$ is itself a metric on $[N]$. Third, distance in this metric is approximately revealed by the access pattern leakage of range queries: if every query accesses either both or neither records i and j, then $\gamma(i, j)$ is likely to be small. These three properties were used extensively in our attacks on range queries, but they are not specific to range queries: we can abstract them using ideas from learning theory.

Distance, generally. Consider any class of queries $\mathcal{Q}$ on $[N]$ and distribution π over those queries, and consider the concept space $(\mathcal{Q}, \mathbb{C})$ with concepts $\mathbb{C} := \{C_i\}_{i \in [N]}$, where each $C_i := \{q \in \mathcal{Q} \mid q(i) = 1\}$. Each query is a point in this concept space, and there is a set corresponding to each possible value in $[N]$ containing the queries that match it. Now, define the *symmetric difference* concept space $(\mathsf{X}, \mathbb{C})^{\Delta} := (\mathsf{X}, \mathbb{C}^{\Delta})$, where $\mathbb{C}^{\Delta} := \{\Delta(C_i, C_j)\}_{i,j \in [N]}$ and $\Delta(\cdot, \cdot)$ is the symmetric difference of the input sets. This new concept space contains, for each pair i, j, the queries which return exactly one of i, j. By Lemma 2, the VC dimension of C^{Δ} is at most twice the VC dimension of $\mathbb{C}$. Next, define the function $\gamma_{\pi}(i, j) := \Pr_{\pi}[\, \Delta(C_i, C_j)\,]$. As above for range queries, where implicitly π was the uniform distribution, this defines a metric on $[N]$. To see that the triangle inequality holds, observe that for any i, j, and k, any query in $\Delta(C_i, C_j)$ is in $\Delta(C_i, C_k)$ or $\Delta(C_k, C_j)$. This allows us to generalize the use of ϵ-nets in ApproxOrder to *arbitrary* query classes. If the adversary observes a set of queries that is an ϵ-net for the symmetric difference concept space, then it must be the case that for any subset S of records with identical access pattern, the underlying values V of those records satisfy $\mathsf{diam}_{\gamma}(V) := \max_{i,j \in V} \gamma_{\pi}(i, j) \leq \epsilon$.

Thus, if we simply group together records that have the same access pattern, then the existence of an ϵ-net provides an upper bound on the distance (with respect to the measure γ) of records in the same group. Essentially, access pattern leakage from *any* query class reveals a kind of approximate equality between the underlying values of the records in the database. This approximate equality depends both on the query class and the query distribution. For range queries, we used this approximate equality to build the ApproxOrder attack and reveal a great deal of information with few queries. However, closeness in the metric γ may not be practically interesting for all query classes and distributions: for ex-

ample, access pattern leakage from the "query class" which is sampled uniformly at random from $2^{[N]}$ is unlikely to reveal anything interesting. Nevertheless, for many query classes used in practice, closeness in this distance metric can lead to serious privacy breaches. For example, for prefix queries, two values being close in this metric implies they have a common prefix. We will show a simple attack that allows an adversary to reveal which records in the database are approximately equal according to the distance metric γ_π.

Approximate equality attack. Consider a set of queries $\mathcal{Q}$, possible record values $[N]$, and resulting concept space $(\mathcal{Q}, \mathbb{C})$, whose VC dimension d we assume is finite and ≥ 2. Let π_q be any distribution over $\mathcal{Q}$. The attack takes as input records $\{r_1, r_2, \ldots, r_R\}$ along with a 0-1 matrix AP with R rows and Q columns, where $\mathsf{AP}_{ij} = 1$ iff query j returns record i. The attack views each row of the matrix as a number in $[0, 2^Q - 1]$ and outputs a partition by grouping all records with the same number. Let $g_i = \{r_1^i, \ldots, r_k^i\}$ be any such group, and let $V = \{v_1, \ldots, v_k\}$ be the underlying values of these records. An application of the ϵ-net theorem lets us immediately conclude that $\Pr_{\pi_q}[\,\mathsf{diam}_\gamma(V) \leq \epsilon\,] > 1 - (2Q)^d 2^{-\epsilon Q/2}$, and this bound holds for all groups simultaneously.

3.4.1 Prefix and Suffix Queries

Next, we show how to instantiate the approximate equality attack for a practically relevant query class. For a set $\Sigma^{\leq \ell}$ of all strings with length $\leq \ell$ from some alphabet Σ, define a *prefix query* q to be a string in the set $\cup_{j=1}^{\ell} \Sigma^j$. In text search, prefix queries are usually indicated by a trailing asterisk "*". For any element $j \in \Sigma^{\leq \ell}$, define the predicate $q(j)$ to be 1 if either $q = j$ or q is a prefix of j, and 0 otherwise.

As an example, take the database $\{\text{cat}, \text{carbon}\}$. A prefix query "c*" on these two values would return both, but "carb*" would return only the second one.

Although prefix queries are technically a subset of range queries, there are three crucial differences which obviate the use of previous attacks on range queries: prefix queries do not reveal order, they cannot overlap without one query being contained in the other, and the number of queries matching any fixed string is constant. (Replacing "prefix" with "suffix" in the discussion above gives an identical query class that matches strings based on a suffix instead of a prefix. Our discussion and attacks easily translate to suffix queries, so we dispense with a separate discussion for them.)

In the the symmetric difference concept space for prefix queries, the concepts $\Delta(C_i, C_j)$ for $i, j \in \Sigma^{\leq \ell}$ are the queries that are prefixes of exactly one of i or j. If i and j themselves have a common prefix, though, some prefix queries will match both i and j. More precisely, if i and j have a length-k common prefix, then $|\Delta(C_i, C_j)| = (|i| - k) + (|j| - k)$. Informally, if the adversary notices that two records are always accessed together or not at all, then it can infer that they share a long common prefix. We will describe how to formalize this intuition with ϵ-nets. Further, if the adversary has a model of the database distribution, it can use frequency analysis to learn the characters of each record, one at a time (reminiscent of the climax of the science-fiction movie *WarGames*).

A *WarGames* attack on prefix search. Most modern text and web search systems support prefix queries on unstructured data [40], and they are ubiquitous in software-as-a-service (SaaS) products like Salesforce, ServiceNow, and Dropbox [125, 134, 37]. A common [37, 125] design pattern for these systems is to send

a prefix query for every character the user types in the search bar. Since users may find their desired result without finishing their query, the distribution of queries is heavily biased towards shorter prefixes.

Our attack in this setting is simple. First, the adversary runs the approximate equality attack described above, obtaining a partition of the records in the database. Then, for each record, it takes the union of all query results containing that record. Here is where we apply the generalized distance notion discussed earlier: with an ϵ-net, we can ensure that each group in the partition contains records with at least a length-one common prefix, and that the unions we form afterwards are exactly the sets of records with the same first character. The first character of each record is then recovered via frequency analysis, and the attack is iterated to learn the second character, then the third, etc.

Analyzing the attack. We model the queries as being sampled via a two-step process. First, a prefix length ℓ_q is sampled from a Zipf distribution on $[\ell]$. (Recall that the standard Zipf distribution on ℓ elements has $\Pr[\,i\,] = (1/i)/H_\ell$, where $H_\ell = \sum_{m=1}^{\ell} 1/m$ is the ℓth harmonic number.) Then, the query is sampled as a uniformly random element of Σ^{ℓ_q}. Call this distribution over queries π_{ts}.

We first consider, for two words $i, j \in \Sigma^{\leq \ell}$, how the length of i and j's common prefix relates to $\Pr_{\pi_{ts}}[\,\Delta(C_i, C_j)\,]$. If i and j have different lengths and share a length-k prefix, then $\Pr_{\pi_{ts}}[\,\Delta(C_i, C_j)\,] = \frac{1}{H_\ell}\left(\sum_{m=k+1}^{|i|} 1/(m|\Sigma|^m) + \sum_{m=k+1}^{|j|} 1/(m|\Sigma|^m)\right)$. Let $\ell_{\min}$ be the length of the shortest string. If the queries observed by the adversary are an ϵ-net for the symmetric difference concept space and for $\epsilon = \frac{1}{H_\ell}\sum_{m=1}^{\ell_{\min}} 1/(m|\Sigma|^m)$, then, for all i, j having no common prefix, we have the distance $\gamma_{\pi_{ts}}(i, j) > \epsilon$ and a query ac-

cessing i and j differently must have occurred. The VC dimension of this concept space is at most 4, so $\mathcal{O}(\frac{1}{\epsilon}\log\frac{1}{\epsilon\delta})$ queries suffice for this attack to recover the first character of every record with probability at least $1-\delta$. This same analysis can be iterated for the rest of the characters.

Experiments. We implemented the attack using last name data from the Fraternal Order of Police (FOP) database dump, posted online in 2016. It contains the personal information of over 600,000 law enforcement officers in the United States. For auxiliary data, we used public US Census statistics [23] on last name frequencies. We also ran the attack on the FAA ZIP code dataset used in the published version of this work, but it performed quite poorly, primarily due to the auxiliary data being a poor model of the ZIP code distribution.

In 9 out of 10 trials with only 500 prefix queries sampled according to the distribution described above, we were able to partition the records into groups with at least a one-character prefix in common. The mean number of queries required to do this was 315. Once we obtain this partition, we recovered the first character for over 70% of the last-name records. With the same number of trials for 40,000 queries, we recovered the first and second characters of over 55% of the last-name records. With 3 million queries, we recovered the first three characters for over 40% of last-name records, and we recovered roughly 1,500 three-character last names exactly. The sample complexities given by the ϵ-net theorem above are 1,491, 120,000, and 6 million for recovering 1, 2, and 3 characters—much higher than our experiments indicated. As we saw above, applying these results can give loose bounds but the "true" constants are usually small.

This attack on prefix queries can be improved. Our goal was not simply to con-

struct an accurate reconstruction attack for prefix queries, but to demonstrate the power of the learning-theoretic approach in building and analyzing reconstruction attacks. We can generalize the prefix attack to obtain the three basic steps for this approach: (1) define a concept space and a metric, (2) use an ϵ-net to analyze the number of queries needed to learn approximate equality, then (3) perform an attack on the information about values revealed by approximate equality. We note also that standard results [91] on intersections and unions of concept classes can extend this approach to *composite* query classes (e.g. a SQL query which intersects the result of a range query on one column and a prefix query on another).

3.4.2 A General Lower Bound on Attacks

The metric γ is defined for *any* query class, and in many cases this leads to privacy implications: for range queries, it is closely related to the distance between record values; for prefix queries, the length of the longest common prefix. A general approximate reconstruction attack should recover values that are close (for γ) to the actual record values, and lower bounds on closeness (for γ) should imply lower bounds on the accuracy of any approximate reconstruction attack. The following theorem gives one such lower bound on the number of queries necessary for any approximate reconstruction attack on any query class, as a function of the desired accuracy ϵ and the VC dimension d of the query class.

Theorem 7. *Let $\mathcal{Q}$ be a class of queries on $[N]$, π_q a query distribution, and $\mathcal{C} = (\mathcal{Q}, \mathbb{C})$ the associated concept space with VC dimension $d > 1$. Let $\gamma(i,j) := \Pr_{\pi_q}[\,\Delta(C_i, C_j)\,]$ be the distance metric induced on $[N]$ by $\mathcal{Q}$ and π_q. Consider any algorithm that takes as input a database of size R with elements in $[N]$, together with the access pattern leakage of m queries sampled from π_q, and outputs an*

approximation DB' such that $\gamma(DB[i], DB'[i]) \leq \epsilon$ for all $i \in [1, \ldots, R]$, with probability of success at least $1 - \delta$ (over the choices of queries from π_q). Then m is in $\Omega(\frac{d}{\epsilon} + \frac{1}{\epsilon} \log \frac{1}{\delta})$.

This result is a direct application of PAC learning theory: an algorithm that takes any database as input and outputs a DB' satisfying the stated condition is a PAC learner for the concept space $\mathcal{C}$ defined in the theorem statement. We can thus apply a general lower bound [39] on the sample complexity of PAC learning to conclude that m must be in $\Omega(\frac{d}{\epsilon} + \frac{1}{\epsilon} \log \frac{1}{\delta})$. With a smaller number of queries m, there will be, with probability at least δ, two values in $[N]$ whose distance γ is strictly greater than ϵ, but which every query given to the algorithm accessed in the same way.

This result is not easy to interpret, so we briefly reflect on its implications. First, it holds even if the adversary knows the exact query distribution and database distribution. Next, note that the same lower bound holds for the existence of an ϵ-net: if the queries fail to form an ϵ-net for the metric γ, then some records that are more than ϵ apart in γ cannot be separated based on access pattern. Since any approximate attack should be able to distinguish such records, in some sense this approximate equality attack is a *minimal* approximate attack. For example, consider the range query attacks from the conference version of this work [56]. Recovering approximate values or a partition into buckets with small diameters implies we are able to group together approximately-equal records. From this perspective, the lower bound on the existence of an ϵ-net for γ may be interpreted as a lower bound on the number of queries necessary for *any* form of approximate attack for which γ is a relevant notion of distance—not only an approximate attack attempting to recover values, as in Theorem 7.

CHAPTER 4

FREQUENCY-SMOOTHING FOR ENCRYPTED DATA STORES

We present PANCAKE, the first system to protect key-value stores from access pattern leakage attacks with small constant factor bandwidth overhead. PANCAKE uses a new approach, that we call *frequency smoothing*, to transform plaintext accesses into uniformly distributed encrypted accesses to an encrypted data store. We show that frequency smoothing prevents access pattern leakage attacks by passive persistent adversaries in a new formal security model. We integrate PANCAKE into three key-value stores used in production clusters, and demonstrate its practicality: on standard benchmarks, PANCAKE achieves 229× better throughput than non-recursive Path ORAM — within 3–6× of insecure baselines for these key-value stores.

Personal contributions. To the best of my recollection, I proposed using the query distribution to efficiently prevent access pattern attacks, as well as what would eventually become this paper's security model. I also suggested the batching mechanism, and wrote most of the formal analyses in the paper.

4.1 Introduction

High-performance data stores, such as key-value stores [35, 1, 70], document stores [92], and graph stores [102, 71], are a building block for many applications. For ease of management and scalability, many organizations have recently transitioned from on-premise to cloud-hosted data stores (e.g., [35]), and from server-attached to disaggregated storage [143, 46, 73, 62]. While beneficial, these deployment settings lead to significant security concerns: data accesses that used

to be contained within a trusted domain (an organization's premises or within a server) are now visible to potentially untrusted entities.

A now-long line of work has shown that, even if the data is encrypted, the observed data access patterns can be exploited to learn damaging information about the data, through access pattern attacks such as frequency analysis (e.g., [65, 24, 69, 56, 76]). These attacks require only a passive persistent adversary, that is, one that observes access patterns but does not actively performs accesses. Existing techniques that are secure against access pattern attacks, such as oblivious RAMs [50], target stronger security models where the adversary can actively perform data accesses; as we discuss in detail in §4.2, these techniques have fundamental performance overheads [19, 82, 110, 144, 81, 108] making them impractical for most settings. Thus, the problem of building high-performance data stores that are secure against access pattern attacks by persistent passive adversaries remains open.

We make three core contributions towards resolving this open problem. First, we introduce a formal security model that captures (just) passive persistent adversaries in encrypted data store settings. Specifically, we model honest users' queries to the data store as a sequence of data access requests sampled from a time-varying distribution. The encryption mechanism can obtain an estimate of the distribution; the adversary both knows the distribution and obtains the transcript of (encrypted) queries and responses. Informally, we say that a mechanism is secure if the adversary is unable to distinguish the transcript from a sequence of uniformly distributed accesses to random bit strings. We capture this security goal in what we call real-or-random indistinguishability under chosen dynamic-distribution attack (ROR-CDDA).

Our second contribution is *frequency smoothing*, a mechanism that is ROR-CDDA secure, that is, provides security against access pattern attacks by passive persistent adversaries.

The key insight underlying frequency smoothing is that, for passive persistent adversaries, data access requests being chosen from a distribution provides a source of "uncertainty" that can be leveraged in a principled manner. For instance, if requests were sampled from a uniform distribution, it is easy to see that the adversary gains no additional information from observing accesses patterns. However, most real world distributions are not uniform. Frequency smoothing uses the estimate of the data access distribution to transform a sequence of requests into uniform accesses over encrypted objects (hereafter, key-value pairs) in the data store.

Frequency smoothing carefully combines four techniques: selective replication, fake accesses, batching of queries, and dynamic adaptation. Selective replication creates "replicas" of key-value pairs that have high access probability relative to others in the data store. This serves to partially smooth the distribution over (replicated) key-value pairs. For the remaining non-uniformity, we combine selective replication with the idea of "fake" queries [86]. These are sampled from a carefully crafted fake access distribution to boost the likelihood of accessing replicated key-value pairs until the resulting distribution is entirely uniform.

Security requires ensuring that fake and real queries be indistinguishable; to achieve this, we issue small batches of encrypted queries, where each query is either real or fake with equal probability.

Finally, we show how one can dynamically adapt to changes in the underlying data access distribution by opportunistically adapting the replica creation as well

as the fake access distribution.

Our third contribution is the design, implementation, and evaluation of an end-to-end system — PANCAKE— that realizes frequency smoothing, and can be used with existing data stores. PANCAKE builds upon the encryption proxy system model used in many deployment settings, where a proxy acts as an intermediary between clients and the data store. PANCAKE uses this proxy to maintain an estimate of the time-varying access distribution (based on incoming requests from the clients), as well as securely execute read/write queries by using pseudorandom functions for keys and authenticated encryption for values. Assuming the distribution estimates are sufficiently good (we make this precise in §4.4), PANCAKE achieves ROR-CDDA security.

We analyze PANCAKE's performance both analytically and empirically. Specifically, we show that PANCAKE's server-side storage and bandwidth overheads are within a constant factor of insecure data stores; while the proxy storage can be large in the worst-case (depending on the underlying data access distributions), our empirical evaluation demonstrates minimal overheads for heavy-tailed, real-world distributions.

We integrate PANCAKE with two key-value stores used in production clusters — a main-memory based key-value store Redis [120] and an SSD-based key-value store RocksDB [122]. Evaluation over a variety of workloads demonstrates that PANCAKE consistently achieves throughput within $3 - 6\times$ of the respective key-value store that does not protect against access pattern leakage attacks. Sensitivity analysis against various workloads, deployment scenarios (within a cloud and across wide-area networks), query loads, and more, demonstrates that PANCAKE maintains its performance across a diversity of evaluated contexts. We also compare

PANCAKE performance against Path ORAM [138], a representative system from the ORAM literature. Across various workloads, PANCAKE achieves significantly better throughput (sometimes by as much as 229×) than PathORAM. Of course, ORAMs are designed to prevent a broader range of attacks (e.g., active injection attacks); our comparison should be interpreted as highlighting the huge efficiency gap between countermeasures in the two threat models. An end-to-end implementation of PANCAKE along with all the details to reproduce our results is available at `https://github.com/pancake-security`.

PANCAKE is a first step toward designing high-performance data stores that are secure against access pattern attacks by passive persistent adversaries. We outline limitations, open research questions, and future research avenues in §4.7.

4.2 The pancake Security Model

We introduce a new security model for capturing passive persistent attacks against encrypted data stores. We also discuss prior approaches for resisting access pattern attacks.

System model. We focus on key-value (KV) stores that support (single-key) get, put, and delete operations on KV pairs (k, v) submitted by one or more clients. Our results can, however, be applied to any data store that supports read/write/delete operations.

We consider outsourced storage settings where one or more clients want to utilize a KV store securely. PANCAKE employs a proxy architecture commonly used by encrypted data stores in practice [130, 99, 30, 111] and in the academic litera-

ture [124, 137, 114]. This deployment setting assumes multiple client applications route query requests through a single trusted proxy. The proxy manages the execution of these queries on behalf of the clients, sending queries to some storage service. Our security model and results apply equally well to a setting with a single client and no proxy.

We assume all communication channels are encrypted, e.g., using TLS. This does not prevent the storage service from seeing requests. The proxy therefore encrypts each KV pair (k, v) by applying a pseudorandom function (PRF) to the key, denoted $F(k)$, and symmetrically encrypting the value, denoted $E(v)$. We assume that the values are all the same size, perhaps via padding —i.e., there is no length leakage. The secret keys needed for F and E are stored at the proxy. Because F is deterministic, the proxy can perform operations for key k by instead requesting $F(k)$. This standard approach is used in a variety of commercial products [130, 99, 30, 111, 7].

Security model. Our security model captures passive persistent adversaries in such encrypted data store settings. The adversary observes all (encrypted) accesses but does not actively perform its own (e.g., via a compromised client).

We model honest client requests as queries sampled from a distribution π over keys: for each key k, the probability of a query (get, put, or delete) on that key is denoted $\pi(k)$. The distribution may change over time. While we primarily focus on the case where queries are independent draws from π, we discuss correlated queries and how this relates to ORAM security in Section 4.11.

In our model, the adversary does not have access to cryptographic keys, but can observe all encrypted queries to, and corresponding responses from, the stor-

age server. The adversary does not change the client queries, the responses, or the stored data. The adversary knows π, but the random draws from π that constitute individual accesses are (initially) hidden. The adversary wins if it can infer *any* information about the resulting sequence of accessed plaintext KV pairs; we formalize this further in §4.4.3. We do not target hiding the time at which a query is made; fully obfuscating timing requires a constant stream of accesses to the data store, which is prohibitively expensive in many contexts. (Our approaches can nevertheless be extended to hide timing in this way.) See §4.7 for more discussion on the limitations of our security model.

Access pattern attacks. Without further mechanisms, the basic PRF and encryption approach leaks the pattern of accesses to the adversary. In various contexts an attacker can combine this leakage with knowledge about π [100, 12, 65, 24] to mount damaging attacks like frequency analysis: order the KV pairs by decreasing likelihood of being accessed $k_1, k_2, \ldots$, and guess that the most frequently accessed encrypted value is k_1, the second most frequently accessed is k_2, etc. In general, in our security model the adversary can use knowledge of the distribution π to:

- infer key identities,
- identify when specific keys are accessed, and,
- detect and identify changes in key popularities over time.

Our goal is to protect against such access pattern attacks.

Prior approaches. Access pattern and related attacks have been treated in the literature before; we briefly overview three lines of work related to our results.

Oblivious RAMs (ORAMs): Existing ORAM designs provide security against access pattern attacks even in settings where the adversary can actively inject its own queries. The core challenge with ORAM based approaches is their overheads — several recent results [19, 82, 110, 144, 81, 108] have established strong lower bounds on ORAM overheads: for a data store with n key-value pairs, any ORAM design must either: (1) use constant proxy storage but incur $\Omega(\log n)$ bandwidth overheads; or, (2) must use $\Theta(n)$ storage at the proxy and incur constant bandwidth overheads. Unfortunately, both of these design points are inefficient for data stores that store billions of key-value pairs [52, 21, 36, 6, 139]. At such a scale, $\Omega(\log n)$ bandwidth overheads result in orders-of-magnitude throughput reduction [27]. On the other hand, state-of-the-art ORAM designs that achieve constant bandwidth overheads in theory [5] have large constants hidden within the asymptotic result (as much as 2^{100} [5]), resulting in high concrete overheads. For many applications, ORAM overheads may be unacceptable.

Snapshot attacks: Another recent line of work has targeted what's called a snapshot threat model, where the adversary does not persistently observe queries and only obtains a one-time copy (snapshot) of the encrypted data store [80, 113, 106]. One of these [80] propose frequency-smoothed encryption, a technique similar to our selective replication mechanism. Unfortunately, the snapshot threat model is currently unrealistic for existing storage systems [58]. More generally schemes designed for it do not resist access pattern attacks by passive persistent adversaries.

Fake queries: Mavroforakis et al. [86] explore the idea of injecting fake queries to obfuscate access patterns in the context of range queries and (modular) order-preserving encryption. In a security model where boundaries between the queries are not known to the adversary, this can provide security albeit with high band-

width overheads. However, if query boundaries are known to the adversary (as in our model and in practice), the adversary can trivially distinguish between real and fake queries because the last query sent is always the real one. That said, our work uses the idea of fake queries from [86], adapting it to our KV store setting (see §4.4.2) and combining it with further techniques to ensure security.

4.3 pancake Overview

We now provide a brief overview of PANCAKE's core technique — frequency smoothing. We relegate the discussion of PANCAKE's design details to subsequent sections.

Frequency smoothing. Most data stores already gather statistics about data access patterns for load balancing, debugging and performance tuning [3, 6]. PANCAKE's design exploits that all clients route their queries via the proxy; thus, the proxy can learn information about the frequency of plaintext accesses. We provide intuition on frequency smoothing technique assuming perfect estimates, i.e., $\hat{\pi} = \pi$. We start with the case that π does not change over time, and discuss the dynamic case at the end of the section.

PANCAKE uses the estimate of π to perform frequency smoothing. The key technical challenge is how to efficiently transform accesses distributed according to π over (plaintext) keys to a uniform distribution over encrypted keys. PANCAKE achieves this through a combination of three techniques: *selective replication*, *fake queries*, and *batching*. In fact, either selective replication or fake queries along with batching could be used to smooth frequency, but with prohibitive performance

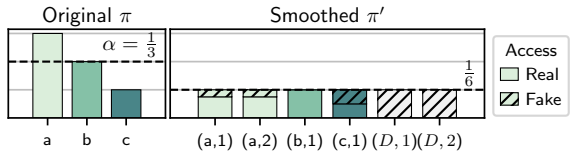


Figure 4.1: **Frequency smoothing example.** (Left) Original distribution over keys. (Right) Distribution over replicas after frequency smoothing. Ratio of real to fake accesses is the ratio of their areas.

overheads as we explain below. The trick will be combining the three together in order to achieve an efficient solution.

Selective replication creates a number of copies of a key k (called *replicas*[1]) proportional to their likelihood of access: the more likely, the more replicas. When accessing a key, one of its replicas is chosen at random. Theoretically, a value α can be selected such that each $\pi(k) = R(k) \cdot \alpha$ for some integer $R(k)$. Key k would get $R(k)$ replicas. This would smooth the distribution to uniform. However, it leads to impractical storage overheads for typical distributions — the overhead for the YCSB workload (§4.6) would be 15×.

Instead, PANCAKE creates $R(k)$ replicas of k, just enough to ensure only that $\pi(k)/R(k) \leq 1/n$ where n is the number of items in the data store. We refer to $\alpha = 1/n$ as the replica threshold. As we will show in §4.4.1, this ensures the total number of replicas n', although dependent on the distribution π itself, is always $\leq 2n$. Since an adversary may learn some distributional information from n', we add a dummy key D with $2n - n'$ replicas, so that the total number of replicas is always exactly $2n$, regardless of π. For example, given the distribution $(\frac{1}{2}, \frac{1}{3}, \frac{1}{6})$ over $n = 3$ keys a, b, c and threshold $\frac{1}{3}$, selective replication creates two replicas of key a (denoted as $(a, 1)$, $(a, 2)$), one replica each of b and c (denoted as $(b, 1)$ and $(c, 1)$, respectively) and two replicas for the dummy key D (denoted as $(D, 1)$ and $(D, 2)$). Figure 4.1 plots the frequencies.

[1]We use the term *replica* to refer to both the original key and its copies.

The resulting distribution over replicas is not quite uniform. In our example, the distribution over $(a, 1)$, $(a, 2)$, $(b, 1)$, $(c, 1)$, $(D, 1)$, $(D, 2)$ is $(\frac{1}{4}, \frac{1}{4}, \frac{1}{3}, \frac{1}{6}, 0, 0)$. PANCAKE therefore uses an equal proportion of fake queries mixed in with real ones in order to ensure a uniform distribution over accesses. To do so, PANCAKE computes a complementary fake access distribution over replicas so that the sum of the probability of a fake access and real access for any given replica is equal to $1/2n$, where $2n$ is the total number of replicas. Every time an access is made, it is chosen to be either fake or real with probability $1/2$. In our example, using a fake access distribution of $(\frac{1}{12}, \frac{1}{12}, 0, \frac{1}{6}, \frac{1}{3}, \frac{1}{3})$ across the four replicas ensures each replica has a total access probability of exactly $\frac{1}{6}$. We will show that adding fake queries in this manner always ensures equal probability for any key being accessed.

To support updates to values, every access is a read followed by a write of a freshly encrypted value. For keys with many replicas, we cannot change all replicas immediately as this would leak that these encrypted values are linked. Instead PANCAKE updates one of the replicas, caches the new value, and opportunistically updates the remaining replicas using subsequent fake or real queries to the replica. This could require a large cache in the worst case, but we show empirically in §4.6 that the cache remains small for typical workloads.

To service a (real) query from a client, PANCAKE performs a sequence of B accesses randomly chosen from either the real or the fake distribution, inserting the actual request into one of those chosen to be real. There is a small chance that the client's request cannot be served, in which case PANCAKE puts the query into a queue until the next client request arrives. We show that with $B = 3$, PANCAKE can ensure delivery of client requests in a timely manner (we make this precise in the next section), while maintaining that the probability of accessing any sequence

of B encrypted keys is equally likely.

One could achieve security without selective replication by increasing the ratio of fake queries to real queries, but a larger value of B will be needed to ensure client requests are not stalled for arbitrarily long time. This, in turn, would result in high bandwidth overheads for many distributions. Thus, the combination of selective replication and fake queries, as in frequency smoothing, is necessary to ensure small overheads. With our chosen parameters, we will prove a storage overhead of $2\times$ and a bandwidth overhead of $3\times$ of insecure KV stores, independent of the underlying distribution. Moreover, we will prove that PANCAKE's protocol is secure if the estimate $\hat{\pi}$ is sufficiently good.

Dynamic query distributions To allow PANCAKE to maintain its security and performance guarantees even when access distributions change, we extend the above design using an efficient algorithm that dynamically updates the fake query probabilities and replica allocations across keys. Recall that the total number of replicas in PANCAKE is always $2n$, regardless of the distribution. This means that when the distribution changes, for every key that must lose a replica, another must gain a replica. Therefore, handling distribution changes simply requires reassigning replicas for all such key-pairs.

PANCAKE uses a specialized replica swapping protocol to efficiently adjust the allocation of replicas in parallel with servicing client requests. The key challenge is that a request must be serviced by one of the old replicas, not a newly allocated one, until all the new replicas have the appropriate value propagated to them. We show that we can temporarily lower the ratio of real to fake queries, which, combined with appropriate temporary caching of values during the transition, maintains

the invariant that every access to the store is uniformly distributed, guaranteeing security (§4.5).

4.4 pancake Design: Static Distribution Case

We now provide details on the design and implementation of PANCAKE. In this section, we focus on the case of a static distribution, and extend PANCAKE's design to efficiently handle dynamic changes in the next section. We start the section with the data storage (§4.4.1) and frequency smoothing (§4.4.2) mechanisms in PANCAKE, and then provide a formal security analysis for PANCAKE's design (§4.4.3). We close the section with performance analysis of PANCAKE's storage requirements and bandwidth overheads for query execution (§4.4.4).

4.4.1 Data Storage

PANCAKE is backward-compatible with existing data stores — it requires no modifications on how data is sharded across multiple cores or machines, and how queries are executed in the underlying data store. Thus, PANCAKE naturally benefits from the many properties of existing data store, e.g., elasticity, fault tolerance, data persistence, etc. The core of the PANCAKE design is a proxy, which we describe below.

The pancake proxy The main functionality of the PANCAKE proxy is to initialize the data store, to implement frequency smoothing, and to execute queries on behalf of clients (encryption/decryption of query requests/responses). The proxy

maintains several data structures to achieve its functionalities:

- **Observed query distribution ($\hat{\pi}$):** The proxy maintains the probability of access for individual keys, based on the histogram of accesses across keys. This "observed" distribution is an estimate of the underlying distribution, and is also used to detect changes in distribution over time.
- **Fake query distribution (π_f):** The proxy also maintains a fake probability of access for each individual key. We will discuss below how the fake distribution is computed.
- **Key $\rightarrow$ replica counts:** PANCAKE's selective replication mechanism may create one or more replicas for KV pairs. The proxy maintains a map $k \rightarrow R(k)$ from keys to their number of replicas, for all keys with $R(k) > 1$.
- **UpdateCache:** To securely handle write queries, we use a data structure that stores a map $k \rightarrow (v, \texttt{UpdateMap})$, where `UpdateMap` is a bitmap of length $R(k)$ denoting whether or not a particular replica of k has been updated or not. We provide more details below.
- **Query queue:** This stores outstanding client queries.

The rest of the section details how the PANCAKE proxy uses these data structures to realize its functionalities. But first we make two observations about proxy storage and scalability.

Regarding PANCAKE proxy storage requirements, we note that storing the probability for a key as floating-point values requires 8 bytes of storage per key; given that the size of values in many real-world applications is of the order of kilobytes [6], storing the real and the fake distributions requires a tiny fraction of the

entire dataset size. For instance, with 4 kilobyte values, the fraction works out to a mere 0.39%. Similarly, the key $\rightarrow$ replica counts data structure is also tiny. The size of UpdateCache, on the other hand, depends on the query distribution as well as the write rates; we evaluate the UpdateCache size empirically for realistic workloads in §4.6.

The PANCAKE proxy is implemented to efficiently scale with multiple cores. For the multi-core implementation, the first four data structures are shared by all PANCAKE proxy cores, while each core maintains its own query queue (for queries "assigned" to that core). Our proxy implementation ensures high performance (highly concurrent read-write rates) for data structures shared across cores. The first three data structures are updated at coarse-grained timescales (e.g., due to significant changes in the query distributions) and thus, simple arrays suffice for our purposes. UpdateCache, on the other hand, requires concurrent read/write operations; to this end, our implementation uses a Cuckoo hashmap [84] that can support 40 million read/write operations per second on a commodity server.

4.4.2 Frequency Smoothing

We now describe PANCAKE's frequency smoothing techniques for static distributions, specifically the algorithms to initialize the data store (with selective replication) and execute queries (with real queries, fake queries, and batching).

It transforms a real query on a plaintext database DB into a fixed number of accesses (say B for "batch size") on an encrypted database DB'. By creating multiple copies ("replicas") of the most queried KV pairs and generating fake queries, we can ensure each of the B accesses are a uniformly random selection of

the items in DB'.

Initializing the data store. PANCAKE transforms a plaintext data store $DB = \{(k_i, v_i)\}$ with n KV pairs into a data store DB' with $n' \geq n$ encrypted KV pairs. At the same time, PANCAKE transforms accesses distributed according to π over the keys of DB to a sequence of uniform accesses over the encrypted keys of DB'. To distinguish between plaintext keys and encrypted ones, we refer to the latter as labels. PANCAKE use an estimate $\hat{\pi}$ of π. During initialization, $\hat{\pi}$ can be assumed to be uniform, and the techniques from §4.5 can later be used to transition to a more accurate estimate. Alternatively, in many settings one will provide a warm start by initializing PANCAKE with a $\hat{\pi}$ learned from performance or other logs.

In generating DB', we use selective replication to add replicas to DB' for keys accessed frequently according to $\hat{\pi}$. If we set a threshold α, then for each $(k, v) \in DB$ we generate $R(k, \hat{\pi}, \alpha) = \lceil \hat{\pi}(k)/\alpha \rceil$ replicas: key-value pairs $((k, j), v)$ where j ranges over 1 to $R(k, \hat{\pi}, \alpha)$. When $\hat{\pi}$ and α are clear from context, we will omit them and simply write $R(k)$.

Each replica (k, i) is then protected by applying a secretly keyed pseudo-random function F (e.g., HMAC) to the replica identifier to generate a label $F(k, i)$. We apply authenticated encryption E to the value. Thus ultimately $DB' = \{(F(k_i, j), E(v_i)\}$ for $1 \leq i \leq n$ and where $1 \leq j \leq R(k_i)$ for each i. For simplicity, we have omitted in our notation the two required cryptographic secret keys, and that we cryptographically bind labels and value ciphertexts together by using the label as associated data with E. A straightforward calculation shows that for any $\hat{\pi}$ and α, $n' \leq n + 1/\alpha$.

The second initialization task is to compute a fake distribution π_f over replicas.

Here we adapt a technique from Mavroforakis et al. [86]. In particular we pick a constant $0 < \delta \leq 1$ (this choice is explained in more detail below) and then craft π_f so that the probability $p(k, j)$ of accessing any replica (k, j) is: (1) equal to $1/n'$ and (2) a convex combination of the probability of truly accessing a replica and performing a fake access. Namely we ensure that

$$p(k, j) = \delta \cdot \frac{\hat{\pi}(k)}{R(k)} + (1 - \delta) \cdot \pi_f(k, j) = \frac{1}{n'} \ . \tag{4.1}$$

This corresponds to the following randomized process. Flip a δ-biased coin. If it comes up heads, randomly choose a replica for some real query drawn according to π; otherwise, choose a replica to access according to the fake distribution π_f.

The constant δ must be chosen so that $\delta \leq R(k)/(n' \cdot \hat{\pi}(k))$ for every key k; otherwise, it may not be possible to assign valid (non-negative) probability $\pi_f(k, j)$ to satisfy Equation 4.1 for some key k. We use $\delta = 1/(n'\alpha)$, which is always valid.

Note that δ corresponds to the proportion of real queries: if α is set too high, then most queries would be fake. At the same time, since $n' \leq 1/\alpha + n$, setting α too low would cause DB' to grow too large. We set $\alpha = 1/n$ since it corresponds to a sweet spot: $n' \leq 2n$, i.e., DB' is at most twice as large as DB, and $\delta \geq 1/2$, i.e., at least half the queries are real.

Dummy replicas. We note that the approach outlined above would result in a different number of total replicas for different distributions (although upper-bounded by $2n$), which leaks information about the distribution. To avoid this leak, PANCAKE preemptively initializes DB' with enough *dummy replicas* so that the total number of replicas is always $2n$.

Dummy replicas are KV pairs $(F(D, j), E(D))$, for $j = 1, \ldots, 2n - n'$ (n' is

```
Init(π̂, DB, α):
n ← |DB|
DB' ← ∅
n' ← 0
For (k, v) ∈ DB:
   R(k) ← ⌈π̂(k)/α⌉
   For j ∈ [1, ..., R(k)]:
      π_f(k, j) ← (α − π̂(k)/R(k)) / (2nα − 1)
      DB' ←∪ {(F(k, j), E(v))}
   n' ← n' + R(k)
For j ∈ {1, ..., 2n − n'}:
   π_f(D, j) ← α / (2nα − 1)
   DB' ←∪ {(F(D, j), E(D))}
δ ← 1 / (2nα)
Return DB', π_f, R, δ
```

```
Batch(k):
j ←$ {1, ..., R(k)}
AddToQueue(k, j)
For i = 1 to B:
   q_type ←δ {0, 1}
   If q_type = 0:
      (k_i, j_i) ←$ π_f
   Else:
      If QueueNotEmpty:
         (k_i, j_i) ← Dequeue()
      Else:
         k_i ←$ π̂
         j_i ←$ {1, ..., R(k_i)}
ℓ ← {F(k_1, j_1), ..., F(k_B, j_B)}
Return ℓ
```

Figure 4.2: **pancake's initialization and batch access algorithms** for a plaintext data store DB, distribution estimate $\hat{\pi}$, and threshold α.

the number of "real" replicas for $\hat{\pi}$), where the dummy key D is unique and does not exist in the original set of keys. Dummy replicas are accessed only with fake accesses; therefore, $\hat{\pi}(D) = 0$ and the fake access probability is $\pi_f(D) = \alpha/(2n\alpha - 1)$ (derived from Eq. 4.1). Note that since the total number of replicas is now $2n$, the proportion of real queries $\delta = 1/(2n\alpha) = 1/2$ for $\alpha = 1/n$.

A pseudocode description of PANCAKE's initialization (including dummy replicas) appears in Figure 4.2.

Query execution. Intuitively, we will follow the randomized process associated to Equation 4.1 to mix fake and real accesses. To increase the probability that a client's real access is handled right away, PANCAKE in fact sends a small batch of accesses to DB' for each client request. In particular, when a client submits an access request for key $k \in DB$, PANCAKE runs the Batch algorithm shown in Figure 4.2. It randomly chooses a replica j of k, adds (k, j) to the query queue, and prepares a batch of B accesses to DB'. By default we set $B = 3$ (we will justify our choice in §4.4.4). For each of these accesses, it samples a bit q_{type} according

to δ that determines whether the access is real (heads) or fake (tails). For each q_{type} that comes up heads (real) in the batch we attempt to send a value from the query queue. If the query queue is empty, then the client simulates a real access by sampling a key from $\hat{\pi}$ itself (denoted $k \leftarrow_{\$} \hat{\pi}$) and choosing a replica at random. For each fake access, the client samples a replica according to π_f. The resulting batch of replicas have the pseudorandom function F applied before being sent to the server. Note that Batch imposes bandwidth overhead exactly $B\times$ over a KV store that just uses encryption and leaks access patterns.

Note that the batching done in the PANCAKE proxy does not require all queries in the batch to be sent to the same shard/server; the batching is completely independent of the sharding mechanism used on the server and queries in the batch are independently forwarded to respective shards. Upon retrieving the associated values, PANCAKE decrypts the ones requested by clients and returns them.

It is critical that PANCAKE only sends a single batch for each client request. If instead the proxy sent batches until the query queue was empty, frequency information about which keys clients access would leak. For example, if one uses $B = 1$ and kept submitting until the queue is empty, then the final access to DB' must be a client request. Thus PANCAKE defers handling a query until a later batch if necessary, increasing latency. We show experimentally that for most loads this latency increase is acceptably low (§4.6.3). In practice PANCAKE can vary B as a function of load: decrease B at high load (to lower bandwidth overhead) and increase B at low load (to lower latency). Such changes to B do not reveal anything new to an adversary, who can anyway estimate aggregate load.

Supporting writes. PANCAKE handles updates (writes) to keys in DB by borrowing a standard technique from the ORAM literature [50]: treat each access as a read followed by a write. After the client receives the B encrypted values from the server corresponding to the batch, it decrypts, possibly updates, then re-encrypts the values and sends them back to the server. Each access therefore consists of a fixed-size batch of reads followed by a fixed-size batch of writes to the same labels. When a key has multiple replicas and its value is updated, the client adds it to the UpdateCache to track which of its replicas still need to be updated (updating all replicas at once leaks information). PANCAKE consults the UpdateCache every time it does a writeback to ensure all updates propagate. Once all of a key's replicas have been updated, its entry is removed from the cache. Note that PANCAKE can use any access (fake or real) to opportunistically propagate updates.

4.4.3 Security Analysis

Intuitively, PANCAKE security stems from the following three points. (1) The cryptographic security of F as a pseudorandom function and E as a (randomized) authenticated encryption scheme. This ensures that the keys $F(k, j)$ appear random and that nothing leaks about values. (2) Assuming client requests are distributed according to π and that our estimate $\hat{\pi}$ of π is sufficiently good, each individual access is uniformly distributed over DB' by Equation 4.1. (3) Fake and real queries cannot be distinguished by the server (i.e., none of the coin tosses q_{type} can be inferred). The third point requires that the number and timing of accesses observed by the server be independent of the coin tosses. We do not attempt to hide the time at which an access is made by a client, but the timing should be independent of which key a client requests and which accesses are fake or real — thus, similar

to ORAM designs [13], PANCAKE implementations must be constant-time.

Formal analysis. To provide a formal analysis, we introduce a security definition called real-versus-random indistinguishability under chosen distribution attack or ROR-CDA. A formal game-based definition of ROR-CDA is given in Appendix 4.8. Briefly, in the real world the adversary is given PANCAKE's encryption of the KV store DB' and a transcript τ generated by running Batch on Q samples from π (where Batch uses $\hat{\pi}$). In the ideal world, the adversary is given a database consisting of random bit strings and a transcript of $Q{\cdot}B$ uniformly random accesses.

Achieving this security goal rules out attacks based on access pattern leakage. Take frequency analysis as an example. If ROR-CDA holds, the frequency with which any label is accessed is independent of the label itself. Thus, frequency analysis and any other attacks which rely on computing the most likely access will fail — all accesses are equally likely, so it is impossible to do better than baseline guessing.

The following theorem establishes the ROR-CDA security of PANCAKE. The theorem reduces to the pseudorandom function security [49] of F, the real-versus-random indistinguishability [123] of E, and to the computational indistinguishability of π and $\hat{\pi}$.

Theorem 8. *Let $Q \geq 0$ and $Q = Q \cdot B$. Let $\pi, \hat{\pi}$ be distributions. For any Q-query* ROR-CDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ such that*

$$\mathbf{Adv}^{\textit{ror-cda}}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{\textit{prf}}_{F}(\mathcal{B}) + \mathbf{Adv}^{\textit{ror}}_{E}(\mathcal{C}) + \mathbf{Adv}^{\textit{dist}}_{Q,\pi,\hat{\pi}}(\mathcal{D})$$

where F and E are the PRF and AE scheme used by PANCAKE*. Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ each use Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Discussion. Details of our formal analysis, including the proof of Theorem 8, are presented in Appendix 4.8. Here we make some salient observations.

Our theorem is "parameterized" by Q, π, $\hat{\pi}$. It applies to any distribution π, and provides security up to the ability to accurately estimate it. In the best case, estimation is perfect, $\hat{\pi} = \pi$, and Theorem 8 is optimal in the sense that the only way to break PANCAKE is to break one of the underlying cryptographic tools. Even if our estimate is not perfect, it just needs to be good enough to be indistinguishable from the real distribution for a limited number of samples. While there exist distributions that are hard to estimate [67, 11, 128], real-world ones with heavy skew allow for sufficiently good estimation.

Our security model is highly pessimistic in that we assume the adversary has perfect knowledge of π. In reality they will not, and so we expect that in practice PANCAKE will provide even greater security than what our theory suggests.

4.4.4 Performance Analysis

PANCAKE incurs a bandwidth overhead of $B\times$, the size of each batch. With $\alpha = 1/n$, the server stores $2n$ replicas (including dummy replicas), so the server storage overhead is $2\times$. Note that PANCAKE bandwidth and server storage overheads are independent of the underlying data access distributions.

PANCAKE proxy storage and query latency overheads are related to query queue length, which itself is a function of batch size B. Experimentally, we observe a near-zero queue length for $B \geq 3$ (§4.6.3). This is supported by results in queuing theory: if we model the number of query arrivals per unit time as Poisson with mean λ, with $\delta = 1/2$ the number of departures per unit time with our scheme is

also Poisson with mean $\lambda \cdot B/2$. Thus, our queue is well-modeled as M/M/1 with $\rho = \lambda/(\lambda \cdot B/2) = 2/B$. Applying standard results on steady-state behavior of such queues [31], as the number of queries goes to infinity, $\Pr[\, i \text{ queries in queue}\,] = (1 - \frac{2}{B})(\frac{2}{B})^i$. Thus the probability that a query waits for i queries ahead of it in the queue is exponentially vanishing in i.

The size of PANCAKE's UpdateCache depends on the query distribution, the threshold α, and the fraction of write queries. A loose bound on UpdateCache size is the number of keys with access probability greater than α. Intuitively, a pathological worst-case could occur when $n-1$ out of n keys have access probability slightly higher than $1/n$; in this case, each of the $n-1$ keys would have 2 replicas, and UpdateCache size could grow to $O(n)$ with very high write rates. We delegate a formal analysis of the worst-case UpdateCache size for specific distributions to future work, but note that our evaluation demonstrates that, for standard benchmark workloads comprising skewed distributions, the UpdateCache size turns out to be a small fraction ($< 5\%$) of the dataset size (§4.6.3).

4.5 Handling Dynamic Distributions

In the previous section, we showed how PANCAKE transforms any static distribution of key-value accesses into a uniformly-distributed one. For some applications, however, distributions will change over time. We now describe how PANCAKE adapts to changes in the query distribution. We start by describing the core dynamic adaptation technique in PANCAKE under the assumption that changes in distribution can be detected instantaneously (§4.5.1), prove PANCAKE security under this assumption (§4.5.2), and, finally, discuss some pragmatic issues of detecting

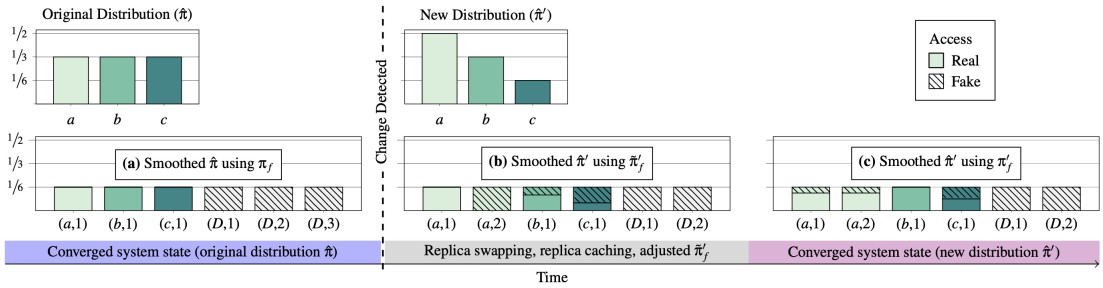


Figure 4.3: **Frequency smoothing for dynamic distributions.** (a) Smoothing for original distribution ($\hat{\pi}$) over replicas in DB' using fake distribution π_f. With $\hat{\pi}$, each of keys a, b, c has one replica, and the dummy key D has 3 replicas. (b) Detection of new distribution ($\hat{\pi}'$) over keys triggers replica swapping. During replica swapping, distribution over replicas in DB' is smoothed using an adjusted fake distribution $\tilde{\pi}'_f$: all real accesses to a are directed to $(a, 1)$, and the real access probability is decreased until $(D, 3)$ and $(a, 2)$ are swapped. (c) Smoothing for new distribution ($\hat{\pi}'$) using new fake distribution π'_f, after replica swapping completes. Key a gains a replica, while D loses a replica.

changes in the underlying distribution (§4.5.3).

4.5.1 Adapting to Changes in Distribution

Once the new query distribution estimate $\hat{\pi}'$ is identified, PANCAKE must adapt to $\hat{\pi}'$ by smoothing it. We note that if all keys need the same number of replicas with $\hat{\pi}'$ as they need with $\hat{\pi}$, PANCAKE easily adapts to $\hat{\pi}'$ by recomputing the fake query distribution π_f as per Equation 4.1. However, when a key's probability $\hat{\pi}'(k)$ increases so much that $\hat{\pi}'(k) \geq R(k, \hat{\pi}, \alpha) \cdot \alpha$, then PANCAKE must change its number of replicas. Figure 4.3 shows an example for frequency smoothing of $\hat{\pi}$ and $\hat{\pi}'$; note that while key a gains a replica, the dummy key D loses one.

Adapting to changes in the query distribution while preserving both efficiency and security is challenging. One approach is downloading the entire database and re-running Init from Figure 4.2 with fresh keys. This is secure but prohibitively bandwidth-intensive, and queries cannot be serviced during reinitialization. One

could instead act only on the replicas for keys whose probabilities have changed; this is insecure since accesses are non-uniform during the change. In Figure 4.3, if we only download a, add a new replica for it and delete one for D, then an adversary can infer that a grew in popularity.

Our solution builds on the latter approach, ensuring efficiency and security using an *online replica swapping* mechanism described next. To make replica swapping performant and secure, it must work in conjunction with two other techniques: adjusting the fake query distribution and caching replicas at the proxy.

Replica swapping. Our key insight in adapting to change in query distributions is that since the total number of replicas for any distribution is always exactly $2n$ (including dummy replicas), a transition from $\hat{\pi}$ to $\hat{\pi}'$ ensures that for each key k_i that gains a replica, there must be another key k_j that loses a replica. Therefore, handling changes in query distributions simply requires, for all such keys, reading k_j's value and writing it to one of k_j's replicas, a process we refer to as replica swapping. PANCAKE performs these swaps without revealing any information about the change by *piggybacking* the replica swaps atop normal client accesses. Maintaining uniform accesses during replica swapping requires changes to the Batch procedure and the fake access distribution as described in §4.4; we describe these changes next.

During replica swapping, the modified Batch uses two lists: G and L. G is the set of replicas that need to be created and L is the set of replicas that need to be removed. Formally, if S is the set of keys that must gain replicas, T is the set of

keys that must lose replicas, and $R(k, \hat{\pi}', \alpha) = \lceil \hat{\pi}'(k)/\alpha \rceil$, then,

$$\mathsf{G} = \{(k, j) \mid k \in S,\ j \in [R(k, \hat{\pi}, \alpha) + 1, \ldots, R(k, \hat{\pi}', \alpha)]\}$$

$$\mathsf{L} = \{(k, j) \mid k \in T,\ j \in [R(k, \hat{\pi}', \alpha) + 1, \ldots, R(k, \hat{\pi}, \alpha)]\}$$

A pseudocode procedure for generating these lists from $\hat{\pi}$ and $\hat{\pi}'$ is given in Figure 4.17 in Section 4.12, along with a description of the modified Batch. It is not hard to see that $|\mathsf{G}| = |\mathsf{L}|$ always (since $|S| = |T|$), and that swapping each replica in L for one in G results in all keys having the right number of replicas under $\hat{\pi}'$. This swapping is done opportunistically by Batch: when a replica in L is read in a batch, either by a real or a fake query, its value is updated to the value associated with a replica in G during the writeback. For security reasons, PRF labels for replicas in G are not changed. Instead, PANCAKE maintains a mapping between the label of replicas in L and the replica in G it will be swapped with. On subsequent queries during the transition, Batch consults the mapping for the right labels. This metadata can be deleted after periodic rotation of the cryptographic keys. We describe key rotation in Section 4.9. When all swaps have occurred, we switch back to the normal Batch procedure for $\hat{\pi}'$.

As a concrete example of replica swapping, consider Figure 4.3. The set G contains the replica $(a, 2)$, while L contains $(D, 3)$. Note that both G and L could contain dummy replicas, depending on how the distribution changes. Batch would swap the replicas for keys a and D on the first access to $(D, 3) \in \mathsf{L}$ by writing back an encryption of key a's value (because $(a, 2) \in \mathsf{G}$) instead of a re-encryption of the dummy value D. To enable this, PANCAKE would maintain a mapping that indicates the label of $(a, 2)$ is $F(D, 3)$.

Adjustments to fake access distribution. Two more modifications are needed during the transition. First, we must use a different fake access distribution to ensure that reads to keys that have gained replicas always succeed. To see why this is necessary, consider again the example in Figure 4.3. If a query tries to read key a by accessing replica $(a, 2)$ before the value of $(D, 3)$ is changed, the read will return D's value instead of a's. Thus replica $(a, 1)$ must be read, but forcing this makes $(a, 1)$'s probability too high, violating security.

PANCAKE handles this by temporarily increasing the threshold α to $\alpha' = \max_k\{\pi'(k)/R(k, \hat{\pi}, \alpha)\}$, and using a temporary fake access distribution $\tilde{\pi}'_f$ to satisfy Equation 4.1 with α'. For each $(k, j) \in \mathsf{G}$, we set $\tilde{\pi}'_f(k, j) = \frac{\alpha'}{2n\alpha'-1}$, and k's existing replicas have $\tilde{\pi}'_f = \frac{\alpha'-\hat{\pi}'(k)/R(k,\hat{\pi},\alpha)}{2n\alpha'-1}$. For other keys, $\tilde{\pi}'_f(k, j)$ is set to $\frac{\alpha'-\hat{\pi}'(k)/R(k,\hat{\pi}',\alpha)}{2n\alpha'-1}$.

Since $\alpha' \geq \alpha$, the real access probability $\delta = 1/2n\alpha'$ is lower during replica swapping. As such, this may lead to some real queries being delayed to later batches. This may increase latency for some queries during replica swapping, but we show in §4.6.2 that replica swapping completes in a few minutes even for drastic changes in the distribution.

Replica caching. PANCAKE computes the mapping between each label in L and the replica in G it will be swapped with when the distribution change is detected. However, the actual values of replicas in G must be propagated to those in L during subsequent accesses to them. Without any additional mechanism, reads to keys with replicas in G may access incorrect values. To ensure correctness, when a replica in G is read during Batch, its value is cached at the proxy. This value is then propagated to the replica in L when it is next accessed, while the actual read

is served from the cache.

Insertion and deletion of keys. We have assumed so far that the support size is fixed; interestingly, the replica swapping procedure can support changes in the set of keys. This can be viewed as a distribution change where $\text{supp}(\hat{\pi}') \neq \text{supp}(\hat{\pi})$. As long as PANCAKE is initialized with enough replicas to handle the maximum support size, new keys can be inserted by swapping a dummy replica for the new key, and vice versa for deletion. Some additional metadata is needed, but similar to the PRF label mapping it can be deleted as soon as cryptographic keys are rotated (see Section 4.9).

4.5.2 Security Analysis

We prove that PANCAKE's accesses remain uniform even for time-varying distributions, under the assumption that changes in distributions can be detected instantaneously. We formalize our goal as a generalization of the static ROR-CDA security notion. We call this new notion "real-or-random security under chosen dynamic distribution attack", or ROR-CDDA. It is similar to its static analogue except that it uses two distributions π and π': after an adversarially chosen number of queries the distribution changes from π to π'. We let $\mathbf{Adv}^{\text{ror-cdda}}(\mathcal{A})$ be the ROR-CDDA advantage of an adversary $\mathcal{A}$. It captures the ability of $\mathcal{A}$ to distinguish between a real PANCAKE execution during a distribution change (ROR-CDDA1) and uniformly random accesses (ROR-CDDA0). The following theorem captures the ROR-CDDA security of PANCAKE.

Theorem 9. *Let $Q \geq 0$ and $Q = Q \cdot B$. Let $\pi, \pi', \hat{\pi}, \hat{\pi}'$ be distributions. For any Q-query* ROR-CDDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$*

such that

$$\mathbf{Adv}^{ror\text{-}cdda}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{prf}_{F}(\mathcal{B}) + \mathbf{Adv}^{ror}_{E}(\mathcal{C}) + \mathbf{Adv}^{dist}_{Q,\pi,\hat{\pi}}(\mathcal{D}_1) + \mathbf{Adv}^{dist}_{Q,\pi',\hat{\pi}'}(\mathcal{D}_2)$$

where F and E are the PRF and AE scheme used by PANCAKE. *Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$ each use at most Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Discussion Full details of the definitions and a proof of Theorem 9 appear in Appendix 4.8. We discuss here only one salient point regarding ROR-CDDA. ROR-CDDA models the shift from π to π' as happening and being detected instantaneously. This may not be realistic in some cases, even with state-of-the-art techniques in detecting distribution changes (as used in PANCAKE, discussed in next subsection). Thus, we cannot rule out the case where PANCAKE processes queries from π' before the change is detected (treating them like samples from $\hat{\pi}$). The distribution of these queries would be non-uniform, with bias related to the difference between π and π'. If the adversary knows the bias, using it in an attack would be possible but challenging—indeed, we are not aware of any published attacks that even consider the possibility of distribution changes.

4.5.3 Detecting Changes in Query Distribution

Detecting distribution changes using statistical tests is a well studied problem [75, 133, 145, 72]. While it is possible to have PANCAKE receive external signals (e.g., from an analyst) when the distribution changes, our implementation incorporates the two-sample Kolmogorov–Smirnov (KS) test [75, 133], a standard statistical tool, to detect such changes automatically. Specifically, recall that

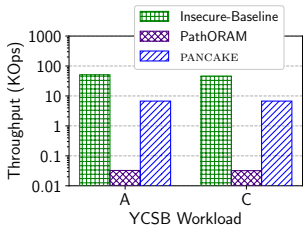
PANCAKE maintains a histogram H of observed accesses to maintain an estimate $\hat{\pi}$ for distribution π. In order to track changes to the distribution, PANCAKE additionally maintains a running histogram $H_{running}$ over a sliding window of the w latest accesses at the proxy. PANCAKE then uses KS test to determine when the underlying distribution corresponding to $H_{running}$ differs from $\hat{\pi}$. If the test indicates a change, PANCAKE uses the current $H_{running}$ snapshot to inform the estimate $\hat{\pi}'$ for the new distribution π'.

Detecting changes in distributions, and responding to these changes involves balancing security and efficiency. If the test is too sensitive the system will waste resources adjusting to spurious changes; on the other hand, as discussed above, an insensitive test could leak information about queries. While it is possible to use other statistical tests [145], or an ensemble of tests to navigate this trade-off between performance and security, no statistical test is perfect. We present several evaluation results related to detecting and adapting to changes in query distribution, along with sensitivity analysis, in §4.6.2.

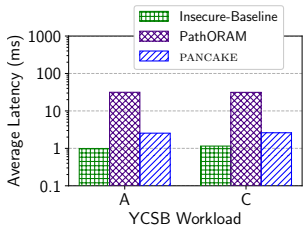
4.6 Evaluation

We now evaluate PANCAKE across a wide variety of scenarios, including main-memory and secondary storage-based data stores, static and dynamic distributions, deployment settings and workloads. We start by briefly describing the evaluation methodology, followed by detailed discussion of our results.

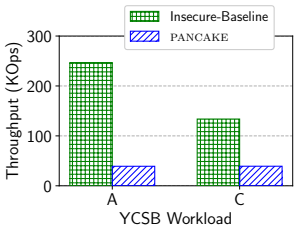
Compared approaches We compare PANCAKE against two approaches: (1) an insecure baseline that provides no security guarantees, and (2) non-recursive



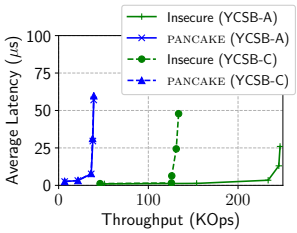
(a) Throughput (single proxy thread)



(b) Latency (single proxy thread)



(c) Throughput (multiple proxy threads)



(d) Throughput vs latency (multiple proxy threads)

Figure 4.4: **Performance for in-memory server storage (Redis).** (a, b) PANCAKE's throughput is over 220× higher than PathORAM and within ∼6.8 – 7.6× of the insecure baseline for a single-threaded proxy; note that the y-axis is in log-scale. (c, d) With multiple proxy threads, PANCAKE's peak throughput is within 3.4 – 6.3× and latency within 2.3 – 2.6× of the insecure baseline.

PathORAM [138] (with $Z = 4$), a state-of-the-art ORAM. The former serves as an upper bound on PANCAKE performance, since it corresponds to a data store with no security overheads. The latter, on the other hand, is the state-of-the-art design that provides security under our model (as well as under stronger models where an adversary can actively inject its own queries). As discussed earlier, our comparison against the latter should be interpreted as highlighting the huge efficiency gap between countermeasures in the two threat models. We use batch size $B = 3$ for PANCAKE's Batch algorithm.

We compare these approaches using two representative storage backends: an in-memory KV store Redis [120], and a persistent SSD-based KV store RocksDB [122]. Our PathORAM deployment used an open-source implementation [127, 27]. For PathORAM and PANCAKE, client queries are forwarded to the data store via a

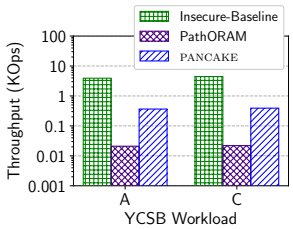
proxy server; for the insecure baseline, client queries are forwarded to the backend storage server without any intermediary proxy.

The PathORAM implementation used in our evaluation [127, 27] is single-threaded. TaoStore [124] and ConcurORAM [26] implement multi-threaded PathORAM; we omit results for them since they employ specialized storage backends adapted for ORAMs, eschewing fair comparison with backends we investigate. We note, however, that the performance reported in [124, 26] is at least an order of magnitude lower than PANCAKE even with specialized storage backends.

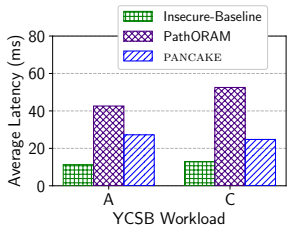
Experimental setup Our experiments run on Amazon EC2. The storage backend runs on a single t3.2xlarge instance with 8 vCPUs, 32GB RAM, and 1Gbps network and disk bandwidth. We use 1Gbps links and proxy/client machines with sufficient resources (r4.8xlarge instances with 32 vCPUs, 244GB RAM, 10Gbps network bandwidth) to highlight the impact of network bandwidth as a bottleneck.

Dataset and workloads We use the Yahoo! Cloud Serving Benchmark (YCSB) [32], a standard benchmark for KV stores, to generate the datasets and workloads. The dataset contains 2^{20} KV pairs, with 8B keys and 1KB values. We confine our dataset size to 1GB since PathORAM has prohibitively large initialization times ($>$ 24 hours) and storage overheads ($> 10\times$) with larger datasets, while PANCAKE performance is essentially independent of dataset size.

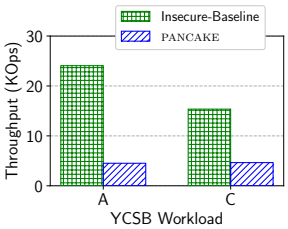
We evaluate system throughput and latency using two YCSB workloads: Workload A (50% reads, 50% writes) and C (100% reads). These workloads represent two extremes in read-write proportions; other YCSB workloads either have in-



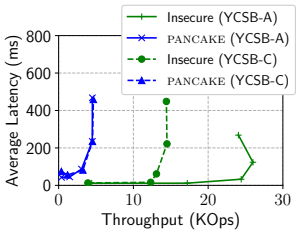
(a) Throughput (single proxy thread)



(b) Latency (single proxy thread)

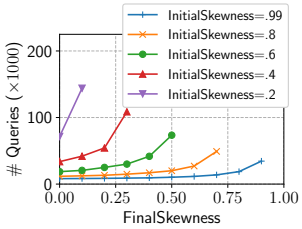


(c) Throughput (multiple proxy threads)

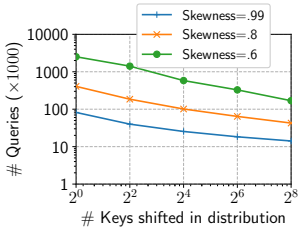


(d) Throughput vs latency (multiple proxy threads)

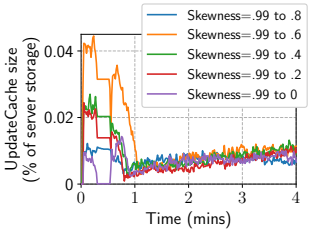
Figure 4.5: **Performance for SSD-based server storage (RocksDB).** (a, b) PANCAKE's throughput is 17.3× higher than PathORAM and within ∼10.7 − 11.3× of the insecure baseline for a single-threaded proxy; note that the y-axis is in log scale for (a). (c, d) Using multiple proxy threads, PANCAKE's peak throughput is within 3.3 − 5.3× and average latency within 2 − 2.4× of the insecure baseline.



(a) Detecting change in Zipf skew



(b) Detecting shift in key popularities



(c) Adapting to distribution change

Figure 4.6: **Handling dynamic distributions.** (a, b) PANCAKE detects larger distribution changes in fewer queries, relative to smaller changes. (c) PANCAKE can adapt from a skewed to uniform distribution with UpdateCache size < .05% of server storage over evaluated workloads.

termediate read-write proportions (e.g., Workload B, D) or contain queries not supported by PANCAKE (e.g., Workload E). YCSB uses a Zipf distribution over key accesses (with skewness parameter = 0.99, i.e., very skewed), which is representative of access patterns in real-world deployments [32].

4.6.1 Performance for Static Distributions

We first compare the performance for different approaches with various storage backends under static query distributions.

In-memory server storage (Redis, Figure 4.4) With a single proxy thread, PANCAKE and PathORAM performance is bottlenecked by the proxy. For this evaluation setting, PathORAM achieves throughput ∼1600× lower compared to the insecure baseline. This is because PathORAM issues 160 storage backend requests ($= Z \log_2 N$, $Z = 4$, $N = 2^{20}$) for every client request, along with complex tree and stash management.

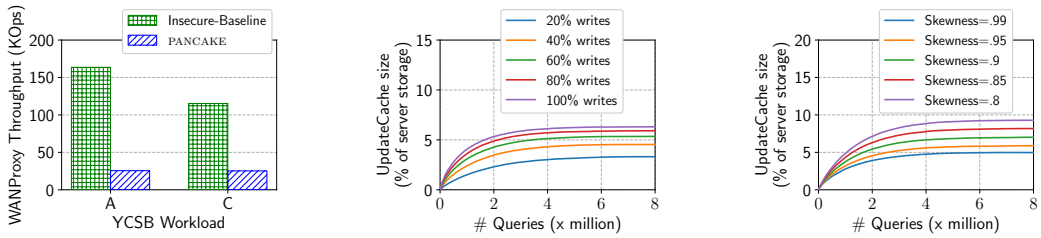
PANCAKE achieves significantly better throughput (as much as 229× better) compared to PathORAM. In comparison to the insecure baseline, PANCAKE average latency is within 2.3 – 2.6× and throughput is within 6.8 – 7.6× (Figure 4.4(a), 4.4(b)). This is a cumulative effect of three factors: (1) 3× bandwidth overhead due to batch size $B = 3$, (2) 2× overhead since each request generates a read and a write request in PANCAKE, and (3) overheads due to encryption/decryption. Our evaluation confirms that adding encryption/decryption to the insecure baseline brings PANCAKE's relative throughput overhead to 6×. We note that PANCAKE's 99th percentile latency (not shown in graphs) is relatively higher (within 4.1 – 5.6× the insecure baseline) due to queueing delays from PANCAKE's Batch algorithm. We note that if reducing tail latency were the goal, one can achieve that at the cost of higher bandwidth overheads by increasing B (§4.6.3).

With multiple proxy threads, PANCAKE peak throughput is within 3.4× of baseline for the read-only workload (YCSB Workload C) — a factor of 2 better

than the single proxy thread. This reduction in relative overhead is due to the shift in performance bottleneck to the network bandwidth in the multi-threaded setting. We note that all network links are *full-duplex*. As such, although every read request generates a read and a write request in PANCAKE, write requests saturate the network bandwidth *to* the server, while read responses saturate the bandwidth *from* the server, i.e., reads and writes saturate different directions of the link. In contrast, the read-only workload for the insecure baseline is only able to saturate one direction of the link. For the 50% read, 50% write workload (YCSB Workload A), PANCAKE's throughput remains the same, while baseline throughput increases by ∼1.8×, since the baseline can now also exploit full-duplex links. The throughput versus latency variation (Figure 4.4(d)) shows that the throughput reported in Figure 4.4(c) corresponds to the knee point in the curve (i.e., the sweet spot for latency and throughput) for both the insecure baseline and PANCAKE.

SSD-based server storage (RocksDB, Figure 4.5) With RocksDB, PathORAM achieves throughput ∼196× worse than the insecure baseline. Compared to the in-memory case, the slight improvement relative to the insecure baseline is due to PathORAM overheads overlapping with higher overheads of accessing data off SSD. As such, the proxy overheads account for a smaller fraction of the end-to-end performance. PANCAKE's performance is ∼17.3× better than PathORAM and within ∼11.3× of the baseline.

With multiple proxy threads, PANCAKE peak throughput is within 3.3× of the insecure baseline for read-only workload and within 5.3× for the 50% read, 50% write workload similar to the in-memory case. Figure 4.5(d) confirms that throughput in Figure 4.5(c) corresponds to the performance knee-point.



(a) Effect of proxy location on throughput

(b) Effect of Write-rate on UpdateCache

(c) Effect of Skew on UpdateCache

Figure 4.7: (a) Due to asymmetric and unpredictable available download and upload speeds over the Internet, both the insecure baseline and PANCAKE observe reduced throughput (0.65 – 0.85×) for the WAN setting when compared to the cloud setting. (b, c) UpdateCache size increases with write rate for a fixed Zipf distribution (skewness = 0.99) and decreases as skew increases for a fixed write-rate (50%), but remains well below 10% of server storage for all evaluated workloads.

Storage overheads PANCAKE's server storage requirements are ∼4× lower than the non-recursive PathORAM implementation that we use ($= 2 \cdot Z \cdot N$, for $Z = 4$) and within 2× of the insecure baseline, consistent with the theoretical storage overheads for both approaches. PANCAKE's proxy storage requirement is a small fraction of the total storage footprint (∼1%), similar to PathORAM (∼0.33%) for all evaluated workloads. PANCAKE proxy storage overheads due to the UpdateCache is dependent on write-rates and skew in the distribution; we evaluate these in §4.6.3.

4.6.2 Adapting to Dynamic Distributions

We evaluate PANCAKE's ability to detect and adapt to changes in distribution in isolation (i.e., without the effect of writes) using YCSB Workload-C (read-only). We present results for the in-memory storage backend. We set the sliding window size w for the running histogram $H_{running}$ to 10 million queries, and the confidence interval for the KS test to 95%.

Detecting distribution change We quantify the cost of detecting distribution change in terms of the number of queries that must be observed before the change is detected. Figure 4.6(a) measures this as the skewness for the Zipf distribution is varied; as expected, the test detects *larger* changes in distribution (e.g., skewness drop from 0.99 to 0.0) in *fewer* queries, relative to much smaller changes (e.g., skewness drop from 0.99 to 0.9). This is consistent with the KS test's sensitivity.

In Figure 4.6(b), we shift the Zipf key popularities by κ, i.e., the most popular key becomes the κth most popular key, the second most popular key becomes the $(\kappa+1)^{th}$ most popular, and so on, while the κ least popular keys become the most popular keys. This models changes in real-world access patterns where some items can suddenly become more popular [6]. Again, we observe that larger changes in distribution (e.g., shift by $\kappa = 256$) is detected in fewer queries (e.g., a few hundred thousand) than smaller changes (e.g., shift by $\kappa = 1$, which may require millions of queries).

The results for both settings show that PANCAKE's mechanism for detecting changes in distribution works well in practice, e.g., at 100K queries per second, PANCAKE can detect changes in $1-2$ seconds.

Adapting to distribution change We evaluate PANCAKE overheads in adapting to dynamic distributions when the underlying distribution changes. In particular, we change the distribution from high skewed (skewness parameter = 0.99) to smaller skews, with the extreme case of a pure uniform access pattern (skewness parameter = 0).

Our results show that PANCAKE can adapt to even drastic changes in distribution (Zipf to pure uniform) in less than ~ 25 minutes (for updating newly assigned

replicas across all keys), while using $< 0.05\%$ of the server storage at the proxy (Figure 4.6(c)). This is interesting for two reasons: (1) PANCAKE observes only a negligible increase in proxy storage during the adaptation period, and (2) the adaptation occurs in the background, i.e., without stopping query execution, and in fact piggybacks on the query execution to carry out the adaptation. As such, higher query rates would lead to even faster adaptation to changes in distribution.

4.6.3 Performance Sensitivity to Parameters

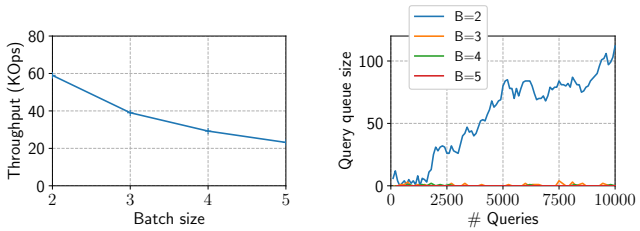
We now analyze the sensitivity of PANCAKE's performance and storage overheads to various parameters. We highlight differences in our experimental setup wherever necessary.

Effect of proxy location (Figure 4.7(a)) We measure the impact of proxy location relative to the storage server by placing the proxy in a university network, connected to the cloud storage via the Internet. The proxy server has a 16-core 2.60GHz Intel Xeon CPU, 128GB RAM and 1Gbps access link to the Internet. Figure 4.7(a) measures the throughput for this setup (which we refer to as WAN proxy) using multiple proxy threads. The throughput for WAN-Proxy is slightly lower than Cloud-Proxy (Figure 4.4(c)), since the available bandwidth over the Internet was lower than 1Gbps and often unstable. Moreover, the measured upload bandwidth was lower than the download bandwidth over the Internet, which resulted in slightly lower throughput ($\sim 0.65\times$) for PANCAKE, and the insecure baseline for Workload A (50% reads, 50% writes).

Impact of write rates (Figure 4.7(b)) and request distributions (Figure 4.7(c)) on UpdateCache We quantify the proxy storage overhead due to UpdateCache by measuring its size for varying fractions of write rates and for varying skew in underlying distribution across keys. Figure 4.7(b) shows that UpdateCache size is well below 10% of the server storage even with 100% writes. For the more realistic case of lower write rates, the storage overhead is much lower, e.g., with 20% write rate, the overhead reduces to $< 3\%$ of server storage.

While most real-world distributions are heavily skewed [6], the fraction of keys with > 1 replica in PANCAKE increases with decrease in skew. This can lead to an increase in UpdateCache size, since PANCAKE caches values for such keys while propagating writes to their replicas. We evaluate this overhead by measuring UpdateCache size for workloads with different degrees of skew for the YCSB Workload A (50% read-50% write). Figure 4.7(c) shows that decreasing skewness from 0.99 to 0.8 increases UpdateCache size from 5% to 9% of server storage, i.e., UpdateCache size remains a small fraction of server storage even at low skew.

Effect of batch-size B (Figure 4.8(a)-4.8(b)) Recall from §4.4.4 that, for a batch size of B, PANCAKE incurs bandwidth overhead of $B\times$; Figure 4.8(a) shows that when network bandwidth is the bottleneck, PANCAKE throughput degrades proportionally to the value of B. At the same time, larger B values leads to lower tail latency, since requests wait in the query queue for fewer batches — while $B = 2$ leads to an unstable queuing system (Figure 4.8(b)), $B > 2$ observes little or no queuing delays. B thus exposes a tradeoff between tail latency and throughput, where $B = 3$ provides a sweet spot for both. We do not evaluate latency vs. batch size since latency is tied to query inter-arrival times. For fixed inter-arrival times, latency overheads can be extrapolated from Figure 4.8(b).



(a) Effect of batch size on throughput

(b) Effect of batch size on query queue

Figure 4.8: (a, b) Impact of batch size on PANCAKE throughput and query queue size. See §4.6.3 for discussion.

4.7 Discussion

PANCAKE is a first step toward designing high-performance data stores that are secure against access pattern attacks by passive persistent adversaries. In this section, we discuss several possible avenues for future research.

Correlated accesses Our security analysis for PANCAKE relies on the assumption that queries are independent; in some application contexts, queries can be correlated. To the best of our knowledge, frequency analysis for correlated queries has not been explored. We present some preliminary results in the full version [53]; specifically, we show that security in a variant of ROR-CDA that allows arbitrary correlations is equivalent to ORAM security, and must therefore suffer from the same lower bounds on ORAM efficiency. However, this result relies on the adversary being able to construct very specific and artificial query correlations. We believe that we need new technical tools to explore access patterns attacks under realistic query correlations.

Stronger adversaries PANCAKE targets a security model where the attacker does not tamper with data or do rollback attacks. PANCAKE's use of authenticated

encryption means tampering is detectable, and preventing rollbacks is possible via authenticated operation counters. However, unlike ORAM, PANCAKE does not provide security against adversaries that can inject their own queries [24, 146]. We discuss how such chosen-query attacks could work on PANCAKE, and how it mitigates such attacks to some extent in Section 4.10.

Informally, we show that PANCAKE does no worse than other efficient schemes against such attacks.

Dynamic distributions For the case of dynamic distribution, PANCAKE's security is proven under the assumption that changes in distribution happen instantaneously and can be detected instantaneously. While our evaluation suggests that PANCAKE can detect changes in distribution within a few seconds, it would be nice to generalize our analysis to capture more gradual changes in distribution.

Improved proxy implementation The current PANCAKE implementation uses a stateful proxy that stores distributions ($\hat{\pi}$, π_f), key→replica counts, and pending writes in the UpdateCache. It would be interesting to explore implementations that allow the proxy to be more scalable (e.g., using a distributed proxy implementation) and fault tolerant (e.g., using techniques similar to [33]).

Variable-sized values Similar to existing ORAM designs, to avoid attacks based on length leakage, current PANCAKE design assumes that values stored in the data store are fixed-size or have been padded to a fixed maximum length. While this is useful for many applications (e.g., values have fixed size when storing tweets, and storage systems like DynamoDB have upper bounds on value sizes), forcing

values to be padded can cause prohibitive space overheads if there is a large difference between the largest and smallest values. It would be interesting to extend PANCAKE design to avoid storage overheads while protecting against attacks based on length leakage.

Hiding access patterns in cache-based systems Many real-world systems execute queries on SSD-based storage with in-memory cache (e.g., MySQL server with memcached as a cache [97]). The problem of hiding access pattern seems to be at odds with achieving high performance in such deployment settings — intuitively, for workloads with skewed access patterns, it is possible to achieve performance gains by serving popular keys from the faster cache [147] at the cost of leaking that keys in cache are accessed more frequently than those not in cache. Hiding access patterns requires all keys to be accessed uniformly thus invalidating the benefits of a cache without any additional mechanism. Our preliminary evaluation, presented in the full version [53], shows that depending on the distribution and available cache size, existing systems including PANCAKE can observe as much as an order-of-magnitude throughput degradation compared to the insecure baseline that can effectively exploit the benefits of cache. It would be interesting to explore techniques that avoid such performance degradation while providing security against access pattern attacks.

4.8 Security Proofs

In this appendix, we give some technical preliminaries and then prove Theorems 8 and 9.

Technical preliminaries. Throughout, we will use the concrete security approach [10]. For a (keyed) function $F\colon \mathcal{K} \times \{0,1\}^* \to \{0,1\}^m$ and adversary $\mathcal{A}$, we define the *pseudorandom function* (PRF) advantage relative to two games. In game PRF1, $\mathcal{A}$ has access to an oracle that accepts inputs from $\{0,1\}^*$ and outputs the PRF value on that point and a uniformly random key (which is the same for all queries). In game PRF0, $\mathcal{A}$'s oracle is a (lazy-sampled) random function from $\{0,1\}^*$ to $\{0,1\}^m$. We define $\mathcal{A}$'s PRF advantage to be

$$\mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{A}) = \left|\Pr\left[\,\mathrm{PRF1}^{\mathcal{A}} \Rightarrow 1\,\right] - \Pr\left[\,\mathrm{PRF0}^{\mathcal{A}} \Rightarrow 1\,\right]\right|$$

where the probability is taken over the random choice of key (in PRF1) or lazy-sampled random function (in PRF0) and the adversary's internal random coins. Below, we will leave implicit the coin spaces involved in probabilities.

An *authenticated encryption with associated data* (AEAD) scheme $E = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is a triple of algorithms. The function $E.\mathsf{KeyGen}$ takes no inputs and outputs elements of $\mathcal{K}$. The function $E.\mathsf{Enc}$ takes a key from $\mathcal{K}$, a plaintext from $\mathcal{M}$, (optionally) some associated data from $\mathcal{A}$, and outputs ciphertexts in $\mathcal{C}$. The function $E.\mathsf{Dec}$ takes a key from $\mathcal{K}$, a ciphertext from $\mathcal{C}$, (optionally) some associated data from $\mathcal{A}$, and outputs a plaintext in $\mathcal{M}$ or $\perp$.

We additionally require AEAD schemes to have a function len which takes a positive integer ℓ representing a plaintext length and outputs the length of any ciphertext of a plaintext of length ℓ. Essentially, the length of any plaintext's ciphertext must be computable given *only* the plaintext length and nothing else. Most natural AEAD schemes have this property.

For an AEAD scheme E and adversary $\mathcal{A}$, we define the *real-or-random* (ROR) advantage of $\mathcal{A}$ against E relative to two games, ROR1 and ROR0. In the first $\mathcal{A}$ has access to an $E.\mathsf{Enc}$ oracle with uniformly random key, and in the second

$\mathcal{A}$'s oracle returns uniformly random bit strings of length $\text{len}(|m|)$ where $|m|$ is the length of the input. We define $\mathcal{A}$'s ROR advantage against E as

$$\mathbf{Adv}_E^{\text{ror}}(\mathcal{A}) = \left|\Pr\left[\,\text{ROR1}^{\mathcal{A}} \Rightarrow 1\,\right] - \Pr\left[\,\text{ROR0}^{\mathcal{A}} \Rightarrow 1\,\right]\right| \ .$$

For a distribution π and adversary $\mathcal{D}$ that outputs a bit, let $\text{DIST}_{q,\pi}^{\mathcal{D}}$ be the game that samples q times from π, runs $\mathcal{D}$ on the resulting outputs, and outputs $\mathcal{D}$'s output. For two distributions π, π' with $\text{supp}(\pi) = \text{supp}(\pi')$, we measure their Q-sample indistinguishability from an adversary $\mathcal{D}$ via the advantage measure

$$\mathbf{Adv}_{Q,\pi,\pi'}^{\text{dist}}(\mathcal{D}) = \left|\Pr\left[\,\text{DIST}_{q,\pi}^{\mathcal{D}} \Rightarrow 1\,\right] - \Pr\left[\,\text{DIST}_{q,\pi'}^{\mathcal{D}}\,\right]\right| \ .$$

This just captures the computational indistinguishability of the two distributions, given Q samples from them.

Frequency smoothing KV schemes. Recall from §4.4 that PANCAKE has two algorithms: Init and Batch. To model distribution estimation errors and adjustments made when distributions change (as per §4.5), we extend our formalism by introducing two further algorithms. More precisely, an encrypted KV scheme $\mathsf{EKV} = (\text{Init}, \text{Batch})$ is a pair of algorithms:

- A randomized initialization algorithm Init that takes as input an estimated distribution $\hat{\pi}$, a KV store DB, and a threshold α, and outputs an encrypted KV store DB', a fake distribution π_f, a function R, and a real query probability δ. We denote running this algorithm by $(DB', \pi_f, R, \delta) \leftarrow\$ \text{Init}(\hat{\pi}, DB, \alpha)$.
- A randomized, stateful batch query algorithm Batch that takes as input a key k, the function R that maps keys to replica counts, and outputs a batch of B labels. We denote running this algorithm by $(\ell_1, \ldots, \ell_B) \leftarrow\$ \text{Batch}(k)$.

Note that to avoid notational clutter we omit from the notation the values $\hat{\pi}, \pi_f, \delta$ and the state that Batch relies upon.

We have assumed distributions have efficient representations, and abuse notation by using the same variables π, $\hat{\pi}$, π_f, etc., as both distributions and their representations. For any fixed distribution π, we assume that Init always outputs an encrypted KV store of a constant size n'. PANCAKE satisfies these assumptions; its algorithms were described in the body.

Notice that our formalization here only handles read queries. As discussed in the body, we perform writes by always doing write-backs. Thus, security analysis can be reduced to the read-only case, greatly simplifying our formalization and security definitions.

Security for static distributions. We now formalize our ROR-CDA definition for a fixed scheme $\mathsf{EKV} = (\text{Init}, \text{Batch})$. We measure the success of an adversary $\mathcal{A}$ in attacking EKV by its ability to distinguish between the games ROR-CDA1 and ROR-CDA0 as defined in Figure 4.9. The game ROR-CDA1 is parameterized by the number of queries Q, the true distribution π and the estimated distribution $\hat{\pi}$. We also take α as an implicit parameter. The adversary runs and chooses a plaintext distribution, then Init is executed followed by a sequence of queries drawn according to π. A transcript of accesses is generated by Batch. The adversary runs again with input the encrypted database and transcript. The two adversary executions can share state.

In game ROR-CDA0, the adversary sees a randomly generated encrypted database and queries chosen uniformly at random. The advantage of $\mathcal{A}$ with q

ROR-CDA1$^{\mathcal{A}}_{Q,\pi,\hat{\pi}}$:	ROR-CDA0$^{\mathcal{A}}_{Q}$:
$DB \leftarrow\$ \mathcal{A}_1$	$DB \leftarrow\$ \mathcal{A}_1$
$(DB', \pi_f, \delta) \leftarrow\$ \mathrm{Init}(\hat{\pi}, DB, \alpha)$	$DB' \leftarrow \emptyset$
$k_F \leftarrow\$ \mathcal{K}; k_{\mathrm{AE}} \leftarrow\$ \mathcal{K}$	For i in 1 to n':
For i in 1 to Q:	$\quad \ell_i \leftarrow\$ \{0,1\}^m$
$\quad w_i \leftarrow\$ \pi$	$\quad v_i \leftarrow\$ \mathcal{C}$
$\quad \ell_1, \ldots, \ell_B \leftarrow\$ \mathrm{Batch}(w_i)$	$\quad DB' \leftarrow DB' \cup \{(\ell_i, v_i)\}$
$\quad$ For j in 1 to B:	For i in 1 to q:
$\quad\quad \tau_B[j] \leftarrow (\ell_j, DB'[\ell_j])$	$\quad$ For j in 1 to B:
$\quad \tau[i] \leftarrow \tau_B$	$\quad\quad \ell \leftarrow\$ \mathrm{Labels}(DB')$
$b \leftarrow\$ \mathcal{A}_2(DB', \tau)$	$\quad\quad v \leftarrow DB'[\ell]$
Return b	$\quad\quad \tau_B[j] \leftarrow (\ell, v)$
	$\quad \tau[i] \leftarrow \tau_B$
	$b \leftarrow\$ \mathcal{A}_2(DB', \tau)$
	Return b

Figure 4.9: Security game for key value store schemes in the static distribution case. The threshold α is an implicit parameter of the left game. The procedures Init and Batch are as defined in Figure 4.2.

queries against EKV is defined as

$$\mathbf{Adv}^{\text{ror-cda}}_{\mathsf{EKV}}(\mathcal{A}) = |\Pr[\text{ROR-CDA1}^{\mathcal{A}}_{q} \Rightarrow 1] - \Pr[\text{ROR-CDA0}^{\mathcal{A}}_{q} \Rightarrow 1]|\,.$$

Next we state a key result, that PANCAKE achieves ROR-CDA security assuming estimation is sufficiently good. In particular this shows optimal security should $\hat{\pi} = \pi$.

Theorem 10. *Let $Q \geq 0$ and $Q = Q \cdot B$. Let $\pi, \hat{\pi}$ be distributions. For any Q-query* ROR-CDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ such that*

$$\mathbf{Adv}^{\textit{ror-cda}}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{\textit{prf}}_{F}(\mathcal{B}) + \mathbf{Adv}^{\textit{ror}}_{E}(\mathcal{C}) + \mathbf{Adv}^{\textit{dist}}_{Q,\pi,\hat{\pi}}(\mathcal{D})$$

where F and E are the PRF and AE scheme used by PANCAKE*. Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ each use Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Proof. We prove Theorem 10 using a series of standard cryptographic game transitions and reductions. We start with the game ROR-CDA1, replacing Init and Batch with the algorithms used in PANCAKE (see Figure 4.2). Game G_1 is the same as ROR-CDA1 except we replace the PRF F with a truly random function. The difference between the success of adversary $\mathcal{A}$ in these two games can be upper bounded by the advantage of a PRF adversary $\mathcal{B}$:

$$\left|\Pr\left[\,\text{ROR-CDA1}_q^{\mathcal{A}} \Rightarrow 1\,\right] - \Pr\left[\,G_1 \Rightarrow 1\,\right]\right| \leq \mathbf{Adv}_F^{\text{prf}}(\mathcal{B})\ .$$

We then move to game G_2, which is the same as G_1 except we replace the authenticated encryption function E with a random function outputting strings in the ciphertext space. The difference between the success rate of $\mathcal{A}$ in G_2 and G_1 can be upper bounded by a real-or-random adversary $\mathcal{C}$ against the encryption scheme E:

$$|\Pr\left[\,G_1 \Rightarrow 1\,\right] - \Pr\left[\,G_2 \Rightarrow 1\,\right]| \leq \mathbf{Adv}_E^{\text{ror}}(\mathcal{C})\ .$$

Finally we let G_3 be the same as G_2 except that we replace $\hat{\pi}$ with π everywhere. A straightforward reduction gives that

$$\left|\Pr\left[\,\text{ROR-CDA1}_q^{\mathcal{A}} \Rightarrow 1\,\right] - \Pr\left[\,G_1 \Rightarrow 1\,\right]\right| \leq \mathbf{Adv}_{Q,\pi,\hat{\pi}}^{\text{dist}}(\mathcal{D})\ .$$

We now come to the core of the argument, that G_3 is identically distributed to ROR-CDA0. In G_3 all labels and values are random strings. Further, each of the accesses is a uniformly random choice from all possible labels.

To see this, observe that each access in a batch is independent and sampled from π with probability δ or π_f with probability $1-\delta$. By construction of the scheme as described in Equation 4.1, the probability of any replica being accessed is the same. Let $\hat{\tau}$ be a random variable representing the output of Batch on input a sample from π, and $\hat{\tau}_i$ be the ith access in the output. Then for all i and any

replica (k, j)

$$\begin{aligned}\Pr[\hat{\tau}_i = (k,j)] &= \Pr[\hat{\tau}_i = (k,j) \mid q_{\text{type}} = 0] \cdot (1-\delta) \\ &\quad + \ \Pr[\hat{\tau}_i = (k,j) \mid q_{\text{type}} = 1] \cdot \delta \\ &= \frac{\alpha - \frac{\pi(k)}{R(k)}}{n'\alpha - 1} \cdot \frac{n'\alpha - 1}{n'\alpha} + \frac{\pi(k)}{R(k)} \cdot \frac{1}{n'\alpha} = \frac{1}{n'} \ .\end{aligned}$$

The theorem follows by the independence of the $\hat{\tau}_i$, and combining terms. □

Security analysis for dynamic distributions. Next we analyze security for dynamic distributions. First we must extend the formalization of frequency-smoothing KV schemes from above to account for the extended semantics. Specifically the batch algorithm Batch can now take an optional additional input $\hat{\pi}'$, representing an updated distribution estimate. This signals to Batch that it must adjust to the new distribution. We denote running Batch as before when given this additional input by $\ell_1, \ldots, \ell_B \leftarrow\$ \text{Batch}(\hat{\pi}', k)$. Recall that Batch is stateful and so when it gets a new estimate $\hat{\pi}'$, it also has access to the old estimate $\hat{\pi}$ as well as other state values. For PANCAKE, the Batch algorithm would use this information to run MakeReplicaLists and to setup its replica bookkeeping (refer to Section 4.12 for details).

We now introduce a security definition ROR-CDDA or, real-or-random indistinguishability under chosen-dynamic-distribution attack. Game ROR-CDDA1 is now parameterized by the query number and four distributions $\pi, \hat{\pi}, \pi', \hat{\pi}'$. The adversary runs and can pick both a plaintext KV store and a change point $c \in [0, Q]$. After the first c queries, keys switch from being sampled according to π to being sampled according to π' and Batch is run with the additional input $\hat{\pi}'$. The ROR-CDDA0 is the same as ROR-CDA0 except for the syntactic change that $\mathcal{A}_1$ outputs the additional value c. Otherwise the distribution over DB' (a KV store

$\underline{\text{ROR-CDDA1}^{\mathcal{A}}_{Q,\pi,\pi',\hat{\pi},\hat{\pi}'}:}$

$(DB, c) \leftarrow\$ \mathcal{A}_1$
$(DB', \pi_f, \delta) \leftarrow\$ \text{Init}(\hat{\pi}, DB, \alpha)$
For i in 1 to c:
  $w_i \leftarrow\$ \pi$
  $\ell_1, \ldots, \ell_B \leftarrow\$ \text{Batch}(w_i)$
  For j in 1 to B:
    $\tau_B[j] \leftarrow (\ell_j, DB'[\ell_j])$
  $\tau[i] \leftarrow \tau_B$
For i in c to Q:
  $w_i \leftarrow\$ \pi'$
  $\ell_1, \ldots, \ell_B \leftarrow\$ \text{Batch}(\hat{\pi}', w_i)$
  For j in 1 to B:
    $\tau_B[j] \leftarrow (\ell_j, DB'[\ell_j])$
  $\tau[i] \leftarrow \tau_B$
$b \leftarrow\$ \mathcal{A}_2(DB', \tau)$
Return b

$\underline{\text{ROR-CDDA0}^{\mathcal{A}}_{Q}:}$

$(DB, c) \leftarrow\$ \mathcal{A}_1$
$n \leftarrow \text{supp}(\pi)$
$DB' \leftarrow \emptyset$; $DB'' \leftarrow \emptyset$
For i in 1 to n':
  $\ell_i \leftarrow\$ \{0,1\}^m$
  $v_i \leftarrow\$ \mathcal{C}$
  $DB' \leftarrow DB' \cup \{(\ell_i, v_i)\}$
For i in 1 to Q:
  For j in 1 to B:
    $\tau_B[j] \leftarrow\$ \text{Labels}(DB')$
  $\tau[i] \leftarrow \tau_B$
$b \leftarrow\$ \mathcal{A}_2(DB', \tau)$
Return b

Figure 4.10: Security games for dynamic key value store schemes. The threshold α is an implicit parameter of the left game.

of uniform bit strings) and τ (qB uniform requests) are the same as before.

The ROR-CDDA advantage of an adversary $\mathcal{A}$ against a scheme EKV is defined as

$$\mathbf{Adv}^{\text{ror-cdda}}_{\mathsf{EKV}}(\mathcal{A}) = \Big| \Pr\left[\text{ROR-CDDA1}^{\mathcal{A}}_{Q,\pi,\pi',\hat{\pi},\hat{\pi}'} \Rightarrow 1\right] - \Pr\left[\text{ROR-CDDA0}^{\mathcal{A}}_{Q} \Rightarrow 1\right] \Big| .$$

One could easily extend this definition to handle a longer sequence of changes: our results extend to this setting as well. We note that the definition also implies that the transcript of queries is indistinguishable from one that is independent of the change point, meaning this information is hidden by schemes that meet the definition.

We now prove the following theorem about the dynamic version of PANCAKE.

Theorem 11. *Let $Q \geq 0$ and $Q = Q \cdot B$. Let $\pi, \pi', \hat{\pi}, \hat{\pi}'$ be distributions. For any Q-query* ROR-CDDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries*

$\mathcal{B}, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$ *such that*

$$\mathbf{Adv}^{ror\text{-}cdda}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{prf}_{F}(\mathcal{B}) + \mathbf{Adv}^{ror}_{E}(\mathcal{C}) + \mathbf{Adv}^{dist}_{Q,\pi,\hat{\pi}}(\mathcal{D}_1) + \mathbf{Adv}^{dist}_{Q,\pi',\hat{\pi}'}(\mathcal{D}_2)$$

where F and E are the PRF and AE scheme used by PANCAKE. *Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$ each use at most Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Proof. Similar to the proof of Theorem 8 above, we use game hops to replace the PRF labels and AE ciphertexts with random strings, and upper-bound the difference in advantage via the PRF and AE adversaries $\mathcal{B}$ and $\mathcal{C}$. We also replace $\hat{\pi}$ with π and $\hat{\pi}'$ with π' in game hops whose difference is upper bounded by the appropriate reductions to distinguishers $\mathcal{D}_1$ and $\mathcal{D}_2$. This brings us to a game G where labels and ciphertexts are random strings, but batches are generated using Batch with π on input samples from π (before the change) or with π' on input samples from π' (after the change).

We claim that the distribution of accesses in game G is uniformly random, the same as in ROR-CDDA0. Because the keys accessed in a batch are independent, it suffices to show a single access of a batch is uniform. Let $\hat{\tau}_i$ be the ith access of a batch generated by a query sampled from π'. Let (k, j) be any replica. Then to compute $\Pr[\hat{\tau}_i = (k, j)]$, there are a few cases. First recall that,

$$\Pr[\hat{\tau}_i = (k, j)] = \Pr[\hat{\tau}_i = (k, j) \mid q_{\text{type}} = 0] \Pr[q_{\text{type}} = 0] + \Pr[\hat{\tau}_i = (k, j) \mid q_{\text{type}} = 1] \Pr[q_{\text{type}} = 1]$$

where $q_{\text{type}} = 0$ means a fake query and $q_{\text{type}} = 1$ means a real query. There are three possible cases:

Case 1: k gained replicas and j is one of its existing replicas. Then the RHS above is equal to:

$$\frac{\alpha' - \pi'(k)/R(k)}{2n\alpha' - 1}\left(1 - \frac{1}{2n\alpha'}\right) + \frac{\pi'(k)}{R(k)} \cdot \frac{1}{2n\alpha'} = \frac{1}{2n}$$

Case 2: $(k, j) \in \mathsf{G}$. Then,

$$\Pr[\hat{\tau}_i = (k, j)] = \frac{\alpha'}{2n\alpha' - 1} \cdot \frac{2n\alpha' - 1}{2n\alpha'} = \frac{\alpha'}{2n\alpha'} = \frac{1}{2n} \ .$$

Case 3: (k, j) is either in L or is any other replica. In this case, $\Pr[\hat{\tau}_i = (k, j)] = 1/2n$ follows from Eq. 4.1. □

4.9 Key Management

In deployment, the cryptographic keys used for computing PRF labels and encrypting/decrypting ciphertexts will be managed by PANCAKE. This allows the PANCAKE proxy to be transparent to clients: they can interact with PANCAKE as they would interact with a key-value store. We envision that PANCAKE will communicate with a hardware security module (HSM) to acquire keys on startup, instead of storing cryptographic keys itself.

Key rotation. Periodically changing cryptographic keys (usually called *key rotation*) is necessary in any system that uses cryptography because cryptographic keys can only be used a limited number of times. It also provides a kind of post-compromise security, ensuring that a one-time key compromise does not give the attacker perpetual access to plaintexts.

In PANCAKE, frequent key rotation is useful for another reason: it allows the proxy to delete key-dependent metadata more frequently. One such source of key-dependent metadata is the label map in the replica-swapping procedure for dynamic distributions described in §4.5. Timely changes to the keys used for the PRF labels ensures the label map can be deleted soon after replica swapping has completed.

There are two ways to perform key rotation in PANCAKE. The first is to download the entire database, decrypt everything, and re-run PANCAKE's Init procedure with new keys. This is the simplest method but requires queries to cease for a period of time while re-encryption takes place. In settings where periodic scheduled downtime is used for maintenance and patching systems, this method can be used without interrupting client workflows.

The other way to perform key rotation is to fetch a small number of additional key-value pairs with each query and re-encrypt and re-upload just those pairs. Key-value pairs are fetched on a fixed schedule (perhaps in order of PRF label) to prevent information leakage. This can be viewed as an "incremental" version of the bulk re-keying method described above. The major benefit of incremental re-keying is it can be done online. It does have some drawbacks: during re-keying, PANCAKE must store metadata to track which keys have already been re-keyed so that subsequent queries fetch the right label. This metadata incurs some space overhead and makes other operations (like swapping replicas to handle distribution changes) more complex. Nevertheless, in settings where downtime of any kind is unacceptable this may be the only way to rotate keys.

4.10 Active Attacks on pancake

Throughout this paper we have assumed a persistent passive attacker that monitors honest client queries but does not get to make any queries of its choosing. This setting is standard in the literature: only two prior works [146, 24] study these active (chosen-query) attacks, but in a different setting than ours.

The goal of an active attack in our setting is recovering other clients' queries. Such attacks could arise against encrypted KV stores if an attacker can convince an honest client to issue a query on its behalf, and sensitive information is ingested from public sources and inserted into the data store. One example is given by [59]: they describe how public web forms can be used to mount an active attack on encrypted customer data. Another example is online appointment scheduling for medical practices. If a practice encrypts appointment records but allows new patients to make appointments online, an attacker can mount an active attack by making fake appointments.

Such an attack would work by querying some target keys and learning the PRF labels that correspond to the keys. Then, when an honest client makes a query, if its label is one the adversary previously queried it can learn the key. Against the "canonical" encrypted KV store where each key has only one replica and no fake queries are used, this attack works very well: the attacker learns the PRF label of its chosen query as soon as it is issued.

Chosen-query attacks against pancake. With an unlimited number of chosen queries the attack will still work against PANCAKE, but the frequency-smoothing techniques used in PANCAKE increase resistance to such an attack.

For example, if δ is the probability of a real query and B is the batch size, then with probability $(1-\delta)^B$ the batch that PANCAKE issues for the attacker's query generates will not even contain its query — only fake ones. Even if the query is in the batch, the attacker must distinguish its label from the B others. To combat this, each chosen query must be issued several times for the attacker to learn the correct label.

The number of chosen queries needed to gain confidence in the label depends on the number of replicas. If the key has only a single replica, two queries should suffice to learn the label, because with probability at least $1-(1-\delta)^{2B}$ the PRF label will occur twice in the two queries. Further, with good probability the correct label will be the only one that occurs twice. If the queried key has multiple replicas, though, its chosen queries will access different replicas each time. By a birthday bound argument, if the key has r replicas the attacker must perform around $\sqrt{r}$ chosen queries to see a label of any replica twice, and at least $2r \log r$ queries to see all replica labels twice.

In conclusion, PANCAKE is no worse than other encrypted KV stores in the face of chosen-query attacks, and increases resistance to them somewhat. We leave studying efficient defenses against chosen-query attacks to future work.

4.11 Correlated Queries

So far, our model assumed that client queries are i.i.d.: such a model is suitable for many real-world settings. However, there are also settings where it may be unsuitable because queries are *correlated* — the distribution of future queries depends on past queries and their results. One example of query correlations arises when

accesses are made to a document store with a keyword index. The index identifies document IDs that are associated with specific keywords. A lookup for a keyword in the index is typically followed by accesses to one or more documents associated with it, i.e., keyword accesses are *correlated* with document accesses.

Unfortunately, when queries are correlated, Theorem 8 no longer applies because its proof relies on independence of queries. In fact, we will show below in Appendix 4.11.1 that security in a variant of ROR-CDA that allows arbitrary correlations is equivalent to ORAM security, and must therefore suffer from the same lower bounds on performance overheads.

Fortunately, this proof relies on the the correlated distributions being very specific: one where all subsequent queries are completely determined by the first — which we argue rarely occurs in practice. For more realistic workloads, we show empirically in Appendix 4.11.2 that PANCAKE already obfuscates correlations enough to thwart frequency-analysis attacks that exploit such correlations. We leave the formal analysis of PANCAKE security in such settings to future work.

4.11.1 Hiding Arbitrary Correlations ≈ ORAM

In this section, we show that a scheme ψ that hides arbitrary query correlations in a known-distribution setting provides the same security as offline ORAM scheme χ, and therefore inherits its performance overheads.

Security definitions for ψ We modify our original definitions for PANCAKE security in the static case (§4.8) to incorporate the notion of correlations between accesses to the key-value store; we capture this in a security definition called real-

ROR-CMDA1$^{\mathcal{A}}_{Q,\psi}$:	ROR-CMDA0$^{\mathcal{A}}_{Q,\psi}$:

$(DB, \pi_{W_1 \ldots W_Q}, \mathsf{st}_{\mathcal{A}}) \leftarrow\$ \mathcal{A}_1(q)$
$DB', \mathsf{st}_{\psi} \leftarrow \psi.\mathrm{Init}(\pi_{W_1 \ldots W_Q}, DB)$
$k_F \leftarrow\$ \mathcal{K}; k_{\mathrm{AE}} \leftarrow\$ \mathcal{K}$
$(w_1, \ldots, w_q) \leftarrow\$ \pi_{W_1 \ldots W_Q}$
For i in 1 to Q:
  $\ell_1, \ldots, \ell_B \leftarrow \psi.\mathrm{Batch}(w_i)$
  For j in 1 to B:
  $\tau_B[j] \leftarrow (\ell_j, DB'[\ell_j])$
  $\tau[i] \leftarrow \tau_B$
$b \leftarrow\$ \mathcal{A}_2(\mathsf{st}_{\mathcal{A}}, DB', \tau)$
Return b
	$(DB, \pi_{W_1 \ldots W_Q}, \mathsf{st}_{\mathcal{A}}) \leftarrow\$ \mathcal{A}_1(q)$
	$DB' \leftarrow \emptyset$
	For i in 1 to $2n$:
	  $\ell_i \leftarrow\$ \{0,1\}^m$
	  $v_i \leftarrow\$ \mathcal{C}$
	  $DB' \leftarrow DB' \cup \{(\ell_i, v_i)\}$
	For i in 1 to q:
	  For j in 1 to B:
	  $\ell \leftarrow\$ \mathrm{Labels}(DB')$
	  $v \leftarrow DB'[\ell]$
	  $\tau_B[j] \leftarrow (\ell, v)$
	  $\tau[i] \leftarrow \tau_B$
	$b \leftarrow\$ \mathcal{A}_2(\mathsf{st}_{\mathcal{A}}, DB', \tau)$
	Return b

Figure 4.11: Security game for key value store schemes in the static distribution case with arbitrary correlations of length q.

versus-random indistinguishability under chosen marginal distribution attack or ROR-CMDA. The adversary $\mathcal{A}$'s success in attacking a scheme ψ is measured by its ability to distinguish between the games ROR-CMDA1 and ROR-CMDA0, as defined in Figure 4.11. In ROR-CMDA1, the adversary gets the output from ψ, and in ROR-CMDA0, the adversary sees a randomly generated encrypted database and queries chosen uniformly at random. The advantage of $\mathcal{A}$ with q queries is:

$$\begin{aligned}\mathbf{Adv}^{\text{ror-cmda}}(\mathcal{A}) = &|\Pr[\text{ROR-CMDA1}^{\mathcal{A}}_{q,\psi} \Rightarrow 1] - \\ &\Pr[\text{ROR-CMDA0}^{\mathcal{A}}_{q,\psi} \Rightarrow 1]|\,.\end{aligned}$$

Security definition for offline ORAM Figure 4.12 provides the game for an offline ORAM scheme χ. The adversary $\mathcal{B}$ generates a set of memory blocks, along with two transcripts T_0 and T_1. The ORAM R is initialized using the blocks and the transcript T_b, where bit b is a parameter of the game. The scheme χ.Exec is run on T_b and the adversary $\mathcal{A}$ attempts to guesses the bit b using the resulting view τ, along with the random coins $\mathsf{st}_{\mathcal{B}}$ used to generate the transcripts. The adversary wins if it can guess the bit accurately. We refer to this security definition as

$\underline{\text{IND-CTA}^{\mathcal{B}}_{b,q,\chi}}$:

blocks $\leftarrow \mathcal{B}_1$
$\mathsf{st}_{\mathcal{B}}, T_0, T_1 \leftarrow \mathcal{B}_2(q)$
$R \leftarrow \chi.\text{Init}(\text{blocks}, T_b)$
$\tau \leftarrow \chi.\text{Exec}(T_b, R)$
$b' \leftarrow \mathcal{B}_3(\mathsf{st}_{\mathcal{B}}, R, \tau)$
Return b'

Figure 4.12: Security game for offline ORAM.

$\underline{\text{Make-}\pi(T)}$:

$$\pi_{W_1,\ldots,W_{|T|}} = \begin{cases} 1, \text{ if } w_1, \ldots, w_{|T|} = T \\ 0, \text{ otherwise} \end{cases}$$

return $\pi_{W_1,\ldots,W_{|T|}}$

$\underline{\chi.\text{Init}(\text{blocks}, T)}$:

$\pi_{W_1,\ldots,W_{|T|}} \leftarrow \text{Make-}\pi(T)$
$DB \leftarrow \emptyset$
For i in 1 to $|\text{blocks}|$:
  $DB \leftarrow DB \cup \{(i, \text{blocks}[i])\}$
Return $\psi.\text{Init}(\pi_{W_1,\ldots,W_{|T|}}, DB)$

$\underline{\chi.\text{Exec}(T, R)}$:

$\pi_{W_1,\ldots,W_{|T|}} \leftarrow \text{Make-}\pi(T)$
$(DB', \mathsf{st}_{\psi} \leftarrow R$
$w_1, \ldots, w_{|T|} \leftarrow T$
For 1 in 1 to $|T|$:
  $\ell_1, \ldots, \ell_B \leftarrow\$ \psi.\text{Batch}(w_i)$
  For j in 1 to B:
    $\tau_B[j] \leftarrow (\ell_j, DB'[\ell_j])$
  $\tau[i] \leftarrow \tau_B$
Return τ

Figure 4.13: Construction of scheme χ using ψ.

indistinguishability under chosen transcript attack or IND-CTA. The advantage of $\mathcal{B}$ with q queries is:

$$\mathbf{Adv}^{\text{ind-cta}}(\mathcal{B}) = \left|\Pr[\text{IND-CTA}^{\mathcal{B}}_{1,q,\chi} \Rightarrow 1] - \Pr[\text{IND-CTA}^{\mathcal{B}}_{0,q,\chi} \Rightarrow 1]\right|.$$

Theorem 12. *Let ψ be a* ROR-CMDA*-secure scheme. Then* $\forall$IND-CTA-

$\underline{\mathcal{A}_1^b(q)}$:
blocks $\leftarrow \mathcal{B}_1$
$\mathsf{st}_{\mathcal{B}}, T_0, T_1 \leftarrow \mathcal{B}_2(q)$
$\pi_{W_1,\ldots,W_q} \leftarrow$ Make-$\pi(T_b)$
$DB \leftarrow \emptyset$
For i in 1 to |blocks|:
  $DB \leftarrow DB \cup \{(i, \text{blocks}[i])\}$
return $(DB, \pi_{W_1,\ldots,W_q}, \mathsf{st}_{\mathcal{B}})$

$\underline{\mathcal{A}_2^b(\mathsf{st}_A, DB', \tau)}$:
$DB'_s \leftarrow \text{Sort}(DB')$
$i \leftarrow 0$
For (l, v) in DB'_s:
  $R[i] \leftarrow v$
  $i \leftarrow i + 1$
return $\mathcal{B}_3(\mathsf{st}_A, \text{blocks}, \tau)$

$\underline{\mathcal{G}^{\mathcal{B}}_{sim,q}}$:
blocks $\leftarrow \mathcal{B}_1$
$\mathsf{st}_{\mathcal{B}}, T_0, T_1 \leftarrow \mathcal{B}_2(q)$
For i in 1 to N:
  $R[i] \leftarrow_{\$} \{0,1\}^k$
For i in 0 to q:
  $i \leftarrow_{\$} \{1, \ldots, |\text{blocks}|\}^k$
$b' \leftarrow \mathcal{B}_3(\mathsf{st}_{\mathcal{B}}, R, \tau)$
Return b'

Figure 4.14: (Left) ROR-CMDA-adversary $\mathcal{A}^b$ using the IND-CTA-adversary $\mathcal{B}$. (Right) Security game $\mathcal{G}^{\mathcal{B}}_{sim,q}$.

adversaries $\mathcal{B}$, $\exists$ROR-CMDA-*adversary* $\mathcal{A}$, *such that,* $\mathbf{Adv}^{ind\text{-}cta}_{\chi}(\mathcal{B}) \leq 2 \cdot \mathbf{Adv}^{ror\text{-}cmda}_{\psi}(\mathcal{A})$, *where* χ *is an* IND-CTA-*secure scheme constructed from* ψ.

Proof. We will show that it is always possible to construct a scheme χ given an ROR-CMDA-secure scheme ψ, and an ROR-CMDA-adversary $\mathcal{A}$ to scheme ψ given an IND-CTA-adversary $\mathcal{B}$ to χ, such that:

$$\mathbf{Adv}^{\text{ind-cta}}_{\chi}(\mathcal{B}) \leq 2 \cdot \mathbf{Adv}^{\text{ror-cmda}}_{\psi}(\mathcal{A})$$

Figure 4.13 (Left) shows the construction of scheme χ using ψ. We define ROR-CMDA-adversaries $\mathcal{A}^0$ and $\mathcal{A}^1$ as shown in Figure 4.14 (Left) and $\mathcal{G}^{\mathcal{B}}_{sim,q}$ as shown in Figure 4.14 (Right).

We have,

$$\Pr[\text{ROR-CMDA1}^{\mathcal{A}^0}_{q,\psi} \Rightarrow 1] = \Pr[\text{IND-CTA}^{\mathcal{B}}_{0,q,\chi} \Rightarrow 1]$$

$$\Pr[\text{ROR-CMDA0}^{\mathcal{A}^0}_{q,\psi} \Rightarrow 1] = \Pr[\mathcal{G}^{\mathcal{B}}_{sim,q} \Rightarrow 1]$$

And therefore,

$$|\Pr[\text{IND-CTA}^{\mathcal{B}}_{0,q,\chi} \Rightarrow 1] - \Pr[\mathcal{G}^{\mathcal{B}}_{sim,q} \Rightarrow 1]| = \mathbf{Adv}^{\text{ror-cmda}}_{\psi}(\mathcal{A}^0)$$

Similarly,

$$|\Pr[\mathcal{G}^{\mathcal{B}}_{sim,q} \Rightarrow 1] - \Pr[\text{IND-CTA}^{\mathcal{B}}_{1,q,\chi} \Rightarrow 1]| = \mathbf{Adv}^{\text{ror-cmda}}_{\psi}(\mathcal{A}^1)$$

Therefore, by triangle inequality,

$$\begin{aligned}
&\mathbf{Adv}^{\text{ind-cta}}_{\chi}(\mathcal{B}) \\
&\quad = |\Pr[\text{IND-CTA}^{\mathcal{B}}_{1,q,\chi} \Rightarrow 1] - \Pr[\text{IND-CTA}^{\mathcal{B}}_{0,q,\chi} \Rightarrow 1]| \\
&\quad \leq |\Pr[\text{IND-CTA}^{\mathcal{B}}_{0,q,\chi} \Rightarrow 1] - \Pr[\mathcal{G}^{\mathcal{B}}_{sim,q} \Rightarrow 1]| \\
&\qquad\quad + |\Pr[\mathcal{G}^{\mathcal{B}}_{sim,q} \Rightarrow 1] - \Pr[\text{IND-CTA}^{\mathcal{B}}_{1,q,\chi} \Rightarrow 1]| \\
&\quad = \mathbf{Adv}^{\text{ror-cmda}}_{\psi}(\mathcal{A}^0) + \mathbf{Adv}^{\text{ror-cmda}}_{\psi}(\mathcal{A}^1) \\
&\quad \leq 2 \cdot \mathbf{Adv}^{\text{ror-cmda}}_{\psi}(\mathcal{A})
\end{aligned}$$

□

We note that while we have adapted the security definitions for the static case (i.e., ROR-CDA), it is straightforward to adapt definitions for the dynamic case (i.e., ROR-CDDA) to reach the same conclusion.

4.11.2 Empirical Analysis of pancake Security for Correlated Accesses

The proof outlined above in Appendix 4.11.1 assumes very artificial distributions; in this section, we empirically evaluate the extent to which PANCAKE's frequency smoothing approach obfuscates correlations for more realistic distributions.

Setup We consider a document store which stores two different mappings: (1) a mapping from documentIDs to documents, and (2) a mapping from specific keywords to documentIDs associated with them (corresponding to a "keyword index"). For simplicity, we consider an access pattern to the document store where a user first accesses the keyword index to lookup the set of documentIDs for a specific keyword, and subsequently accesses the document for to a randomly selected documentID in the set. If k_i and d_i correspond to keyword and document accesses respectively, then the access transcript would appear as $\{k_1, d_1, k_2, d_2, ...\}$.

This representative access pattern for document store highlights correlated accesses: each access k_i is correlated to access d_i; even if accesses across individual k_i's and d_i's were indistinguishable, an adversary could apply frequency analysis to the joint distribution over (k_i, d_i) pairs. To this end, we empirically evaluate the extent to which PANCAKE can smooth the joint distribution; we consider the Enron dataset [74] with $\sim$ 500 thousand emails (documents) and $\sim$ 32 thousand keywords. The accesses across the keywords themselves are Zipfian (same as the YCSB workload). However, the accesses to documents are determined by the keywords that are accessed and the pre-defined associations between keywords and documents. Both the keyword index as well as the documentID-document mappings are assumed to be stored in the same key-value store. PANCAKE employs frequency smoothing at the granularity of KV pairs, but does not employ any additional techniques to smooth the correlations any further.

Attack Strategy We assume that the adversary can only exploit frequency-based leakage — we ignore all other leakage profiles (e.g., length, volume, etc.). For an encrypted key-value store that does not employ any frequency smoothing, the adversary can exploit both regular frequency analysis, as well as frequency

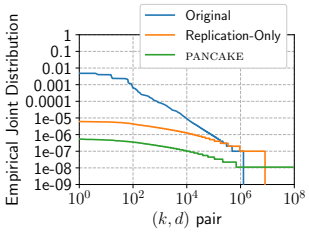
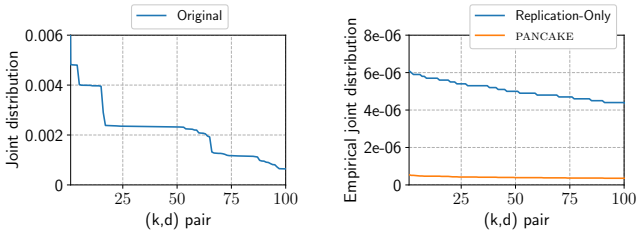


Figure 4.15: Original joint distribution and empirical joint distribution computed over access transcript generated by replication-only approach and PANCAKE.



(a) Original Joint Distribution (b) Empirical Joint distribution

Figure 4.16: (a) Original joint distribution over the 100 most popular keyword-document pairs. (b) Empirical joint distribution computed over access transcript generated by replication-only approach and PANCAKE, for the 100 most popular keyword-document pairs.

analysis on the joint distribution over (k_i, d_i) pairs; for PANCAKE, the adversary can only exploit the latter. Our attack analyzes consecutive batches only, exploiting the observation that alternating batches contain a real keyword access and a real document access respectively[2]. For each pair of batches of size B, the attacker considers all the B^2 pairs of accesses (since they are indistinguishable) and computes a histogram over them. The adversary is assumed to know the original joint distribution over (k, d) pairs, and compares it to the empirically computed joint distribution using the access transcripts, akin to regular frequency-analysis attacks.

Results Figure 4.15 shows the joint distribution empirically computed over distinct (k, d) pairs from a 10 million query transcript generated by PANCAKE for

[2]With small probability, a PANCAKE batch may contain no real accesses, but we ignore this possibility in the attack.

the Enron dataset, and compares with the original distribution (note that both x and y-axes are in log-scale). The key observation from the figure is that PANCAKE is able to flatten the joint distribution considerably through replication and fake accesses. We also measure the empirical joint distribution resulting from an approach that only uses PANCAKE's replication approach (without any fake queries), highlighting the contribution from both of the schemes to smooth the joint distribution. Figures 4.16(a) and 4.16(b) respectively show the original joint distribution, and empirical joint distribution computed for transcripts from the replication-only approach and the PANCAKE approach, for the 100 most frequent pairs only.

We note that a majority of the smoothing can be attributed to replication; the intuitive reason behind this is that for every (k, d) pair, $\pi(k, d) < \pi(k)$ and $\pi(k, d) < \pi(d)$; however, accesses for (k, d) are spread across $R(k) \cdot R(d)$ replica pairs. As a consequence, more popular (k, d) pairs get spread across significantly more replica-pairs. To analyze the effect of fake accesses, we note that these are drawn from an iid distribution in PANCAKE. Consequently, while they contribute to the less popular (k, d) pairs, they also generate accesses to spurious (k, k), (d, d) and (d, k) pairs, that were non-existent in the original joint distribution. Since these spurious pairs are indistinguishable to an adversary (due to PRF security), it makes frequency analysis even harder. In fact, our analysis of the original and PANCAKE transcripts revealed that *none* of the 100 most-frequent (k, d) pairs in the original joint distribution were among the 100 most-frequent replica pairs accessed with PANCAKE— we attribute this to the combination of the two phenomena described above.

Discussion While the above does not provide a formal analysis of the security afforded by PANCAKE for correlated accesses, it does provide strong evidence that

```
MakeReplicaLists(π, π', α):
If supp(π) ≠ supp(π') then Return ⊥
n ← |supp(π)| ; n' ← 0 ; n'' ← 0
For k ∈ supp(π):
   r ← ⌈π(k)/α⌉; r' ← ⌈π'(k)/α⌉
   If r > r':
      L ← L ∪ {(k, ℓ) | ℓ ∈ [r' + 1, ..., r]}
   Else If r' > r:
      G ← G ∪ {(k, ℓ) | ℓ ∈ [r + 1, ..., r']}
   n' ← n' + r ; n'' ← n'' + r'
n'_d ← 2n − n' ; n''_d ← 2n − n''
If n'_d > n''_d:
   L ← L ∪ {(D, ℓ) | ℓ ∈ [n''_d + 1, ..., n'_d]}
Else If n''_d > n'_d:
   G ← G ∪ {(D, ℓ) | ℓ ∈ [n'_d + 1, ..., n''_d]}
Return L, G
```

Figure 4.17: Pseudocode for preparing replica swapping metadata.

PANCAKE significantly obfuscates the access correlations, even without any modifications. We plan to explore a formal analysis for the same as future work. It may also be possible to modify PANCAKE to use a model of the correlations in its frequency smoothing instead of assuming queries are i.i.d.. We leave this as another interesting open problem for future work.

4.12 Details for Dynamic Distributions

In this appendix we describe in greater detail the modified Batch algorithm used during a distribution change, as well as provide a formal security analysis. Aspects of the algorithm are described in §4.5; here we include the full description for completeness. We assume the change has been detected and the MakeReplicaLists procedure (see Figure 4.17) has been run to produce lists L and G of replicas that can be removed and that need to be created, respectively.

The modified Batch algorithm maintains the following additional bookkeeping data structures:

- Updates tracks which replicas in G have not yet been created by overwriting a replica in L. It maps a key to a set of replicas that need to be updated and the new value, and stores information needed for any actual write queries that have not finished (see §4.4).
- LabelMap records which replicas have been transferred and therefore do not have a label corresponding to their actual key and replica number. Any replica (w, j) whose label ℓ is *not* (w, j) has an entry in this mapping table. Since each of the n keys always has at least 1 replica, no matter how the distribution changes (assuming, as we do, that the support stays the same), there can be at most n entries in this mapping table.
- UninitReps tracks which keys gained replicas after a distribution change and whose values still need to be cached to create its new replicas. For each such key it records the replicas not yet initialized.
- ValueCache is a cache of values of keys in G, so that we can write them back to initialize new replicas. Our implementation combines the ValueCache and UpdateCache into one; we empirically evaluated the storage required for this cache in §4.6.2 and §4.6.3.

Accesses are done in much the same way as in the static case, except on each access the various bookkeeping data structures are consulted to ensure two things: the correct label is used to access replicas (using LabelMap) and the correct value is written back (using UninitReps and Updates).

In particular, on generating each access in a batch the following steps are followed: first, flip a coin to determine whether the access is real or fake. If it is fake, draw a sample from the temporary fake access distribution $\tilde{\pi}'_f$. If it is real, either pull a client query from the queue or sample a key to access from $\hat{\pi}'$. If the

client query is a write for a key in UninitReps, remove it from UninitReps and add each of its replicas in G to Updates along with the "paired" replica in L that its value will be replacing.

If the access is real, a replica must be sampled for it. If the key k is in G, sample a replica is from $[1, \ldots, R_{\hat{\pi}}(k)]$, else sample it from $[1, \ldots, R_{\hat{\pi}'}(k)]$. If the replica is in G and in UninitReps, remove it from UninitReps.

After receiving and decrypting the values, Batch uses its metadata to write back the correct value for each key. If the replica (k, j) is in L and a replica (k', ℓ) in G has a cached value in ValueCache and is in UninitReps, add an entry to LabelMap indicating that (k', j') is getting the label of (k, j). This is the actual swapping step, where a new replica is created by overwriting the value of an old one. Else, perform the writeback as usual, except if the key k has gained replicas and its value is not in ValueCache, add it.

Eventually the UninitReps data structure is empty, and replica swapping is done. We can delete all cached values and temporary metadata except LabelMap, which is needed to find the correct label for created replicas.

CHAPTER 5

CONCLUSION

5.1 Conclusion

In this thesis we have studied nearly every aspect of encrypted databases, from theoretical to practical. My main conclusions are as follows.

Encrypted databases can meaningfully improve security. My work has demonstrated many attacks on encrypted databases, but I remain bullish. I do not believe any of my attack results represent insurmountable flaws in the entire research program.

More work is needed to understand security in real settings. I think my work more points to the need for more study, both on attacks and constructions. In ongoing work, I'm trying to instantiate a real snapshot threat model by building *forgetful systems*, addressing the issues pointed out in Chapter 2. I also plan to study (simulated) attacks *in situ* by doing large-scale measurements of database deployments, to understand what leakage would be available to an attack for a real workload.

BIBLIOGRAPHY

[1] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling Queries on Compressed Data. In *NSDI*, 2015.

[2] InnoDB adaptive hash index. `https://dev.mysql.com/doc/refman/5.7/en/innodb-adaptive-hash.html`.

[3] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *EuroSys*, 2011.

[4] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with Cipherbase. In *CIDR*, 2013.

[5] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. In *EUROCRYPT*, 2020.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*, 2012.

[7] Baffle. `https://baffle.io`.

[8] Marco Balduzzi, Jonas Zaddach, Davide Balzarotti, Engin Kirda, and Sergio Loureiro. A security analysis of Amazon's elastic compute cloud service. In *SAC*, 2012.

[9] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, 2007.

[10] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *FOCS*, 1997.

[11] Gyora M. Benedek and Alon Itai. Learnability with respect to fixed distributions. *Theor. Comput. Sci.*, 1991.

[12] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. The tao of inference in privacy-protected databases. *PVLDB*, 2018.

[13] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *CCS*, 2015.

[14] The binary log. `http://dev.mysql.com/doc/refman/5.7/en/binary-log.html`.

[15] mysqlbinlog — utility for processing binary log files. `http://dev.mysql.com/doc/refman/5.7/en/mysqlbinlog.html`.

[16] Microsoft BitLocker. `https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview`.

[17] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred Warmuth. Classifying learnable geometric concepts with the Vapnik-Chervonenkis dimension. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 273–282, New York, NY, USA, 1986. ACM.

[18] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, 2009.

[19] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ITCS*, 2016.

[20] Matt Bromiley. MongoDB forensics. `http://tinyurl.com/zkexl86`, 2015.

[21] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *ATC*, 2013.

[22] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. AmazonIA: When elasticity snaps back. In *CCS*, 2011.

[23] US Census Bureau. US Census Bureau name statistics. `https://www.ssa.gov/OACT/babynames/`, 2016.

[24] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[25] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*. 2013.

[26] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client oram. In *NDSS*, 2019.

[27] Zhao Chang, Dong Xie, and Feifei Li. Oblivious ram: a dissection and experimental evaluation. *VLDB*, 2016.

[28] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT*, 2010.

[29] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, 2007.

[30] Ciphercloud. `http://www.ciphercloud.com`.

[31] Jacob Willem Cohen and Anthony Browne. *The single server queue*. 1982.

[32] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[33] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, 2018.

[34] Sanjoy Dasgupta and John Langford. A tutorial on active learning. `http://hunch.net/~active_learning/`.

[35] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[36] Deep Learning Meets Heterogeneous Computing. `https://bit.ly/3hCoPz8`.

[37] Dropbox, 2018. `https://www.dropbox.com`.

[38] Muhaimin Dzulfakar. Advanced MySQL exploitation. Black Hat Las Vegas, 2009.

[39] Andrzej Ehrenfeucht, David Haussler, Michael Kearns, and Leslie Valiant. A general lower bound on the number of examples needed for learning. *Inf. Comput.*, 82(3):247–261, September 1989.

[40] Elasticsearch. `http://www.elasticsearch.org`.

[41] InnoDB tablespace encryption. `https://dev.mysql.com/doc/refman/5.7/en/innodb-tablespace-encryption.html`.

[42] OSX FileVault. `https://support.apple.com/en-us/HT204837`.

[43] Peter Frühwirt, Marcus Huber, Martin Mulazzani, and Edgar R Weippl. InnoDB database forensics. In *AINA*, 2010.

[44] Peter Frühwirt, Peter Kieseberg, Sebastian Schrittwieser, Markus Huber, and Edgar Weippl. InnoDB database forensics: Reconstructing data manipulation queries from redo logs. In *ARES*, 2012.

[45] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K. Cunningham. SoK: Cryptographically protected database search. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 172–191, May 2017.

[46] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[47] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *HotOS*, 2005.

[48] Tingjian Ge and Stan Zdonik. Fast, secure encryption for indexing in a column-oriented DBMS. In *ICDE*, 2007.

[49] Oded Goldreich, Shaffi Goldwasser, and Silvio Micali. How to construct random functions. *JACM*, 1986.

[50] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[51] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[52] How Google Search works. https://bit.ly/3hGWt70.

[53] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. Technical report, 2020. https://github.com/pancake-security.

[54] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agrawal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *Usenix Security*, 2020.

[55] 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 *ACM CCS*, 2018.

[56] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE S& P*, 2019.

[57] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, 2016.

[58] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *HotOS*, 2017.

[59] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *IEEE S&P*, 2017.

[60] Bernardo Damele Assumpção Guimarães. Advanced SQL injection to operating system full control. *Black Hat Europe*, 2009.

[61] David Haussler and Emo Welzl. Epsilon-nets and simplex range queries. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 61–71, New York, NY, USA, 1986. ACM.

[62] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP≈RDMA: Cpu-efficient remote storage access with i10. In *NSDI*, 2020.

[63] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging

what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. In *VLDB*, 2011.

[64] MySQL information schema. `https://dev.mysql.com/doc/refman/5.7/en/information-schema.html`.

[65] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[66] Seny Kamara and Tarik Moataz. SQL on structurally-encrypted databases. Cryptology ePrint Archive, Report 2016/453, 2016.

[67] Michael Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. On the learnability of discrete distributions. In *STOC*, 1994.

[68] Michael J. Kearns, Umesh Virkumar Vazirani, and Umesh Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[69] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[70] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *NSDI*, 2016.

[71] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. Zipg: A memory-efficient graph store for interactive queries. In *SIGMOD*, 2017.

[72] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting change in data streams. In *VLDB*, 2004.

[73] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash $\approx$ local flash. *SIGARCH*, 2017.

[74] Bryan Klimt and Yiming Yang. The enron corpus: New dataset for email classification research. In *ECML*, 2004.

[75] Andrey Kolmogorov. Sulla determinazione empirica di una lgge di distribuzione. *Inst. Ital. Attuari, Giorn.*, 1933.

[76] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *IEEE S&P*, 2019.

[77] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[78] 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)*, pages 297–314, May 2018.

[79] Marie-Sarah Lacharité and Kenneth G. Paterson. A note on the optimality of frequency analysis vs. ℓ_p-optimization. Cryptology ePrint Archive, Report 2015/1158, 2015. `http://eprint.iacr.org/2015/1158`.

[80] Marie-Sarah Lacharité and Kenneth G. Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically-encrypted data. IACR ePrint, 2017. `https://eprint.iacr.org/2017/1068`.

[81] Kasper Green Larsen, Tal Malkin, Omri Weinstein, and Kevin Yeo. Lower bounds for oblivious near-neighbor search. *arXiv preprint arXiv:1904.04828*, 2019.

[82] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO*. Springer International Publishing, 2018.

[83] Kevin Lewi and David J Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *CCS*, 2016.

[84] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys*, 2014.

[85] Linux Unified Key Setup (LUKS). `https://gitlab.com/cryptsetup/cryptsetup/blob/master/README.md`.

[86] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. Modular order-preserving encryption, revisited. In *SIGMOD*, 2015.

[87] Carlo Meijer and Bernard Van Gastel. Self-encrypting deception: weaknesses in the encryption of solid state drives. In *IEEE S& P*, 2019.

[88] Microsoft. DBCC for SQL server. `https://msdn.microsoft.com/en-us/library/ms188796.aspx`, 2016.

[89] Microsoft. Microsoft SQL Server caching mechanisms. `https://msdn.microsoft.com/en-us/library/cc293623.aspx`, 2016.

[90] Jason Millman. Health care data breaches have hit 30m patients and counting. `https://www.washingtonpost.com/news/wonk/wp/2014/08/19/health-care-data-breaches-have-hit-30m-patients-and-counting/`, 2015.

[91] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, New York, NY, USA, 2nd edition, 2017.

[92] MongoDB. `http://www.mongodb.org`.

[93] Mongo. Monitoring in MongoDB. `https://docs.mongodb.com/manual/administration/monitoring/`, 2016.

[94] MongoDB. MongoDB client-side encryption. `https://www.wired.com/story/field-level-encryption-databases-mongobd/`, 2019.

[95] Replica set oplog. `https://docs.mongodb.com/manual/core/replica-set-oplog/`.

[96] Microsoft transparent data encryption. `https://msdn.microsoft.com/en-us/library/bb934049.aspx`.

[97] InnoDB memcached Plugin. `https://bit.ly/3edTmRD`.

[98] Moni Naor and Vanessa Teague. Anti-persistence: History independent data structures. In *STOC*, 2001.

[99] Navajo Systems. `http://tinyurl.com/y85obds6`.

[100] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *CCS*, 2015.

[101] Muhammad Naveed, Manoj Prabhakaran, and Carl A Gunter. Dynamic searchable encryption via blind storage. In *S&P*, 2014.

[102] Neo4j. `http://neo4j.com/`.

[103] Office of Personnel Management. Cybersecurity incidents. `https://www.opm.gov/cybersecurity/cybersecurity-incidents/`, 2015.

[104] Capital One. Information on the Capital One cyber incident. `https://www.capitalone.com/facts2019/`, 2019.

[105] Oracle transparent data encryption. `http://www.oracle.com/technetwork/database/options/advanced-security/index-099011.html`.

[106] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *OSDI*, 2016.

[107] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind Seer: A scalable private DBMS. In *S&P*, 2014.

[108] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. What storage access privacy is achievable with small overhead? In *PODS*, 2019.

[109] MySQL performance schema. `http://dev.mysql.com/doc/refman/5.7/en/performance-schema-statement-tables.html`.

[110] Giuseppe Persiano and Kevin Yeo. Lower bounds for differentially private rams. In *EUROCRYPT 2019*, 2019.

[111] Perspecsys: A Blue Coat Company. `http://perspecsys.com/`.

[112] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system. Cryptology ePrint Archive, Report 2016/591, 2016.

[113] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: an encrypted database using semantically secure encryption. *VLDB*, 2019.

[114] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[115] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building web applications on top of encrypted data using Mylar. Cryptology ePrint Archive, Report 2016/893, 2016.

[116] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using Mylar. In *NSDI*, 2014.

[117] Raluca Ada Popa, Nickolai Zeldovich, and Hari Balakrishnan. Guidelines for using the CryptDB system securely. Cryptology ePrint Archive, Report 2015/979, 2015.

[118] Postgres. The statistics collector. `https://www.postgresql.org/docs/current/static/monitoring-stats.html`, 2016.

[119] MySQL query cache. `https://dev.mysql.com/doc/refman/5.7/en/query-cache.html`.

[120] Redis. `http://www.redis.io`.

[121] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.

[122] RocksDB. `http://rocksdb.org`.

[123] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In *EUROCRYPT*, 2006.

[124] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *IEEE S&P*, 2016.

[125] Salesforce.com, 2018. `https://www.salesforce.com`.

[126] N. Sauer. On the density of families of sets. *Journal of Combinatorial Theory, Series A*, 13(1):145–147, 1972.

[127] A unified testbed for evaluating different Oblivious RAM. https://github.com/InitialDLab/SEAL-ORAM.

[128] Rocco A Servedio. Lower bounds for learning discrete distributions.

[129] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. Social Hash: An assignment framework for optimizing distributed systems operations on social networks. In *NSDI*, 2016.

[130] Skyhigh Networks. https://www.skyhighnetworks.com/.

[131] Swaminathan Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *SIGMOD*, 2012.

[132] The slow query log. http://dev.mysql.com/doc/refman/5.7/en/slow-query-log.html.

[133] Nikolai V Smirnov. Estimate of deviation between empirical distribution functions in two independent samples. *Bulletin Moscow University*, 1939.

[134] ServiceNow, 2018. https://www.servicenow.com.

[135] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *S&P*, 2000.

[136] Apache Spark. Monitoring and instrumentation. http://spark.apache.org/docs/latest/monitoring.html, 2016.

[137] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE S&P*, 2013.

[138] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious ram protocol. In *CCS*, 2013.

[139] The Infrastructure Behind Twitter: Scale. https://bit.ly/2zLrDsI.

[140] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 1984.

[141] V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971. Translation by B. Seckler.

[142] Verizon data breach incident report. `https://regmedia.co.uk/2016/05/12/dbir_2016.pdf`, 2016.

[143] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, 2020.

[144] Mor Weiss and Daniel Wichs. Is there an oblivious ram lower bound for online reads? In *TCC*, 2018.

[145] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1945.

[146] Yupeng Zhang, Jonathan Katz, and Babis Papamanthou. All your queries are belong to us: the power of file injection attacks. In *USENIX Security*, 2016.

[147] Wenting Zheng, Frank Li, Raluca Ada Popa, Ion Stoica, and Rachit Agarwal. MiniCrypt: Reconciling encryption and compression for big data stores. In *EuroSys*, 2017.