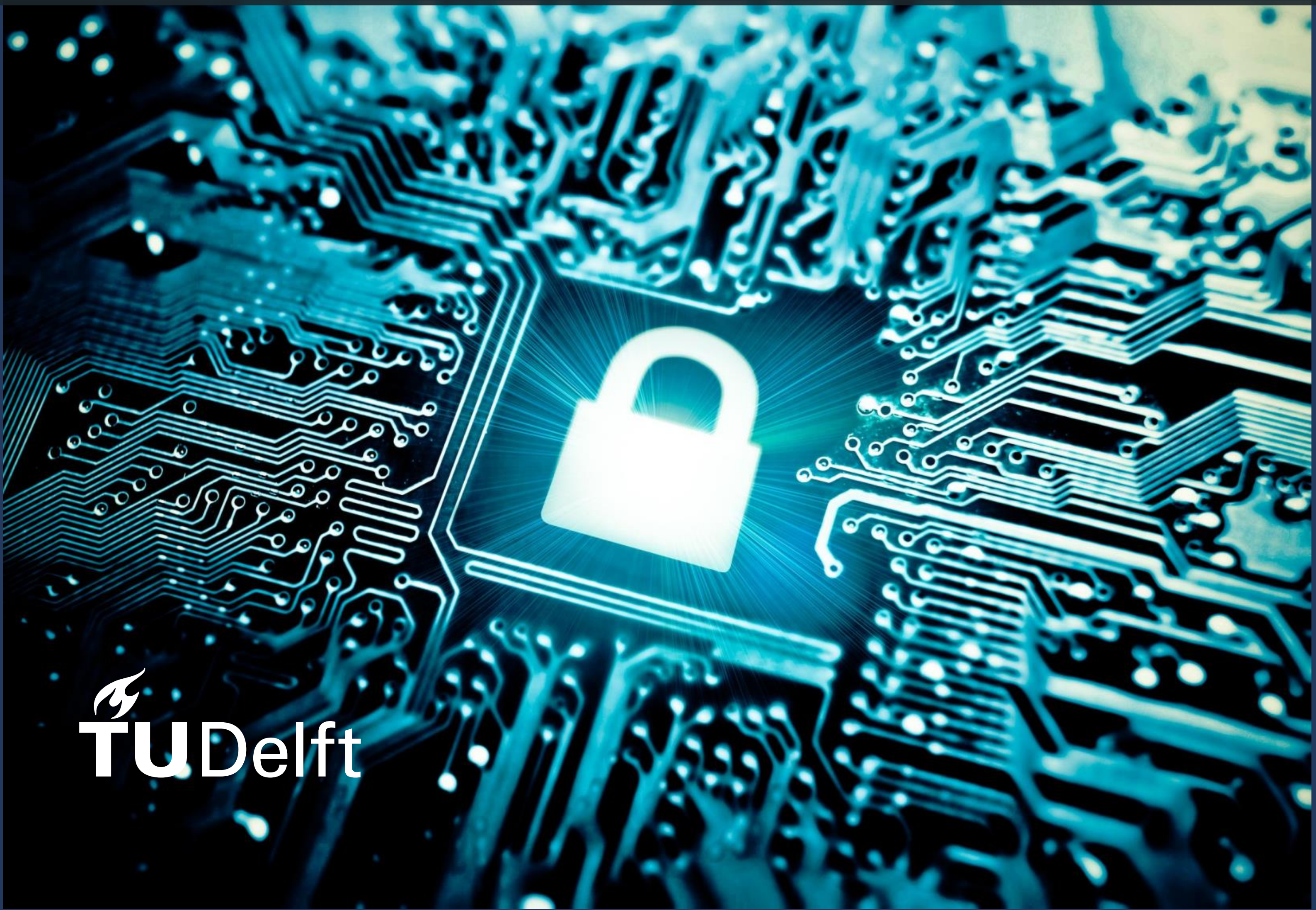
Engineering Encrypted Multi-Maps with Controlled Access and Volume Leakage for Multi-Dimensional Queries

Viraj Biharie



Engineering Encrypted Multi-Maps with Controlled Access and Volume Leakage for Multi-Dimensional Queries

Thesis report

by

Viraj Biharie

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on September 11, 2025 at 12:30

Thesis committee:

Chair:	Prof. G. Smaragdakis, Faculty EEMCS, TU Delft
Daily supervisor:	Dr. E.A. Markatou, Faculty EEMCS, TU Delft
Co-daily supervisors:	Dr. K. Liang, Faculty EEMCS, TU Delft
	H. Chen, M.Sc., Faculty EEMCS, TU Delft
External examiner:	Dr. P. Pawelczak
Place:	Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS), Delft
Project Duration:	September, 2024 - September, 2025
Student number:	5107350

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS) · Delft University of Technology

Abstract

This work addresses the challenge of performing expressive, multi-dimensional range queries directly over encrypted data while balancing query efficiency against privacy leakage. Existing searchable encryption and encrypted multi-map (EMM) schemes either reveal access or volume patterns or incur substantial overhead by fully hiding all leakage (e.g. via ORAM) making them unpractical. Building on the static, multi-dimensional EMM framework of Falzon et al. [1], we introduce a family of five EMM variants that provide tunable leakage profiles, spanning from full access- and volume-pattern exposure to near-complete concealment through adaptive padding and dummy-access techniques. The empirical evaluation of our five EMM variants reveals a clear, quantifiable spectrum of privacy-performance trade-offs. On large-range workloads, the access-hiding schemes offer the best overall balance, with measured average latency slopes of ≈ 0.012 ms/label. For workloads dominated by small result sets, a volume hiding scheme excels, achieving an even lower slope of 0.0032 ms/label by tuning its padding to realistic occupancy bounds. In contrast, fully padded schemes like incur substantially higher overheads, up to two orders of magnitude greater, making them suitable only when maximal leakage resilience is required. These results allow cloud providers with quantitative guidance to deploy encrypted range search that meets both privacy requirements and performance expectations in real-world, multi-attribute database services.

Preface

The journey of writing this thesis has been, without doubt, one of the most transformative periods of my academic life. When I first started on this project, I was filled with excitement but also had my doubts: uncertain of my direction, anxious about my capacity to contribute something meaningful, and worried that the scope of the work might overwhelm me. Yet, through countless hours of reading, coding, experimenting, and rewriting, what began as an uncertain exploration grew into a deeply rewarding investigation into encrypted multi-maps and controlled leakage for multi-dimensional queries.

I am profoundly grateful to my supervisors, whose encouragement and expertise made this journey possible. To Dr. Lilika Markatou, my main supervisor, thank you for your unwavering support, for patiently answering my late-night questions, and for always challenging me to think more rigorously and creatively. Your feedback sharpened my arguments and enhanced my understanding of the field. I would also like to extend my heartfelt thanks to Dr. Kaitai Liang, whose incisive comments on early drafts of this thesis helped me to refine both the direction and presentation of my work. Finally, I am deeply indebted to Huanhuan Chen, who formed the backbone of this thesis and who spent countless hours brainstorming optimizations and troubleshooting experiments.

My sincere appreciation goes to Dr. Georgios Smaragdakis for his generous mentorship and for taking the time-despite an exceptionally busy schedule-to discuss emerging ideas and offer strategic guidance at each pivotal stage. Those conversations not only clarified complex concepts but also reminded me of the broader significance of this research.

To my family and friends, who provided moral support and a listening ear, thank you. Your faith in me carried me through the long nights and tight deadlines, and I dedicate this thesis to you.

It has been an amazing adventure, and I look forward to the next chapter-both in research and beyond.

Viraj Biharie Delft, September 2025

Contents

List of Figures

List of Tables

Acronyms

EMM	Encrypted Multi-Map	2
SSE	Searchable Symmetric Encryption	6
ESA	Encrypted Search Algorithms	5
ORAM	Oblivious RAM	5
FHE	Fully Homomorphic Encryption	5
DET	Deterministic Encryption	5
OPE	Order-Preserving Encryption	5
AUC	Area Under the Curve	17
MM	Multi-Map	22
FPEMM	Fixed-length Padding Encrypted Multi-Map	23
AHEMM	Access-Hiding Encrypted Multi-Map	23
VHAHEMM	Volume-Hiding Access-Hiding Encrypted Multi-Map	24
DPRFMM	Delegatable Pseudo-Random Function Encrypted Multi-Map	24
realDPRFMM	Realistic Delegatable Pseudo-Random Function Encrypted Multi-Map	25
PRF	Pseudo-Random Function	13
dPRF	Delegatable Pseudo-Random Function	25
KDF	Key Derivation Function	32
PPT	Probabilistic Polynomial-Time	45
STE	Structured Encryption	45

Part I

Foundations

1 Introduction

In this chapter we introduce the central theme of this thesis: the design and evaluation of privacy-preserving, multi-dimensional range queries over encrypted data. It begins by outlining the motivation behind this work, particularly the growing need to balance data utility with privacy requirements in outsourced database settings. The chapter then presents the problem statement, showing the limitations of current approaches that either compromise on performance or expose sensitive access patterns. It also defines the scope of the research, focusing on passive adversaries and static datasets, and introduces the core research questions and objectives that guide the study.

1.1. Context & Motivation

In recent times, organizations across a wide spectrum of sectors, such as finance, healthcare and geospatial services, have increasingly outsourced the storage and management of large datasets to third-party providers [2], [3]. These datasets often contain sensitive information, for instance, financial transaction logs, patient records with timestamps and lab values, or detailed spatial coordinates for critical infrastructure, that must remain confidential. At the same time, end-users demand flexible range queries (for example, retrieving all records whose geographic coordinates fall within a defined range) without sacrificing data privacy. Encrypted multi-dimensional range search schemes answer this need by allowing servers to process queries directly on ciphertexts, but not without revealing some well-defined information, or *leakage*, about the queries and underlying data.

This topic is particularly timely due to several concurrent trends. First, regulatory frameworks such as GDPR and HIPAA are imposing stricter requirements on data confidentiality and auditability, this makes it such that organizations are being pushed towards encryption-first architectures [4], [5]. Second, (recent) high-profile breaches of unencrypted or poorly secured database systems have reinforced the real cost of data exposure [6], [7], [8] . Finally, as cloud infrastructures grow in performance, stakeholders expect low latencies on queries, even on encrypted databases [9].

Building on a previously developed Encrypted Multi-Map (EMM) for range search by Falzon et al. [1], this research extends the original scheme by designing and evaluating EMM variants with tunable leakage profiles. Specifically, we construct versions that range from fully concealing both volume and access patterns to fully exposing them, enabling a detailed comparison of leakage-performance trade-offs. Through systematic experiments over multi-dimensional datasets, we demonstrate how varying levels of leakage affect query latency and privacy guarantees. By doing so, we provide:

- **Data Owners:** with clear guidance on selecting the appropriate leakage profile to meet regulatory and operational requirements.
- **Cloud Service Providers:** with quantitative metrics to tune system implementations for desired performance-privacy trade-offs.
- **Security Researchers:** with empirical evidence to assess and certify encrypted range search schemes.
- **End-Users:** with assurance that the chosen EMM variants align with their needs for query speed and data confidentiality.

The design and evaluation performed in this thesis ensures that each group can make informed decisions when deploying encrypted multi-dimensional range search in real-world settings.

1.2. Problem Statement

Although EMMs enable expressive range queries on ciphertexts, existing constructions either leak detailed volume and access patterns or impose heavy performance penalties in pursuit of absolute confidentiality. This lack of intermediate, configurable options leaves data owners and system integrators without a principled, systematic way to navigate the privacy-performance trade-off. The problem this thesis addresses is thus twofold: (1) *how to systematically introduce tunable leakage profiles into a multidimensional EMM*, and (2) *how to measure and compare privacy guarantees and performances of each profile*. By solving this, we allow stakeholders to select the encrypted search scheme that best meets their operational and security requirements.

1.3. Research Questions & Objectives

This section provides an overview of the primary research objective and research questions in this project.

Research Objective

The primary objective of this research project is to design, implement, and empirically evaluate encrypted multi-dimensional range search EMM variants with distinct leakage profiles, and to quantify how varying levels of volume-pattern and access-pattern leakage impact query performance and privacy guarantees.

Research Question 1

How do the EMM leakage profiles influence the average trapdoor generation time, search time, resolve time, and result count under identical dataset and query workloads?

Research Question 2

What quantitative relationship exists between specific leakage metrics and each performance metric?

Research Question 3

How does varying the dataset configuration impact the performance metrics of each EMM variant?

Research Question 4

Which EMM variants achieve the most favorable security–performance trade-off across different datasets and query selection scenarios?

1.4. Scope & Contributions Overview

For the adversary model in this work, we consider a passive, honest-but-curious third-party adversary. This adversary observes query transcripts and encrypted data structures, but does not actively interfere with the protocol or manipulates the data in any way.

Regarding the functional scope, this thesis focuses on range queries over multi-dimensional datasets, evaluation EMM variants differing in volume-pattern and access-pattern leakage. We do not address active attacks, side-channel leakage beyond the aforementioned patterns, or updates to the encrypted dataset (e.g., insertions and deletions).

Contributions Overview:

- **EMM Variant Design:** We extend the baseline multi-dimensional EMM by implementing five variants with different leakage profiles, ranging from full leakage to complete concealment of both volume and access patterns.
- **Implementation & Experimental Framework:** We update the open-source implementation of Falzon et al. [10] to support all five leakage profiles.
- **Benchmark Extension:** We update the existing benchmark suite to be compatible with multi-dimensional, multi-valued datasets for encrypted range searches.
- **Synthetic Data Generator:** We develop a configurable data generation module capable of producing synthetic datasets that reproduce latitude-longitude points of road network intersections in California, with tunable parameters.
- **Empirical Evaluation:** We conduct controlled experiments on synthetic multi-dimensional datasets, measuring average trapdoor generation time, search time, resolve time, and result count under identical query workloads.
- **Trade-off Analysis:** We quantify the relationship between leakage metrics and performance outcomes.

1.5. Thesis Outline

This thesis is structured into four main parts, each building upon the previous from foundational concepts through to original contributions and final insights.

Part I: Foundations (Chapters 1–3) establishes the context for this research, introduces key concepts, and surveys related work. Chapter 1 presents the motivation, objectives, research questions, and contributions of the thesis. Chapter 2 offers background on searchable encryption, leakage types, adversary models, and multi-dimensional data structures. Chapter 3 reviews prior work in encrypted search, identifying limitations in existing approaches that this thesis addresses.

Part II: Methodological Framework (Chapters 4–5) introduces the proposed EMM variants and describes the experimental methodology used to evaluate them. Chapter 4 details six EMM constructions that offer different trade-offs between leakage, efficiency, and storage. Chapter 5 outlines the experimental design, including dataset generation, query protocols, performance metrics, and implementation details.

Part III: Results & Analysis (Chapters 6–8) presents the performance results of the EMM variants, validates their correctness and security, and interprets their implications. Chapter 6 reports benchmarking results across synthetic datasets for all schemes. Chapter 7 provides security proofs, complexity analyses, and correctness checks for the constructions. Chapter 8 discusses the broader impact of the findings, comparing results with prior work and reflecting on the limitations.

Part IV: Closure (Chapters 9–10 and Appendices) summarizes the thesis outcomes and points to future research directions. Chapter 9 concludes the thesis by reviewing how the research questions were addressed and lists the key takeaways. Chapter 10 suggests ways for extending this work, such as supporting dynamic updates and improving leakage resilience. The appendices contain supplementary materials, including extended figures and algorithms referenced in the main chapters.

2 Background

This chapter provides the necessary background and theoretical foundations for understanding the challenges and approaches involved in enabling efficient and secure range queries over encrypted data. It surveys relevant literature on searchable encryption, leakage profiles, and encrypted multi-map data structures, showing both their capabilities and inherent trade-offs. This chapter also introduces key concepts and definitions used throughout the thesis, such as volume and access-pattern leakage, and contextualizes the motivation behind different security-performance trade-offs.

2.1. Encrypted Search Algorithms

The goal of the client is to privately query their data stored on an untrusted server, in such a way that any third party adversary is unable to figure out the detail of this query. Specifically, the adversary should ideally be unable to determine which data was accessed or the volume of data retrieved. Given that the data on the untrusted server is encrypted, performing a search over this encrypted dataset requires the use of Encrypted Search Algorithms (ESA). These algorithms are designed to help with efficient and secure searching within encrypted data spaces.

While ESA provide an efficient and secure means of searching over encrypted data, they are not without limitations, particularly in terms of privacy leakage through access and volume patterns. Other advanced cryptographic techniques, such as Oblivious RAM (ORAM) [11] and Fully Homomorphic Encryption (FHE) [12, 13], further enhance the privacy and security of these systems.

FHE enables a client to perform mathematical operations, such as addition and multiplication, directly on encrypted data. This capability is helpful when such computations are resource-intensive and must be offloaded to the server. FHE allows the server to execute these operations without decrypting the data, thereby preserving its confidentiality. However, this approach adds a significant performance overhead. The process typically requires multiple rounds of communication between the client and the server to ensure security, as a result FHE is not yet a practical solution for real-world, high-performance workloads.

ORAM is a cryptographic protocol that hides a client's entire access pattern by continually reshuffling and re-encrypting memory blocks as they are read and written. In an ORAM system, every logical read or write by the client is transformed into a sequence of indistinguishable physical accesses, so that an observer cannot tell which memory block is being accessed or how often. Security is typically established via a simulation-based proof: for any real ORAM execution leaking nothing but its overall length, there exists an efficient simulator that produces an indistinguishable transcript without knowing the client's true access sequence [11, 14, 15]. Despite this strong privacy guarantee, ORAM adds very high performance costs, often an order-of-magnitude or more in bandwidth and latency overhead. Due to this high cost, ORAM, like FHE, remains largely impractical for low-latency, large-scale deployments.

Before discussing some more advanced schemes, it's helpful to understand two simpler schemes that trade off privacy for efficiency: Deterministic Encryption (DET) [16, 17] and Order-Preserving Encryption (OPE) [18, 19]. DET encrypts each plaintext value to the same ciphertext every time. This enables *equality* searches (e.g., "find all records where field = X") by directly comparing ciphertexts. This makes DET very fast to index and query and require minimal storage and computation overhead. However, the trade-off is that DET leaks duplicate-value frequencies, making it vulnerable to frequency-analysis attacks when the plaintext distribution is skewed or known in advance. OPE encrypts values in a way that preserves their sort order, allowing range queries (e.g., “find all records where field < Y”) by comparing ciphertexts. OPE supports inequality and range queries directly on ciphertext without additional indexing. Having said that, OPE leaks the entire plaintext order, enabling statistical or inference attacks. Additionally, the formal security of OPE is weak compared to FHE or ORAM.

2.2. Searchable Symmetric Encryption

One instantiation of ESA is Searchable Symmetric Encryption (SSE) [16, 19, 20, 21, 22], which enables efficient, sub-linear keyword querying over encrypted data by generating search tokens from a secret key and secure index structures. SSE enables a client to outsource encrypted data to an untrusted server while still admitting efficient keyword- or attribute-based search over ciphertexts. Rather than decrypting an entire dataset, the client generates a lightweight *trapdoor* for each query, which the server uses to locate matching encrypted records. SSE offers a practical middle ground, allowing for more expressive queries than simple DET or OPE, yet more efficient than the performance heavy ORAM or FHE. However, in exchange for this effiency, SSE reveal some well-defined information, or *leakage*, about the queries and underlying data.

SSE schemes come in two types: static and dynamic. In static SSE, a single encrypted index is built once over a fixed dataset, and all queries operate on this immutable structure. In contrast, dynamic SSE supports secure insertions, deletions, and updates, enabling modification of the encrypted data without rebuilding the index. In this work, we focus exclusively on static SSE and its multi-dimensional extension. We build on the foundation laid by Falzon et al. [1], who also considered static-only SSE, and reserve the exploration of dynamic updates for future research. From this point on, any reference to SSE refers specifically to **static** SSE, unless explicitly noted otherwise.

A typical SSE scheme follows a clear sequence of steps, illustrated in the protocol flow below. Refer to Figure 2.1 for a graphical representation of this protocol flow:

1. KeyGen: The client generates a secret key K.
2. BuildIndex: Using K, the client constructs an encrypted index (e.g., inverted lists, hash tables) over the dataset and uploads it to the server.
3. Trapdoor: For each query (e.g., “all records with attribute X in range Y”), the client uses K to compute a trapdoor token T.
4. Search: The server applies T to its encrypted index, retrieves matching ciphertexts (CT, and returns them to the client.
5. Decrypt & Resolve: The client decrypts the results and performs any necessary post-processing (filtering, sorting, aggregation) to recover the plaintext (PT).

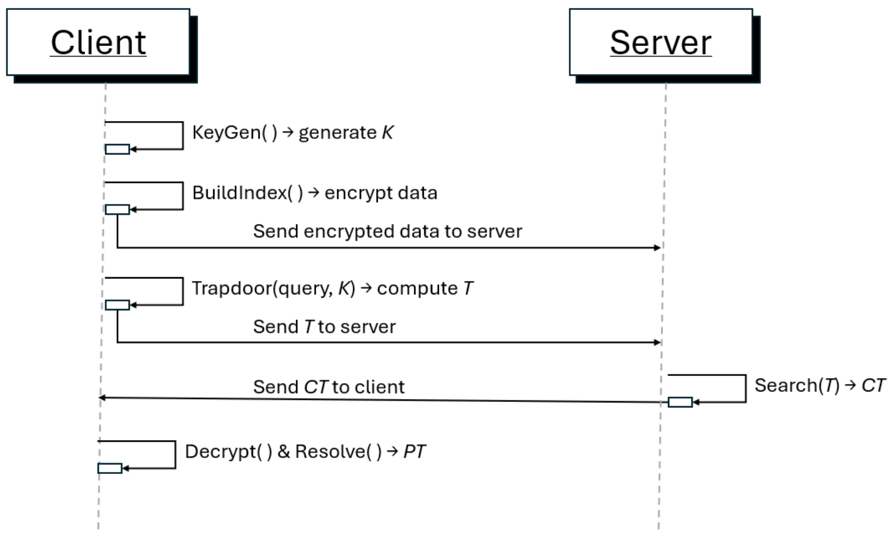

Client
Server
KeyGen( ) → generate K
BuildIndex( ) → encrypt data
Send encrypted data to server
Trapdoor(query, K) → compute T
Send T to server
Send CT to client
Search(T) → CT
Decrypt( ) & Resolve( ) → PT

Figure 2.1: High-level SSE Workflow: KeyGen, BuildIndex, Trapdoor, Search, Decrypt & Resolve.

Under SSE, encrypted data is typically organized in one of these index structures [16, 23, 24]:

- **Encrypted Inverted Lists**: mapping each keyword or attribute value to a list of encrypted document identifiers.
- **Hash-Based Indices**: providing constant-time lookup at the cost of more complex key management.
- **Tree-Based Indices**: supporting range queries by preserving a logical order within encrypted nodes.

In this work, certain proposed EMMs use hash-based index structures to organize encrypted mappings efficiently.

2.3. Leakage Patterns

As stated before, SSE schemes enable a client to outsource an encrypted document collection to an untrusted server, while still being able to issue keyword-search queries over that collection. In practice however, achieving sublinear search time and reasonable communication overhead typically requires revealing some *leakage* about the data and query process. Formalizing and understanding this leakage is therefore important both to interpreting the practical security of SSE schemes and for defending against so-called *leakage-abuse attacks*.

Intuitively, leakage encompasses any information that the server learns beyond the bare fact that "a search query returned some encrypted documents". Formally, each SSE scheme is proven secure relative to a *leakage profile* [25]:

$$\Lambda = \big(\mathcal{L}_{\text{setup}},\ \mathcal{L}_{\text{query}}\big) \tag{2.1}$$

where $\mathcal{L}_{\text{setup}(D)}$ (setup leakage) is whatever the server learns at setup time from the (encrypted) collection D and $\mathcal{L}_{\text{query}((D,q_1,\ldots,q_t))}$ (query leakage) is whatever the server learns by observing a sequence of t keyword queries $q_1, \ldots, q_t$. A leakage-parameterized security proof shows that only these functions and nothing else about the plaintext data or queries, can be inferred by any polynomial-time adversary. We distinguish three common leakage patterns, each of which can be exploited in different ways.

2.3.1. Access Pattern

The access pattern (also called the identifier pattern) captures exactly which encrypted records match each query. Even though the server only sees ciphertexts, every time it evaluates a trapdoor it learns the set of ciphertexts (or record-identifiers) that satisfy the query. Formally defined as [25]:

Definition 1 (Access Pattern) *Let $D = (C_1, C_2, \ldots, C_n)$ be the encrypted database comprising ciphertexts C_i, and let $\mathcal{W} = (T_1, T_2, \ldots, T_t)$ be a sequence of trapdoor tokens corresponding to t queries. The* access pattern *is the function*

$$\mathsf{AP}(D, \mathcal{W}) \;=\; \big(\mathsf{ids}(T_1),\, \mathsf{ids}(T_2),\, \ldots,\, \mathsf{ids}(T_t)\big),$$

where each

$$\mathsf{ids}(T_i) \;=\; \{\, j \in [n] : C_j \textit{ matches the trapdoor } T_i \}$$

is the set of record-identifiers whose ciphertexts satisfy the i-th query.

The access pattern captures exactly which encrypted records the server retrieves in response to each query. Concretely, every time the client issues a trapdoor T_i, the server returns a set of ciphertexts. Even though their contents remain encrypted, the identity of those ciphertext slots leaks to the server. Over multiple queries, an adversary can correlate these observed sets to infer relationships among plaintexts.

For example, suppose two distinct queries T_a and T_b both return ciphertext slots $\{C_{17}, C_{42}\}$. The server now knows that records 17 and 42 share both attributes a and b. By combining this with other query results, it can build up co-occurrence graphs, frequency profiles, or even reconstruct portions of the underlying data. In worst-case scenarios, such leakage has enabled inference attacks that recover up to 95% of keywords in simple SSE settings [25, 26].

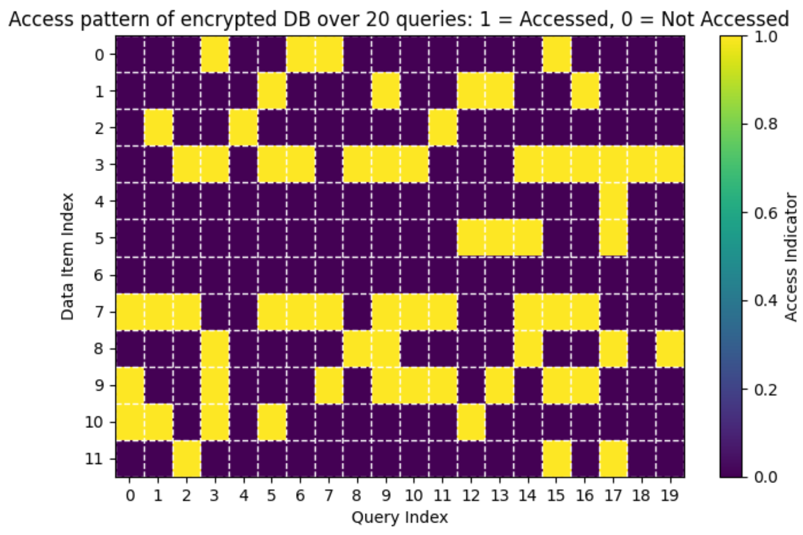


Figure 2.2: Heatmap showcasing the access pattern on an encrypted database

Figure 2.2 illustrates an access-matrix heatmap over 20 successive queries against an encrypted database of 12 items. Each row corresponds to a single encrypted data block (indexed 0-11) and each column to one query in time (0-19). A bright (yellow) cell indicates that the server observed a block being accessed during that query; a dark (purple) cell indicates no access. *Note: This heatmap is purely illustrative. The simulated access patterns do not derive from, nor relate to, any datasets discussed elsewhere in this work.*

A quick scan reveals three "hotspot" rows: row 3, row 7 and row 9, which show access in a large fraction of queries. All other rows light up relatively infrequently and more randomly. In particular:

- **Row 3:** accessed in 65% of the time steps.
- **Row 7:** accessed in 60% of the time steps.
- **Row 9:** accessed in 45% of the time steps.
- **Other rows:** accessed at a baseline rate of about 30% or less.

This skewed pattern stands out even though each data block remains encrypted; the server cannot see the contents, but it can easily count how often each block is touched and when.

An adversary who passively records this matrix can exploit several pieces of information. First of all, by ranking blocks by access frequency, the adversary can identify which items are “hot”, i.e. frequently accessed: rows 3, 7 and 9. In many applications, hotspot maps can directly link to popular records, e.g. frequently read medical files or trending products. In addition, if the adversary also knows approximate query times, and therefore also the query indices, (e.g. office hours vs. off-hours), they can infer when certain data is in demand. For instance, if row 3 spikes during morning queries, it might correspond to a morning-only report. Over repeated observations, the adversary can build a profile of user behavior. This allows them to answer questions like ”Which encrypted records correlate with specific workflows?” or ”How do access frequencies change over time (e.g. seasonal or weekly patterns)?” Finally, if the adversary can inject known queries or correlate with side channels (like network traffic size), they may begin to map block indices to actual data items. Once a mapping is established for one block, frequency and co-access patterns can help to de-anonymize other blocks, allowing an adversary to reconstruct the entire encrypted database.

Because the access pattern effectively reveals a deterministic “pointer set” for each query, hiding it requires heavy machinery, such as fetching uniformly sized dummy results, shuffling result order, or using ORAM-style oblivious accesses, all of which add substantial bandwidth and computation overhead. Our baseline EMM construction uses a response-hiding technique to conceal which individual records are retrieved (revealing only the number of results), and our tunable variants explore the spectrum between that full concealment and the raw, unprotected access pattern, thereby allowing us to quantify precisely how much performance must be sacrificed to defend against these powerful inference vectors.

2.3.2. Volume Pattern

The volume pattern reveals, for each issued trapdoor T_i, the number of ciphertext returned by the server, even if their identities remain hidden. Formally, if the encrypted database $D = (C_1, ..., C_n)$ and trapdoor sequence $W = (T_1, ..., T_t)$ are as before, the volume pattern is the function [25]:

Definition 2 (Volume Pattern) *Let D and $\mathcal{W}$ be as in Definition 1. The* volume pattern *is*

$$\mathsf{VP}(D, \mathcal{W}) \;=\; \bigl(|\mathsf{ids}(T_1)|,\ |\mathsf{ids}(T_2)|,\ \ldots,\ |\mathsf{ids}(T_t)|\bigr),$$

where each $\mathsf{ids}(T_i)$ *is the set of record indices matching* T_i.

Even though the server does not learn which records correspond to a query, observing only the counts $v_i = |ids(T_i)|$ across many queries can leak statistical information about the dataset. For instance, repeated queries that return the same volume suggest identical or similar predicates, enabling an adversary to cluster queries by result size. More importantly, if the adversary has auxiliary knowledge of the underlying data distribution (e.g., typical city-population ranges or identifiable peak traffic volumes), they may map volumes back to likely plaintext values.

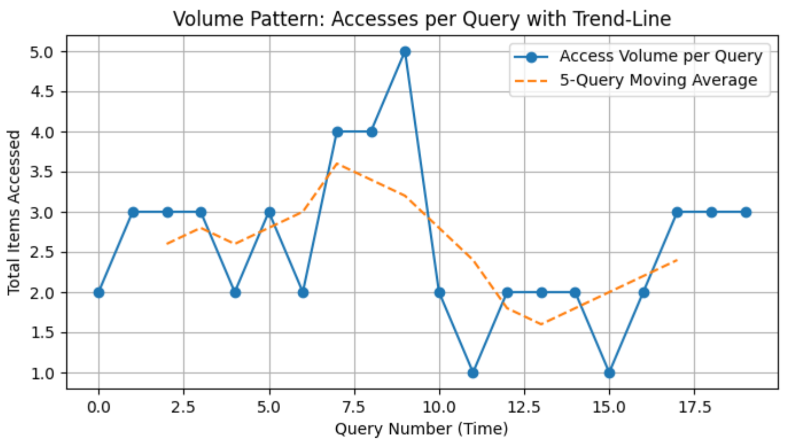


Figure 2.3: Trend-line illustrating the volume pattern on an encrypted database

Figure 2.3 illustrates a line plot of the volume pattern, showing the total number of accessed items per query over 20 queries, with a 5-query moving average as a trend-line:

- **Blue markers (connected by solid line):** Actual access volume at each query (sum of the column in the access matrix).
- **Dashed line:** 5-query moving average, revealing medium-term trends in query size.

In figure 2.3 it can be seen that certain queries (e.g., Query 9) peak with 5 accesses, while others dip to as low as 1. The trend-line smooths out short-term noise, highlighting that overall the system sees a mild rise in volume around Queries 6-9, followed by a dip and then a gradual uptick again.

An adversary monitoring just these volumes, without seeing any encrypted payload, can distinguish "heavy" operations (e.g. full-table scans or complex range queries) from "light" lookups. Over time, the attacker could correlate larger volumes with specific user actions (batch reports vs. single-record fetches), refine models of user behavior, and potentially fingerprint query types purely from these counts.

In practice, volume leakage has enabled size-oracle attacks where adversaries recover sensitive numerical attributes, such as salary ranges or geospatial densities, from encrypted query traffic [25]. Mitigating this leakage typically requires padding results to a fixed size or adding dummy ciphertexts, both of which increase bandwidth and client-side filtering costs. Our EMM variants systematically vary the degree of volume hiding, from no padding (full leakage) to maximal padding (zero leakage), so that we can measure exactly how much extra communication and processing overhead is required to defend against such volume-based inference attacks.

2.3.3. Search Pattern

The search pattern captures when two or more trapdoors correspond to the same underlying query predicate. Even if each trapdoor is randomized under the client's key, schemes that permit deterministic or keyed-deterministic token generation will produce identical tokens for identical queries. By observing token equality, the server learns whether the client is repeating the same search, which can reveal behavioral or temporal correlations.

Definition 3 (Search Pattern) *Let $\mathcal{W} = (T_1, T_2, \ldots, T_t)$ be the sequence of trapdoor tokens issued by the client. The* search pattern *is the equivalence relation*

$$\mathsf{SP}(\mathcal{W}) \;=\; \{\,(i,j) \in [t] \times [t] : T_i = T_j\}.$$

Equivalently, $\mathsf{SP}(\mathcal{W})$ *partitions the set of queries into classes of identical predicates.*

Because the server can directly test equality of tokens T_i, it immediately knows whenever the client repeats a query. This leakage allows an adversary to link queries over time. They can track when the same predicate is issued (e.g., "show my recent transactions" run daily). Furthermore, they can infer query patterns or interests by clustering repeated searches (e.g., monitoring frequently accessed ranges). Finally, they can combine search-pattern linkage with access and volume patterns to more rapidly reconstruct query semantics or user activity.

Mitigating search pattern leakage typically requires randomized or stateful trapdoor generation, ensuring that even identical predicates map to fresh tokens. However, such approaches can complicate caching or server-side optimization and may introduce additional communication to synchronize token states.

In our experiments, all five EMM variants inherit the baseline's deterministic trapdoor behavior, so they fully expose the search pattern. We include the search pattern in this section for completeness, but our EMM variants do not attempt to mitigate or empirically evaluate search pattern leakage in this work. Exploring randomized trapdoors remains an area for future enhancement.

2.3.4. Structure Pattern

Falzon et al. [1] identify a form of leakage that they call structure pattern, i.e. the pattern of co-occurrence of subqueries. It can be formally defined as:

Definition 4 (Structure Pattern) *Let $\mathcal{W} = (T_1, T_2, \ldots, T_t)$ be the sequence of trapdoor tokens produced by the client's range-cover algorithm, and for each token T_i let*

$$\mathsf{labels}(T_i) \;=\; \{\ell_1, \ell_2, \ldots, \ell_{k_i}\}$$

be the (ordered) set of canonical subrange labels that cover the query q_i. The structure pattern *is the sequence*

$$\mathsf{SP}_{\mathsf{struct}}(\mathcal{W}) \;=\; \big(\mathsf{labels}(T_1),\, \mathsf{labels}(T_2),\, \ldots,\, \mathsf{labels}(T_t)\big).$$

Although the server only sees encrypted tokens, it learns exactly which canonical subranges co-occur in each query. By analyzing these label-tuples, an adversary can infer the shape and granularity of a user's range requests, potentially distinguishing, say, long thin rectangles from more square regions, or correlating subranges to specific multi-dimensional predicates. Such leakage can enable structure-abuse attacks, where the adversary builds a query co-occurrence graph to reverse-engineer the client's intended ranges.

We include the structure pattern in this section for completeness, but our EMM variants are not specifically designed to mitigate or measure structure-pattern leakage. Nevertheless, if any variant increases the amount or distinguishability of subrange co-occurrences versus the Falzon et al. [1] baseline, we will report that fact alongside our other leakage measurements.

2.4. Threat and Adversary Model

We assume a **passive, honest-but-curious adversary** that controls (or observes) the untrusted server storing the encrypted data and index. The adversary:

- **Follows the protocol faithfully**, returning correct ciphertexts and never deviating from prescribed algorithms.
- **Observes all client–server communications**, including encrypted index uploads, trapdoor tokens, and returned ciphertext bundles.
- **Learns only the defined leakage functions** (e.g. access, volume, search, structure patterns), but not plaintext content or the secret key.

This simplified two-step interaction model (client issues trapdoor(s) $\longleftrightarrow$ server returns encrypted results) captures all information available to the adversary during search operations. Formally:

$$\mathsf{View}_{\mathcal{A}} = \big(\mathcal{L}_{\mathsf{BuildIndex}}(D),\ \mathcal{L}_{\mathsf{Search}}(D, T_1), \ldots, \mathcal{L}_{\mathsf{Search}}(D, T_t)\big),$$

where $\mathcal{L}_{\mathsf{BuildIndex}}$ and $\mathcal{L}_{\mathsf{Search}}$ are the leakage functions defined by the SSE/EMM scheme.

Building on this model, the adversary may combine observed leakage with auxiliary knowledge:

- **Passive Observation:** The adversary can record $(\text{trapdoor},\ \text{ciphertexts})$ pairs for many queries, but does not inject or modify messages.
- **Leakage Abuse:** The adversary can exploit leaked patterns (access, volume, structure) to correlate ciphertexts, recover query predicates, or reconstruct portions of the underlying data.

While our focus is on quantifying and trading off access and volume leakage in multi-dimensional EMM variants, this honest-but-curious model equally supports attacks exploiting search and structure patterns, which we discuss for completeness in Sections 2.3.3 and 2.3.4.

2.5. Encrypted Multi-Dimensional Maps

Encrypted Multi-Dimensional Maps extend the concept of an encrypted multimap, where a single attribute or keyword is mapped to a set of encrypted record identifiers, to datasets with two or more dimensions. In a traditional, single-dimensional SSE multimap, the client builds an index that associates each keyword with its list of document ciphertexts; during a search, the server receives a trapdoor for one keyword and returns the corresponding encrypted postings. Falzon et al. [1] formalize this idea in higher dimensions by partitioning a multi-attribute domain into a fixed collection of canonical "cells" or hyper-rectangles, each of which is treated as a label in the map . A client's arbitrary range query over d attributes is then translated, via a range-cover algorithm, into the union of these cells, and trapdoors are generated for each included label.

Multi-dimensional EMMs differ from their single-dimensional counterparts in several fundamental ways. First, the label space grows combinatorially with the number of dimensions, so range-cover efficiency becomes critical: Falzon et al. [1] compare linear, range, quadratic and tree-based covers to balance the number of tokens against lookup overhead. Second, leakage expands beyond the usual access and volume patterns to include the structure pattern, which reveals exactly which combination of cells a query spans. Finally, the storage and runtime costs of encrypting and traversing a high-dimensional map can be orders of magnitude greater-requiring thoughtful optimizations to the underlying SSE primitives.

Research into multi-dimensional encrypted multimaps is still in its early stages, making the work of Falzon et al. [1] one of the first to demonstrate practical, privacy-preserving range queries over ciphertext in more than one dimension. Consequently, our EMM variants, which introduce tunable leakage profiles on top of this multi-dimensional foundation, are themselves novel contributions. By tweaking our baseline setup to show or hide different levels of access and data leakage, we broaden searchable encryption into more real-world scenarios. This gives practitioners practical, hands-on advice for rolling out multi-dimensional encrypted metadata models while keeping performance and privacy in check.

3 Related Work

This chapter surveys the most relevant literature in encrypted range search, focusing on foundational definitions of leakage, constructions of EMM schemes, and practical attacks on existing designs. The chapter then critically analyzes these works. The insights from these works provide the basis for the design goals and innovations introduced in this thesis.

3.1. Related Work

The story of privacy-preserving encrypted search begins with Oblivious RAM, introduced by Goldreich and Ostrovsky [11]. ORAM formalizes obliviousness: any observed access sequence must reveal no information about the data beyond its length. In practice, ORAM schemes (e.g. the square-root or hierarchical constructions) face orders-of-magnitude overhead. In other words, hiding every access is extremely expensive. This motivated more practical searchable encryption models that trade some leakage for efficiency.

Over time it became clear that this leakage must be controlled. In a seminal result, Kellaris et al. [27] showed that essentially every encrypted search scheme leaks either the access pattern or the response size (or both). They developed reconstruction attacks using repeated range queries to recover entire data sequences from just access and volume leaks. The conclusion was that any residual leakage can be exploited. In light of this, researchers explored ways to suppress one form of leakage while tolerating the other.

A key line of work focused on hiding the volume (response size) of queries. Kamara and Moataz [28] introduced practical computational volume-hiding schemes for encrypted multi-maps. Their constructions (VLH and AVLH) use small-domain pseudo-random graph encodings to mask list lengths, providing tunable privacy–efficiency trade-offs: by adjusting padding lengths, Pseudo-Random Function (PRF) output range, or graph density, one can control how much volume information is exposed. Importantly, these relax information-theoretic guarantees in favor of efficiency. In parallel, Patel et al. [29] gave a formal simulation definition of volume-hiding leakage and proposed the dprfMM scheme. Their design uses cuckoo hashing and PRFs so that every (key, label) lookup returns a fixed-length response, thus hiding the true number of matching records. These works showed that one could mask response sizes much more cheaply than ORAM. (Later follow-up built dynamic variants with similar guarantees, though this thesis focuses on the static case.)

In parallel with single-attribute search, the community tackled multi-attribute (multi-dimensional) range queries. Falzon et al. [1] did this, by adapting classic spatial indexes to encrypted multi-maps. They presented six “range-cover” schemes (Linear, BRC, Quad-BRC, etc.), each derived from a plaintext range tree or quadtree, to support orthogonal range queries in d-dimensions. This was the first general framework for encrypted multi-dimensional range search, but each scheme fixed a particular storage–query trade-off. In other words, the Falzon framework achieved functionality but with a discrete set of leakage–performance points.

Importantly, attacks soon caught up to these multi-dimensional schemes. Just as Kellaris demonstrated generic attacks in 1D, Markatou et al. [26] showed how residual leakage from multi-dimensional range tokens can betray data structure. They formalized structure-pattern leakage: the co-occurrence of subrange tokens across dimensions. By observing which combinations of encrypted subqueries yield results (along with response sizes), an adversary can infer the underlying spatial relationships among records. In fact, Markatou et al. [26] designed reconstruction attacks on canonical multi-dimensional schemes (Linear, Union-of-Rectangles, etc.) that recover entire datasets up to geometric symmetries. These results reinforced the lesson from Kellaris that hiding only one leakage channel is insufficient. Effective multi-dimensional schemes must manage both access-pattern and volume leakage simultaneously.

Taken together, this evolution points toward tunable encrypted multi-maps that allow practitioners to control privacy vs. efficiency. Early work maximized security (full ORAM) at great cost [11], but gradually research developed intermediate models. For example, Boldyreva and Tang [30] recently proposed an ORAM-based EMM that hides queries, access patterns, and volume completely, which shows that full security is still attainable (at ORAM-like overhead). In contrast, other approaches inject padding and randomness to expose more leakage in return for performance. Modern designs (including this thesis's contributions) span the spectrum: by tuning dummy padding and randomized access, one can adjust how much volume or access information leaks [28, 29, 14, 30]. In this way, the field has progressed from absolute obliviousness toward flexible privacy–performance trade-offs.

3.2. Critical Analysis and Identified Gaps in Existing Work

Despite the progress made in encrypted range search, a review of the related literature reveals several persistent gaps, limitations, and challenges. These shortcomings motivate and shape the contributions of this thesis.

Most existing schemes offer a fixed balance between leakage and performance. ORAM (Goldreich & Ostrovsky) guarantees full access-pattern hiding but suffers from prohibitive overhead. Falzon et al. provide a discrete spectrum of six fixed trade-off points across their constructions. However, none of these works allow developers to tune leakage along a continuum based on application-specific constraints. This thesis introduces tunable-leakage EMM variants that provide a flexible interface to control padding rates and dummy accesses, offering more control over the leakage-efficiency curve.

Key contributions such as Kamara & Moataz's VLH/AVLH and Patel et al.'s dprfMM/dpMM concentrate solely on single-attribute searchable encryption. While Markatou et al. demonstrate the vulnerability of multi-dimensional range queries to inference, existing protections have not been adapted to this broader context. Our work directly addresses this gap by generalizing volume- and access-pattern hiding techniques to the multi-dimensional case, allowing for more realistic and secure query models.

Many prior solutions assume static datasets. Although Kamara & Moataz sketch dynamic extensions for VLH and AVLH, and ORAM handles updates with high overhead, practical, efficient, dynamic multi-dimensional schemes remain unexplored. While our current work focuses on static datasets, we explicitly identify the challenges of dynamic padding and suggest a direction for future multi-dimensional update mechanisms under tunable leakage constraints.

Much of the literature centers on volume hiding (e.g., Kamara, Patel), leaving access-pattern leakage unaddressed or inadequately mitigated in multi-attribute settings. Yet, as Markatou et al. show, co-occurrence of subquery tokens can leak spatial structure through access patterns alone. Some of our EMM variants provide the first parameterizable techniques for suppressing access-pattern leakage in the context of geometric EMMs.

While many papers evaluate latency and bandwidth, few provide a quantitative analysis of leakage. Kellaris et al. give a theoretical attack model, and Markatou et al. present successful real-world attacks. However, no prior work evaluates the impact of specific padding or access strategies on adversarial success. Our framework bridges this gap by benchmarking under varying leakage configurations, enabling more informed privacy-performance decisions.

Most surveyed works assume an honest-but-curious server and do not address active or side-channel adversaries. Only ORAM explores tamper-resistance through versioning. We retain the honest-but-curious model for tractability but propose clear extensions in our future work. These can include techniques such as dynamic defenses, authenticated queries, and hybrid leakage-resilient schemes.

Finally, many theoretical contributions (e.g. Patel, Kamara) lack system integration, leaving their practical deployability ambiguous. Falzon et al. provide the most directly usable framework, but with limited customization and analysis tooling. We built on their prototype by adding customizable settings and a fully open-source, modular implementation, making it easier to plug into real-world systems.

Part II

Methodological Framework

4 Design of EMM Variants

This chapter presents the design and rationale behind a family of EMM variants developed to explore and control the trade-offs between privacy and efficiency in multi-dimensional range search.

4.1. Research Paradigm

This study is *quantative* in nature [31, 32]. All primary research activities, including the design of EMM variants, generating synthetic datasets and measuring metrics, produce numerical data that can be analyzed using statistical methods. In particular, controlled experiments ensure that each EMM variant is evaluated under identical dataset and query conditions. Additionally, performance metrics are aggregated over multiple queries to compute means, slopes, and Area Under the Curve (AUC) scores [33, 34, 35]. By focusing on quantifiable measurements, we can objectively compare privacy-efficiency trade-offs across variants.

While our *quantitative* framework provides clear numerical evidence, the study could be augmented by *qualitative* insights from practitioners [31, 32]. For example, structured interviews or surveys with database engineers and compliance officers could reveal how real-world constraints (e.g., ease of integration, perceived risk tolerance) influence the choice of leakage profile, thus adding context to the numbers and informing more user-centered design choices.

4.2. Comparison of EMM Schemes

This section gives a brief overview of all EMM schemes that will be introduced in this chapter and summarizes this in Table 4.1.

Baseline EMM: The original EMM stores each record individually encrypted under a deterministic label-based index. Concretely, each value is encrypted as a separate ciphertext keyed by a hash of its label and position. This yields one ciphertext per actual record (no dummy padding) and minimal storage overhead. The server thus learns exactly how many values match each query (full volume leakage) but, owing to the hashing, cannot directly link ciphertexts to plaintext records. In effect, only the number of results (not their identities) is revealed. Access-pattern leakage (at the key or range level) is not protected: the server sees which labels/tokens are queried, but it sees no information about individual record accesses. Overall, Baseline EMM offers the lowest overhead (no padding or grouping) but leaks full volume (result counts) and unprotected key-level access patterns.

FPEMM (Fixed-length Padding EMM): This variant pads every key's posting list to a fixed length L_{max} equal to the largest list in the dataset. After padding, each key has exactly L_{max} encrypted entries, so the server sees the same number of ciphertexts per key. Queries retrieve L_{max} ciphertexts for each matching key (with dummy values filtered by the client). This fully hides the volume pattern: the number of real results is indistinguishable (constant) across all queries. However, like the baseline, FPEMM still leaks which keys are queried (it does not hide access patterns). The trade-off is significant storage and bandwidth overhead: every small list is inflated to length L_{max}, leading to potentially many dummy entries. In datasets with skewed lists, this can greatly expand storage (to roughly $N \times L_{max}$ ciphertexts for N keys) and query cost. FPEMM thus achieves perfect volume privacy at the cost of high fixed padding overhead.

AHEMM (Access-Hiding EMM): AHEMM hides access patterns by encrypting each entire posting list as a single ciphertext. Instead of multiple ciphertexts per key, each key maps to one large encrypted tuple of all its values. The server then only observes which key-level ciphertexts are fetched, with no visibility into which individual values they contain. Consequently, access leakage is eliminated as the server cannot distinguish which records within a list are accessed. However, volume leakage remains, since the decrypted size of each tuple equals the true list length, so exact result counts are revealed to the client (and thus measurable). Storage overhead is low (no padding), and the total number of stored records is unchanged, but there are fewer ciphertexts overall (one per key). Performance-wise, AHEMM can reduce the number of encryptions/decryptions for large lists, though each ciphertext is larger. In summary, AHEMM removes acces–pattern leakage (one ciphertext per key) at very low overhead, while still leaking full volume information.

VHAHEMM (Volume-Hiding, Access-Hiding EMM): VHAHEMM combines FPEMM's padding with AHEMM's grouping. Each key's list is first padded to L_{max} (as in FPEMM) and then encrypted as a single tuple (as in AHEMM). The encrypted index thus has one ciphertext per key containing exactly L_{max} values. This uniform structure leaks neither the access pattern nor the volume pattern: the server cannot tell how many real entries are in each tuple (since it sees constant-size ciphertexts) and cannot see which values are accessed within a tuple. Both volume and access leakage are eliminated. The cost is the overhead of padding every list to L_{max} and encrypting large ciphertexts. Storage is inflated to $N \times L_{max}$ entries (one per key), and each query fetches one full-length ciphertext per key. Compared to AHEMM, VHAHEMM incurs only "modest" extra cost: still one encryption/decryption per key, but with a larger padded payload. In short, VHAHEMM achieves maximal privacy (no volume or access leakage) at the expense of global fixed padding.

DPRFMM (Delegatable PRF EMM): DPRFMM achieves strong volume- and access-hiding by using a pseudorandom placement (cuckoo hashing) and delegatable PRFs. Each key-value pair is placed into a small number of fixed-size buckets (plus a stash), and all buckets and stash slots are padded to their maximum capacity. Queries are issued via a single dPRF token that lets the server find the two buckets and stash for that key without client transmission of all positions. Importantly, the server always returns a constant-sized bucket of ciphertexts for each query, filling with dummies as needed. This means every query for any key returns exactly the same number of ciphertexts, fully masking both which key was queried (beyond seeing a token) and how many real results existed. In other words, DPRFMM hides both access and volume patterns perfectly. The trade-off is very high overhead: the index stores padded cuckoo buckets and stash for all keys, and each query returns many dummy values. Storage grows to (#$buckets \times B + stash$) entries (worst-case bucket size B), and each query decrypts $2B + S$ ciphertexts per key. DPRFMM thus offers the strongest leakage protection (constant-size oblivious retrieval) but incurs the highest storage and computation costs.

realDPRFMM (Realistic DPRFMM): This variant relaxes DPRFMM's padding to reduce overhead, at the cost of controlled leakage. realDPRFMM pads buckets only to a smaller bound P (the typical, not worst case, occupancy). Each query now returns at most P ciphertexts per bucket, introducing some variability in response size. As a result, approximate volume information leaks (the server sees how many up to P items matched) and some access information (e.g. inferring if a key's bucket is near-full). However, by tuning P the scheme bounds this leakage. The upside is much lower overhead: storage and bandwidth scale with P instead of the extreme bucket size. In practice, realDPRFMM lies between DPRFMM and the lightweight schemes in both performance and leakage. It yields far fewer dummies than DPRFMM and thus higher efficiency, at the cost of controlled leakage.

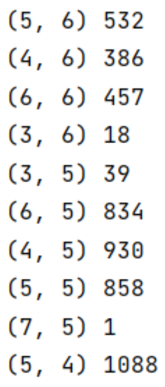
Scheme	Access Leakage	Volume Leakage	Storage Overhead	Comments
Baseline EMM	Records hidden, but keys exposed (no special access hiding)	Full (exact counts)	None (only actual entries)	Minimal overhead; reveals query result sizes; simplest construction.
FPEMM	Keys exposed (same as baseline)	None (constant L_{max} per key)	High (pad all lists to L_{max})	Perfect volume hiding; heavy padding storage and bandwidth cost.
AHEMM	None (only key-level ciphertexts seen)	Full (exact tuple lengths)	Low (sum of lists unchanged)	Hides access patterns by tuple encryption; leaks sizes; one ciphertext/key.
VHAHEMM	None (tuple-level hiding)	None (fixed-size L_{max} lists)	High (pad to L_{max} per key)	Hides both; stores one padded tuple per key; moderate overhead over AHEMM.
DPRFMM	None (uniform fetch patterns)	None (constant-size responses)	Highest (pad all buckets & stash)	Full oblivious retrieval via cuckoo hashing; constant $2B + S$ per query. Very costly.
realDPRFMM	Partial (coarse occupancy info)	Partial (bounded by pad P)	Moderate (pad to $P \leq B$)	Tunable trade-off: relax DPRFMM padding to P for efficiency; leaks up to P entries.

Table 4.1: Overview of all EMM schemes presented in this thesis.

Each scheme reflects a point on the leakage-performance spectrum. Baseline EMM and AHEMM have minimal overhead but expose full volume, while FPEMM and DPRFMM fully hide volume (at high cost). VHAHEMM offers full concealment of both leakage types with moderate padding cost, and realDPRFMM sits in between by padding only to a practical bound.

4.3. Synthetic Data Generation

We build upon the public California road-network dataset originally used by Falzon et al. [1] and [36]. In its canonical form, this dataset is a collection of *(x,y)* latitude-longitude coordinate pairs, each mapped to a single numeric attribute. Please refer to figure 4.1 for an excerpt of what the data looks like.



```
(5, 6) 532
(4, 6) 386
(6, 6) 457
(3, 6) 18
(3, 5) 39
(6, 5) 834
(4, 5) 930
(5, 5) 858
(7, 5) 1
(5, 4) 1088
```

Figure 4.1: Excerpt of original California road-network data

Ideally, we would use this data as is. However, the single-valued nature per coordinate of this dataset made it incompatible with certain EMM variants that expect a different data distribution. Therefore, to test our EMM variants, we developed a synthetic generator that generalizes the California schema to be more compatible with these EMMs. Instead of only storing a single numeric value per coordinate, now we store, at each unique *(x,y)* key, a **tuple** of values. Each tuple is drawn randomly from {1, 2, ..., *m*}, where *m* is a user-configurable maximum.

Here are all the adjustable parameters listed for this synthetic data generation process:

- The **Key Range Sampling:** rather than using a geographic bounding box, we simply specify a *key_range:* $\{k_{min}, k_{max}\}$. Each coordinate *(x,y)* is chosen by sampling *x* and *y* independently and uniformly from the interval $[k_{min}, k_{max}]$.
- **Number of Entries** *N*: this represents the total count of distinct *(x,y)* keys generated.
- **Maximum Tuple Size** *m*: as explained above, this is the upper bound on the length of the value tuples; actual lengths are randomly drawn from the interval [1,*m*].
- **Value Domain**: Each element of every tuple is sampled uniformly from a specified numeric interval (e.g. $[v_{min}, v_{max}]$); these represent the actual "payload" values that our EMM protects.

Algorithm 1 describes this procedure in pseudo-code.

Algorithm 1 Synthetic Data Generation

```
 1: procedure SynthesizeData($n, K, V, m$)
 2:                 ▷ $n$: number of entries; $K = (k_{min}, k_{max})$; $V = (v_{min}, v_{max})$; $m$= max tuple size
 3:     Clear existing data
 4:     all_keys $\leftarrow$ CartesianProduct($K$)
 5:     if $n >$ length(all_keys) then
 6:         raise Error("Too many entries")
 7:     end if
 8:     chosen_keys $\leftarrow$ random sample of $n$ keys from all_keys
 9:     buckets $\leftarrow$ [ ]
10:     for $e$ from $\lfloor \log_{10}(v_{min}) \rfloor$ to $\lfloor \log_{10}(v_{max}) \rfloor$ do
11:         $\ell \leftarrow$ max $v_{min}, 10^e)$
                  (
12:         $h \leftarrow$ min($v_{max}, 10^{e+1} - 1$)
13:         if $h > \ell$ then
14:             Append $(\ell, h)$ to buckets
15:         end if
16:     end for
17:     for each $k$ in chosen_keys do
18:         $\ell \leftarrow$ random integer in $[1, m]$
19:         $v \leftarrow$ [ ]
20:         for $i = 1$ to $\ell$ do
21:             $(low, high) \leftarrow$ random choice from buckets
22:             Append random integer from $[low, high]$ to $v$
23:         end for
24:         data[$k$] $\leftarrow$ tuple($v$)
25:     end for
26: end procedure
```

The `SynthesizeData` procedure (line 1) begins by wiping any previous contents (line 2) so that we build a fresh dataset each time. We then enumerate all possible keys in the 2-dimensional integer grid $K = [k_{min}, k_{max}]^2$ (line 3). If the user requests more unique entries than exist in that grid, we immediately error out (lines 4-6), otherwise we randomly sample exactly n distinct keys (line 7). We do this, because the data is stored in a structured *dictionary*, this means that every key needs to be distinct.

Next, we create the value range $V = [v_{min}, v_{max}]$ into "order-of-magnitude" buckets. This step was created to have a more balanced distribution of the values, since if the value range V was very large, then due to randomness, the application was more inclined to pick larger numbers over smaller numbers. Whereas now, we create these buckets and randomly select one of them for a fairer distribution between values. Specifically, this is done by looping exponent e from $log_{10}v_{min}$ up to $log_{10}v_{max}$ (line 9). After which, we form subranges $[max(v_{min}, 10^e), min(v_{max}, 10^{e+1} - 1]$ (lines 10-11). The positive width values are then stored in the list `buckets` (lines 12-14).

Finally (starting at line 16), we visit each sample key k. We decide at random how many values to attach, anywhere from 1 up to the maximum tuple size m (line 17), and then for each position in the tuple, pick one of our precomputed buckets and draw a uniform integer from that bucket (lines 19-21). This gives a variable-length tuple of integers whose magnitudes are spread across different orders of magnitude. We store that tuple under key k in the result map (line 23).

To summarize, this design ensures several key outcomes. First of all, uniform coverage of the key space, due to sampling without replacement. Then, it allows for log-uniform scaling of values, by choosing buckets per order of magnitude. And lastly, variable tuple lengths, which is the biggest difference in comparison to the original California dataset.

4.4. Baseline EMM Construction

Falzon et al. [1] define an EMM as a tuple of four algorithms Σ = (Setup, Query, Eval, Result) that operate over a plaintext Multi-Map (MM) mapping each label (in our case, a canonical range) to a set of record values. In our application:

1. **Σ.Setup:** The client builds MM by assigning each node's canonical range to the set of records it covers, then runs Σ.Setup(λ, MM) to produce a secret key K and an EMM stored by the server (with λ being the security parameter).
2. **Σ.Query:** To search range q, the client computes a set of subranges via the **range-cover algorithm**, then invokes Σ.Query(K, ℓ) for each canonical subrange label ℓ to obtain search tokens.
3. **Σ.Eval:** The server evaluates each search token over the encrypted multimap (Σ.Eval) and returns the ciphertext bundles.
4. **Σ.Result:** The client decrypts and aggregates results via Σ.Result.

Regarding the **range-cover algorithms** mentioned above. Encrypted multimaps operate over a fixed set of canonical "labels," but a user's range query q may not align exactly with those labels. A range-cover algorithm bridges this gap by decomposing any arbitrary interval q into a small collection of canonical subranges whose union equals q. This step is essential because it translate a high-level range request into the precise tokens the EMM can handle. Falzon et al. [1] propose a few cover strategies. One of those strategies is the *linear* scheme, which labels sequentially. Another strategy are the *tree-based* schemes, which use a hierarchical decomposition. Although the tree version can sometimes reduce the number of subranges, in practice the linear cover proved both simpler to implement and more reliable at highlighting performance differences across leakage settings. Accordingly, we adopt the **linear-range** cover as our primary instantiation, with some modifications to handle our multi-valued dataset.

Algorithm 2 describes the process of building the encrypted index for the original EMM in pseudo-code. This process is the central component of the EMM schemes and represents the key distinguishing factor among the various variants.

Algorithm 2 Original EMM Build Encrypted Index

```
1: procedure BuildIndex($key, plaintext\_mm$)
2:     $hmac\_key \leftarrow$ HashKDF($key$, PURPOSE_HMAC)
3:     $enc\_key \leftarrow$ HashKDF($key$, PURPOSE_ENCRYPT)
4:     $encrypted\_db \leftarrow \emptyset$
5:     for each $(label, values) \in plaintext\_mm$ do
6:         $token \leftarrow$ HMAC($hmac\_key, label$)
7:         for $index, value \in values$ do
8:             $ct\_label \leftarrow$ Hash($token +$ bytes($index$))
9:             $ct\_value \leftarrow$ SymmetricEncrypt($enc\_key, value$)
10:            $encrypted\_db[ct\_label] \leftarrow ct\_value$
11:        end for
12:    end for
13:    return $encrypted\_db$
14: end procedure
```

The `BuildIndex` procedure (line 1) transforms an in-memory plaintext multi-map `plaintext_mm` into an encrypted lookup structure `encrypted_db`. It begins by deriving two separate symmetric keys from the master secret *key* (lines 2-3): one `hmac_key` for deterministic labeling via HMAC, and one `enc_key` for confidentiality of the values.

On each iteration over the input map's entries (line 5), we first compute a per-label "token" by applying HMAC to the cleartext label (line 6). This token both randomizes the label and ensures it remains consistent across repeated encryptions of the same label.

Within that label's value list (line 7), we enumerate each `index, value` pair. We hash the concatenation of the token and the byte-encoding of the index to produce a unique ciphertext key `ct_label`. This hides both the original label and the position in the tuple, while still allowing lookups by index if the same token is reused (since we are using DET).We apply a standard symmetric encryption to the raw value under `enc_key` , producing `ct_value`, which protects confidentiality. Then, we insert the pair `ct_label → ct_value` into `encrypted_db`, effectively building an oblivious index where neither labels nor values appear in the clear. Finally, once all label-value pairs have been processed, the fully populated `encrypted_db` is returned (line 13).

4.5. FPEMM

The Fixed-length Padding Encrypted Multi-Map (FPEMM) is the first and most privacy-preserving variant introduced in this work to mitigate volume-pattern leakage. This scheme operates under a simple principle: pad all multi-map entries to the maximum tuple length observed across the dataset. While the approach is conceptually straightforward, it establishes a baseline for strong volume-hiding guarantees against passive adversaries.

FPEMM begins by scanning the input dataset to determine the largest posting list size, denoted as $L_{max} = max_k|MM[k]|$, where $MM[k]$ is the set of values associated with key k. Every other key is then padded with dummy values so that each list reaches length L_{max}.

Once padding is complete, each entry (including dummy values) is encrypted as in a standard EMM construction. The server stores a fixed-length encrypted list for each key, ensuring that the ciphertext volume for each key is uniform. This makes it infeasible for the server to infer any information about the actual tuple sizes based on ciphertext length or query response size.

During a query, the client generates tokens for all keys falling within the queried range. The server retrieves L_{max} encrypted entries for each matching key and returns them to the client. The client decrypts each entry and filters out dummy values using embedded metadata or padding markers. This guarantees that all queries reveal only the range of queried keys, but not the number of actual values associated with them.

FPEMM offers perfect volume-hiding: the server observes the same number of encrypted items per key and per query, regardless of the underlying data distribution. This effectively reduces the volume leakage function to a constant and simulates the uniform access behavior of ORAM-based solutions, albeit without hiding the access pattern itself.

The privacy guarantees of FPEMM come at the cost of a overhead. In datasets with high skew, where most keys have small tuple sizes and only a few have long lists, padding to L_{max} causes significant storage expansion. Query bandwidth also increases, as the client must process and filter a large number of dummy entries. Thus, while FPEMM sits at the high-privacy end of the leakage-efficiency spectrum, it is primarily suited for use cases where volume privacy is of great importance.

4.6. AHEMM

The Access-Hiding Encrypted Multi-Map (AHEMM) is designed to mitigate access-pattern leakage by modifying the encryption strategy used in the baseline EMM. In traditional EMM constructions, each value associated with a key is encrypted individually. While this enables flexible retrieval, it also exposes access patterns to the server, revealing which specific encrypted values are returned for each query. AHEMM addresses this vulnerability by aggregating and encrypting entire value-tuples associated with each key as a single unit.

In AHEMM, instead of encrypting each value in a posting list separately, the entire list of values for a key is grouped into a single structure and encrypted as one ciphertext. This design ensures that the server only observes access to the key-level ciphertexts, without visibility into which individual records within the list are retrieved. Thus, any fine-grained inference about internal structure is effectively blocked.

The ciphertext storage under AHEMM differs from baseline EMM only in granularity: rather than maintaining multiple ciphertexts per key, each key maps to a single ciphertext that encapsulates the full tuple. This compact representation reduces the number of stored entries while simultaneously improving access pattern hiding. The client decrypts the ciphertexts and recovers the full lists of associated values. Since the server always sees one ciphertext per key and cannot distinguish internal access behavior, access-pattern leakage is eliminated.

AHEMM completely removes access-pattern leakage at the cost of preserving volume leakage. Although the server cannot determine which individual values are accessed, the decrypted size of each tuple may still be inferred after client-side decryption. As such, AHEMM is ideal for scenarios where access privacy is important, but volume leakage is tolerable.

The tuple-level encryption strategy may reduce the overall number of encryptions and decryptions, especially for keys with large posting lists. However, this comes with increased ciphertext size, and depending on the application, bandwidth may increase if not all records are relevant. Still, the performance trade-off can be considered more favorable when compared to more complex and storage-heavy methods such as ORAM or full padding.

4.7. VHAHEMM

The Volume-Hiding Access-Hiding Encrypted Multi-Map (VHAHEMM) combines the core ideas behind FPEMM and AHEMM to eliminate both volume-pattern and access-pattern leakage. This construction serves as the most privacy-preserving variant in the proposed framework, ensuring that the server cannot infer either the number of values associated with a key or which specific values are being retrieved during a query.

VHAHEMM applies fixed-padding to conceal the volume pattern and tuple-level encryption to hide the access pattern:

- Volume Hiding: As in FPEMM, the maximum posting list size L_{max} is computed across all keys. Each key's list is then padded with dummy values to reach this maximum length.
- Access Hiding: As in AHEMM, each fully padded list is aggregated and encrypted as a single ciphertext. This hides which individual entries within the tuple are accessed, since the entire list is returned and decrypted by the client.

In the encrypted index, every key maps to exactly one ciphertext, which contains an encrypted tuple of L_{max} values. This uniformity ensures that both the number and size of ciphertexts per key are constant across the dataset, preventing any statistical inference by the server.

Because the server sees only one ciphertext per key and cannot distinguish between real and dummy values, both access and volume leakage are effectively neutralized. Thus, VHAHEMM eliminates both access-pattern and volume-pattern leakage. The ciphertext structure prevents key-specific access pattern inference. And fixed-size encrypted tuples ensure that the server learns nothing about the underlying list lengths.

Despite offering the strongest privacy guarantees, VHAHEMM incurs only modest overhead compared to AHEMM:

- Encryption and Decryption: Only one encryption/decryption operation is performed per key.
- Ciphertext Size: Slightly larger due to padding, but still aggregated into a single ciphertext per key.
- Bandwidth: Comparable to AHEMM in terms of ciphertext count, with a minor increase in payload size. In practice, the performance impact is minimal since encryption and decryption costs dominate, and the difference in tuple size is typically not substantial.

4.8. DPRFMM

The Delegatable Pseudo-Random Function Encrypted Multi-Map (DPRFMM) is a more advanced construction inspired by the work of Patel et al. [29], designed to provide strong volume-pattern privacy both during setup and at query time. While previous constructions like FPEMM successfully hide volume leakage in the storage phase through fixed padding, they still leak information during queries due to variable-sized responses. DPRFMM overcomes this limitation by using pseudo-random placement and constant-size query responses.

FPEMM hides volume leakage by padding all posting lists to the same length, but it does not ensure uniform query behavior: depending on the number of keys matched by a query, the server observes a different number of accesses. Thus, even though all keys store the same number of entries, query result sizes still reveal information about the underlying data distribution. DPRFMM addresses this by enforcing fixed-size query responses regardless of the number of matching records.

DPRFMM relies on cuckoo hashing with a stash to store each key–value pair efficiently in one of two possible hash locations. A small-domain PRF determines which two buckets a key maps to, enabling random, yet deterministic, storage. To prevent leakage via bucket content size, all buckets and stash entries are padded to a fixed maximum size, ensuring structural uniformity in storage.

To reduce client–server communication, DPRFMM employs a Delegatable Pseudo-Random Function (dPRF). This cryptographic primitive allows the client to send a compact token that lets the server evaluate the PRF on a predefined domain (e.g. a set of bucket indices) without learning anything beyond the results. The server uses this token to determine which bucket entries correspond to the queried key, eliminating the need for the client to transmit large sets of PRF outputs.

When querying a key, the client sends a single dPRF token. The server uses this to compute the relevant PRF outputs, retrieve the corresponding buckets and stash entries, and return a fixed-sized list of ciphertexts. All responses, regardless of how many real values match the key, contain the same number of entries. Dummy values are inserted and later filtered out by the client during decryption.

DPRFMM achieves volume hiding at setup and query time, because all lists are padded, and every query returns a constant-size response, eliminating observable size variations. Then, DPRFMM hides the access pattern, because the server always retrieves the same number of ciphertexts from predictable structures. The server sees only uniform fetch patterns, regardless of query specifics.

4.9. realDPRFMM

The Realistic Delegatable Pseudo-Random Function Encrypted Multi-Map (realDPRFMM) is a performance-optimized variant of the DPRFMM scheme. It is designed to prioritize efficiency while accepting controlled leakage of access and volume patterns. In contrast to the original DPRFMM, which strictly enforces fixed-size query responses and full padding to hide all leakage channels, realDPRFMM relaxes these constraints to achieve lower storage, bandwidth, and computational costs.

The realDPRFMM targets use cases where absolute privacy is of less importance than responsiveness and scalability. By softening the volume- and access-hiding guarantees, this variant achieves much higher efficiency, making it practical for large-scale or latency-sensitive deployments. It sits between high-privacy schemes like DPRFMM and lightweight schemes like baseline EMM, offering a customizable trade-off.

Unlike DPRFMM, which pads all buckets and stash entries to the worst-case maximum size, realDPRFMM pads these structures to a smaller fixed bound , chosen to support typical, rather than worst-case, occupancy. This significantly reduces the overhead of storing and processing dummy values.

Query responses in realDPRFMM are no longer strictly uniform. Instead, the server returns as many entries as required by the hash structure, up to the padding bound P . While this introduces some volume leakage (e.g. response size correlates with actual data distribution), the leakage is coarse and predictable. The padding depth P acts as a means of controlling this exposure.

RealDPRFMM also leaks more about access patterns than DPRFMM. Since bucket structures are padded less aggressively, the server may infer approximate occupancy patterns per key. However, this leakage is bounded and parameterized, allowing it to be tailored to application-specific tolerances.

5 Experimental Setup

This chapter outlines the experimental setup used to evaluate the performance and privacy characteristics of the EMM constructions proposed in this thesis.

5.1. Benchmark Suite

The benchmark suite implemented for this thesis evaluates and compares the performance the various EMM constructions. The evaluation process is structured as a modular pipeline, as visualized in the flowchart in figure 5.1, and executed through a unified benchmarking framework. This framework ensures that each EMM scheme is assessed under identical conditions, using a shared dataset, query workload, and system interface.

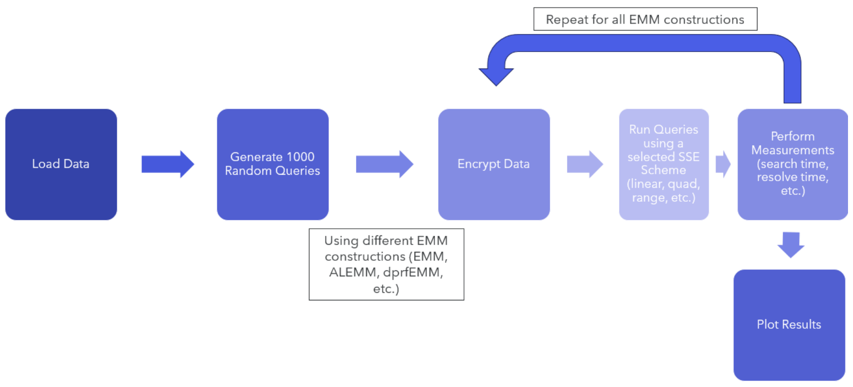

Repeat for all EMM constructions
Load Data
Generate 1000 Random Queries
Encrypt Data
Run Queries using a selected SSE Scheme (linear, quad, range, etc.)
Perform Measurements (search time, resolve time, etc.)
Using different EMM constructions (EMM, ALEMM, dprfEMM, etc.)
Plot Results

Figure 5.1: Flowchart of the Benchmark Suite

The benchmarking process follows a fixed sequence of operations:

1. Load Data: A point-to-tuple map is deserialized from a pickled dataset file and converted into a standardized format.
2. Generate Random Queries: A set of 1000 range queries is randomly generated. These are rectangular in shape and are stratified into coverage buckets based on the fraction of the domain they span, allowing performance to be measured as a function of query selectivity.
3. Encrypt Data: For each EMM engine (e.g. FPEMM, DPRFMM, realDPRFMM), the benchmark suite builds a fresh encrypted index using the EMM engine interface.
4. Run Queries: For each scheme, the full query workload is executed. This includes generating trapdoors, performing server-side search, and resolving ciphertexts into plaintext results.
5. Perform Measurements: The suite records nanosecond-level timings for three stages of query execution: trapdoor generation, encrypted search, and result resolution.
6. Plot Results: All performance data is aggregated, confidence intervals are computed, and comparative plots are generated to visualize the behavior of different schemes across varying query sizes.

This process is repeated for each EMM engine to ensure a fair comparison across constructions.

To assess performance across a spectrum of query selectivity levels, the suite generates a large pool of candidate query rectangles, computes their area as a percentage of the full domain, and groups them into buckets by coverage. A fixed number of queries are then sampled from each bucket to create a balanced workload.

5.2. Metrics & Measurements

Each query execution is broken down into three phases:

- Trapdoor Time: Time to generate encrypted query tokens.
- Search Time: Time to perform encrypted search over the index.
- Resolve Time: Time to decrypt and interpret the result set.

Each timing measurement is repeated across 1000 queries, grouped by coverage percentage, and summarized using means and 95% confidence intervals. These metrics show how the performance scales with query size and dataset layout.

In addition to query-time performance, the suite also records:

- Construction Time: The time required to build the encrypted index from plaintext.
- Storage Footprint: Total memory used to store the encrypted database, including ciphertexts and padding.

These metrics help contextualize each scheme's one-time setup costs versus its ongoing runtime performance.

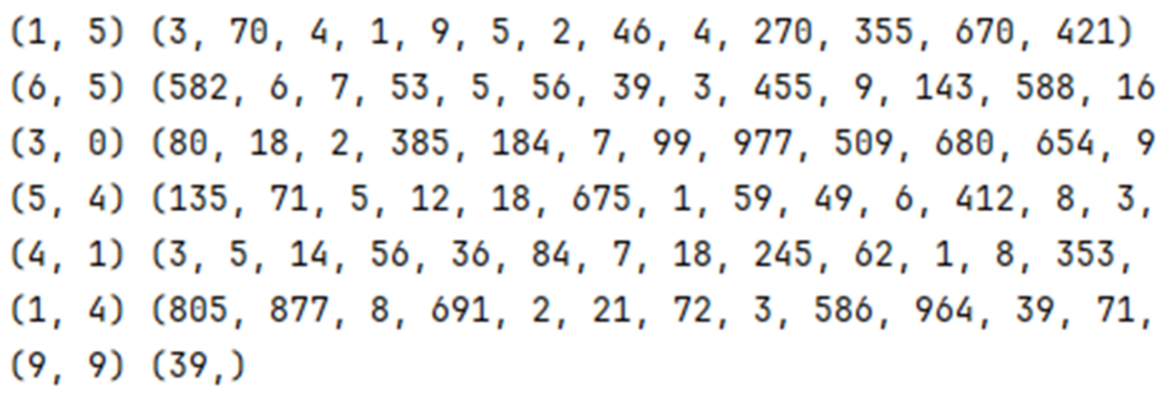
5.3. Dataset Description

The first dataset used in our experimental evaluation is a synthetic multimap designed to provide a controlled environment for validating core functionality and measuring baseline performance. Its parameters are easily adjustable to explore the effects of domain size, tuple density, and padding overhead:

- Number of entries: 32 key–value pairs (points)
- Key domain: integer coordinates in the range [0, 10]
- Value domain: integer values sampled uniformly from [1, 1000]
- Tuple size range: each key maps to between 1 and 50 associated values, chosen uniformly at random

These settings return a spatial domain with widely varying list lengths, making it ideal for observing how padding and tuple-level encryption affect storage and query behavior in the simplest case. By tuning the tuple size range and value distribution, we can directly measure overheads introduced by FPEMM's fixed-padding and DPRFMM's hashing strategies under minimal workload.

Below in Figure 5.2 is a structural excerpt of the raw multimap before encryption. It merely demonstrates how each (x,y) key maps to a list of values (varying in length).



```
(1, 5) (3, 70, 4, 1, 9, 5, 2, 46, 4, 270, 355, 670, 421)
(6, 5) (582, 6, 7, 53, 5, 56, 39, 3, 455, 9, 143, 588, 16
(3, 0) (80, 18, 2, 385, 184, 7, 99, 977, 509, 680, 654, 9
(5, 4) (135, 71, 5, 12, 18, 675, 1, 59, 49, 6, 412, 8, 3,
(4, 1) (3, 5, 14, 56, 36, 84, 7, 18, 245, 62, 1, 8, 353,
(1, 4) (805, 877, 8, 691, 2, 21, 72, 3, 586, 964, 39, 71,
(9, 9) (39,)
```

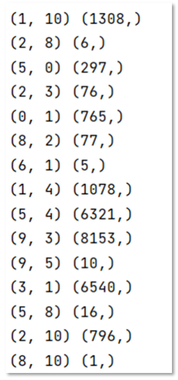
Figure 5.2: Sample screenshot of raw Dataset 1. Only a small subset of the full 32 coordinate →tuple mappings is shown to illustrate the data structure (variable-length tuples).

The second dataset provides a contrasting setup with maximal uniformity across all entries. This dataset is designed to isolate the effects of access-pattern leakage while eliminating the impact of volume variability:

- Number of entries: 32 key–value pairs (points)
- Key domain: integer coordinates in the range [0, 10]
- Value domain: integer values sampled uniformly from [1, 10,000]
- Tuple size range: fixed at 1 value per key

By ensuring that each point maps to exactly one associated value, this dataset avoids any need for volume padding, creating a baseline for evaluating schemes under idealized, low-entropy conditions. It is particularly useful for assessing the overhead introduced by schemes like AHEMM or realDPRFMM, where access-pattern mitigation is the focus. This configuration also allows precise comparison of query execution cost without the confounding influence of large or variable-sized results.

Figure 5.3 shows an excerpt from the second dataset.



```
(1, 10) (1308,)
(2, 8) (6,)
(5, 0) (297,)
(2, 3) (76,)
(0, 1) (765,)
(8, 2) (77,)
(6, 1) (5,)
(1, 4) (1078,)
(5, 4) (6321,)
(9, 3) (8153,)
(9, 5) (10,)
(3, 1) (6540,)
(5, 8) (16,)
(2, 10) (796,)
(8, 10) (1,)
```

Figure 5.3: Sample screenshot of raw Dataset 2. Only a small subset (each coordinate maps to exactly one value) is shown to illustrate the uniform structure.

Figures 5.2 and 5.3 show only a small, random sample of the full datasets to give the reader a sense of the data structure (number of keys, tuple-length variation, value domains). Only the overall shape (e.g. how many values per key, key-domain size) matters for our performance and leakage measurements, whereas the specific integer values themselves do not influence the results if they are of reasonable size (less than 32 bits).

5.4. Experimental Environment

All benchmarks were executed on a single Ubuntu 22.04 LTS virtual machine provisioned with the following hardware resources, chosen to fully accommodate our 16-process Python workload (each holding $\sim$10-12 GB RAM in memory and writing high-throughput logs). Please refer to Table 5.1 for the specifications:

Table 5.1: System Components and Specifications

Component	**Specification**
vCPUs	16 physical cores (NUM_PROCESSES = 16)
RAM	64-128 GB (local peaks of $\sim$12 GB to avoid swapping)
Storage	500 GB NVMe SSD (or EBS GP3, ≥300 MB/s throughput)
Networking	≥10 Gbps
OS	Ubuntu 22.04 LTS
Compiler Toolchains	GCC, Clang
Language & Libraries	Python 3.11; OpenSSL; tqdm; matplotlib;

To ensure no swap-induced variability, we allocated at least 64 GB of RAM despite observed peaks around 12 GB. The NVMe (or equivalently provisioned EBS GP3) volume sustained our log writes (hundreds of MB/s), and the 10 Gbps network link was sufficient for any remote data pulls or multi VM coordination.

5.5. Reproducibility

To ensure that our timing and leakage measurements can be fully replicated, we (1) fix all pseudorandom seeds in Python, (2) document precise software and VM configurations, and (3) publish our codes. A more detailed treatment of randomness sources, seed-management strategies, and statistical stability analyses is deferred to 6.2. The full source code and experimental scripts supporting this thesis are publicly available in the GitHub repository at `https://github.com/vaabiharie/Tunable-EMM-schemes` [37].

Part III

Results & Analysis

6 Experimental Results

In this chapter we present the experimental results obtained from applying the benchmark class to the datasets discussed in section 5.3. We evaluate and report the performance of the datasets based on the metrics outlined in section 5.2.

6.1. Query Generation Protocol

All query workloads are generated using a seeded pseudo-random number generator initialized with a fixed seed. This makes it so that the queries show statistical properties similar to uniformly random sampling. As a result, the generated queries are not distinguishable from truly uniform samples by any efficient adversary.

In order to assess how performance varies with query size, we partition the generated queries into groups based on their coverage of the underlying data domain. Specifically, we define query size as the percentage of the domain covered by the query result (referred to as the ”Relative Query Area”), and we generate sets of queries targeting specific coverage levels (i.e. 5%, 10%, etc.) For each coverage level, we randomly select several queries whose result sizes are close to that coverage, while staying within domain boundaries.

This sampling approach minimizes the variance in performance metrics that could otherwise arise from uneven query distributions. This allows for fairer and more meaningful comparisons across query sizes. In addition, it enables us to observe systematic changes in performance metrics as a function of query size, helping to isolate the effects of coverage from other factors such as data layout or query shape.

6.2. Randomness and Reproducibility

To ensure both the reproducibility and statistical robustness of our results, we adopt a consistent experimental protocol involving controlled randomness and repeated trials.

Each experiment is conducted in five independent batches, corresponding to five distinct random seeds: 0, 1, 2, 3, 4. These seeds are fixed and used consistently across all experimental runs, allowing for exact replication of results. By using the same predefined set of seeds, anyone can rerun our experiments and obtain identical outputs, thereby securing reproducibility.

In terms of randomness inherent in the evaluation process, especially concerning the generation of query sets, we ensure statistical robustness by repeating each experiment five times with independently sampled query sets. Specifically, each batch consists of 1,000 queries sampled independently, and the metrics reported are the mean values computed across these five batches. This aggregation approach mitigates the influence of stochastic variability, reducing the likelihood that observed results are due to random fluctuations.

Should any metric display sensitivity to randomness such that repeated evaluations under the same setup could lead to varying outcomes, we explicitly indicate this in the corresponding analysis.

6.3. Trapdoor Time Comparison Dataset 1

Figure 6.1 presents the mean trapdoor generation time of all evaluated EMM variants, measured over a set of generated queries on dataset 1. In the context of our experiments, trapdoor generation refers to the client-side process of producing a search token corresponding to a given query label. This token is then sent to the server and used to retrieve the matching encrypted entries from the index. Because each query label is mapped to a secure token through cryptographic operations, trapdoor generation forms an important part of the end-to-end latency for SSE schemes. Measuring this phase independently allows us to isolate and assess the performance cost of token derivation in each EMM variant, independent of server-side search and resolution phases.

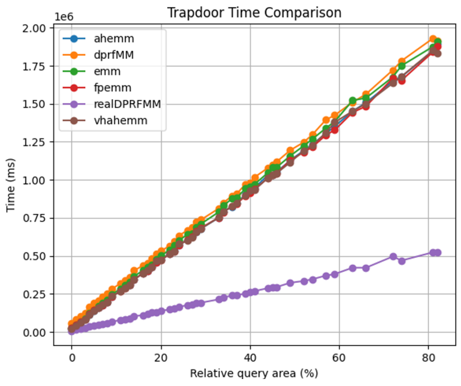


Figure 6.1: Trapdoor Time Comparison of all EMM variants on dataset 1

This figure illustrates the behavior of trapdoor generation across different EMM constructions as the query size varies. The x-axis (Relative query area) represents the geometric size of each query rectangle as a percentage of the total index space, while the y-axis shows the average per-query latency in milliseconds (multiplied by $1e6$). Concretely, for instance the point located at coordinates (40,0.25) indicates that, for the realDPRFMM variant on queries whose rectangles cover 40% of the domain, the average trapdoor generation time was 250 seconds. Each data point reflects the mean result of five independent experimental runs to ensure statistical reliability. However, due to the computational overhead involved, producing the full set of measurements across all variants is a lengthy process that extends over several hours (typically requires over six hours).

In figure 6.1 the ”EMM” curve (*green line*) serves as our baseline. This EMM was based on the EMM originally build by Falzon et al. [1]. All variants show a performance close to this baseline, with the realDPRFMM variant being the one clear exception. Across the full range of query sizes, realDPRFMM consistently achieves faster trapdoor generation than any other construction. The dprfMM curve appears slightly higher than its peers in this figure, which would suggest that this variant has the slowest performance. However, additional experiments with varied random seeds yielded no statistically significant difference among the non-realDPRFMM constructions. In contrast, realDPRFMM’s superior speed remained robust across all trials.

As outlined above, among all evaluated EMM variants, the RealDPRFMM engine consistently shows significantly lower trapdoor generation times across all query sizes. This performance disparity can be attributed to differences in how cryptographic key derivation is handled during the trapdoor phase.

In other EMM designs such as the baseline EMM variant, the trapdoor generation procedure involves two core cryptographic operations: (1) a Key Derivation Function (KDF) to derive an HMAC key from the master secret, and (2) an HMAC computation over the query label using that derived key. While this design ensures that the trapdoor generation is stateless and secure even in multi-session contexts, it also means that every trapdoor generation experiences the full computational cost of a KDF invocation. Since KDFs are deliberately designed to be computationally intensive (to resist brute-force attacks), this overhead becomes a performance bottleneck when trapdoors must be generated frequently.

In contrast, the RealDPRFMM variant optimizes this workflow by computing the derived HMAC key once during the index setup phase and caching it for the lifetime of the engine. As a result, subsequent trapdoor generations require only a single HMAC evaluation over the label, avoiding repeated KDF invocations. Given that HMAC operations are typically several orders of magnitude faster than KDFs, this design leads to substantial performance gains, particularly when generating trapdoors in bulk, as is the case in our experiments.

Importantly, this optimization does not compromise the security assumptions of the system, provided the engine instance remains isolated and the cached key is not exposed across contexts. Rather, it optimizes the performance by reusing derived keys within a controlled execution scope. This design philosophy is not unique to the RealDPRFMM engine and can be adopted by other EMM variants with minimal architectural adjustments. By storing and reusing derived keys within a single execution context, any scheme that currently re-derives keys on each trapdoor invocation can achieve similar performance benefits.

6.4. Search Time Comparison Dataset 1

Figure 6.2 presents the mean search time of all EMM variants. In our experimental setup, search time refers to the server-side computation required to evaluate the trapdoor token against the encrypted index and retrieve the associated ciphertexts. This phase consists of all operations performed by the server after receiving the token, including any key lookups, hash evaluations, ciphertext collection, and intermediate filtering, depending on the design of the EMM scheme. Measuring search time separately allows us to analyze the computational efficiency of the server's response mechanism and to compare how different constructions impact the cost of query evaluation under varying leakage and efficiency trade-offs.

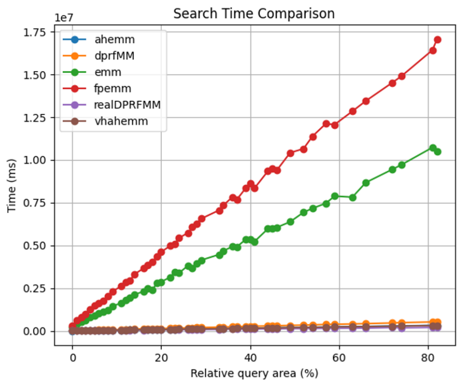


Figure 6.2: Search Time Comparison of all EMM variants on dataset 1

Examining figure 6.2 shows that the curves for the FPEMM (red line) variant and the original EMM (green line), which serves as our baseline, show significantly higher search time costs compared to the other constructions. In particular, the FPEMM curve is the only one that consistently shows a higher search cost than the baseline for this metric. This indicates that the FPEMM construction always takes more time to evaluate the trapdoor tokens and retrieve the corresponding ciphertexts. The reason for this increased cost is that FPEMM applies fixed padding to its encrypted database to mask the volume of each result set. However, it still uses the same search function as the original construction. As a consequence, this variant must search through a larger padded database, without applying any advanced techniques to make the search process more efficient.

Because the FPEMM and original EMM curves dominate the plot so heavily, it becomes difficult to distinguish the performance differences among the remaining variants. To enable a clearer analysis, we remove the dominating curves in the next plots, allowing the behavior of the other constructions to be more easily compared.

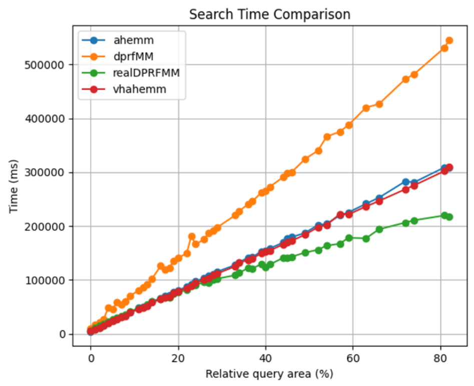


Figure 6.3: Search Time Comparison of EMM Variants (Excluding Baseline EMM & FPEMM) on dataset 1

Comparing figures 6.2 and 6.3 reveals that four variants (AHEMM, DPRFMM, realDPRFMM, and VHAHEMM) all show lower search time costs than the original EMM. However, there are notable differences in performance among them.

Starting with DPRFMM, this construction consistently shows the highest search time among the variants in figure 6.3. This is because DPRFMM applies padding to both its cuckoo hash tables and its query responses in order to hide volume leakage, which increases the search space. In fact, considering both figures, DPRFMM must explore the largest search space of all variants, including FPEMM and EMM. Surprisingly, despite this overhead, it still outperforms both FPEMM and the original EMM. This efficiency can be attributed to its use of delegatable pseudo-random functions, which enable more efficient searching. This improvement comes at the cost of increased search pattern leakage [16, 38, 39], which is outside the scope of the leakage patterns considered in this work.

Next, AHEMM and VHAHEMM demonstrate nearly identical performance. This similarity is expected, as both constructions are structurally alike. The primary difference is that VHAHEMM adds volume-hiding padding on top of the access-pattern protection already present in AHEMM. While there are slight differences in their performance curves, these are not substantial enough to draw any strong conclusions for this metric.

Finally, realDPRFMM achieves the lowest search time across all constructions. This result aligns with its design goal of prioritizing efficiency by selectively allowing certain types of leakage, including access and volume patterns. Since realDPRFMM shares much of its structure with DPRFMM, it also leaks some level of structure pattern, though this is not a focus of the current analysis.

6.5. Resolve Time Comparison Dataset 1

Figure 6.4 presents the mean resolve time of all EMM variants. Resolve time in our experiments refers to the client-side effort required to process the encrypted results returned by the server and extract the final, usable plaintext values. This includes decrypting the ciphertexts and potentially filtering out irrelevant matches based on query semantics. For the sake of our analysis, this step does not discard any dummy or padded entries, but in practice it would do so as part of ensuring the correctness and utility of the final result set. Resolve time captures the final stage of the query pipeline from the client's perspective and is especially relevant in schemes that shift part of the filtering burden to the client, such as those with certain volume-hiding or access-hiding properties.

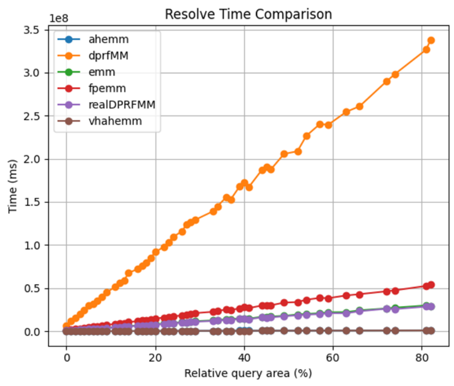


Figure 6.4: Resolve Time Comparison of all EMM variants on dataset 1

Figure 6.4 shows that the DPRFMM construction has a significant overhead compared to the other variants, consistently requiring more time for this resolution step by a considerable margin. The reason for this is that DPRFMM is explicitly designed to offer strong volume-hiding by ensuring that the number of ciphertexts returned per label is fixed and indistinguishable across queries. It achieves this by employing a cuckoo hashing construction where each keyword maps into two buckets (each of size B) and a shared stash (of size S). The server returns all entries in these buckets and the stash, yielding a total of $2B + S$ ciphertexts per queried label.

This volume-hiding strategy results in two dominant sources of computational overhead. First of all, regardless of how many actual matches exist for a given label, the client must receive and process the full padded set. This increases both the total data transferred and the number of ciphertexts that must be deserialized and handled at the client. Especially when B and S are sized for worst-case label density, the overhead becomes significant. Second, to identify which ciphertexts are genuine (i.e. correspond to real data), the client must perform an authenticity check or decryption attempt on every single ciphertext returned. Dummy entries are indistinguishable from real ones until this filtering process is complete. Additionally, because the stash is shared across labels, its handling often involves linear scans or indirect lookups that compound the resolve-time further. These factors together create a substantial computational burden for DPRFMM relative to other EMMs. While lighter schemes such as AHEMM or realDPRFMM allow for more selective decryption (or shorter padded lists), DPRFMM's uniform ciphertext return policy ensures that every query triggers the maximum expected client-side workload.

Similar to the previous section, the dominant cost of DPRFMM makes it difficult to compare the other constructions. Therefore, we excluded DPRFMM from figure 6.5.

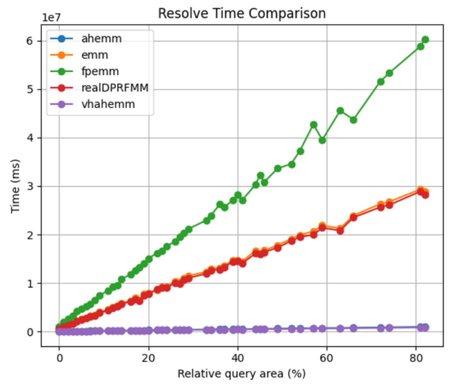


Figure 6.5: Resolve Time Comparison of EMM variants (Excluding dprfMM) on dataset 1

The FPEMM variant shows the steepest resolve time growth in this plot. Its padding to the global maximum list length L_{max} means that the number of ciphertexts per label remains constant and large. As a result, the total number of ciphertexts to decrypt increases rapidly with query area. The occasional jumps in the curve are due to variations in the number of labels covered by individual queries.

Among the constructions analyzed in figure 6.5, the baseline EMM and realDPRFMM stand out due to their closely aligned performance profiles. Despite having different leakage characteristics, both schemes achieve similarly low resolve times, and their curves follow each other closely throughout the full range of query areas.

The baseline EMM does not implement any form of volume hiding. It returns exactly one ciphertext for each matching document, and the client decrypts only these entries. This minimal decryption load results in very low and predictably increasing resolve times as query area expands. The curve for EMM reflects this linear relationship between query coverage and the number of documents matched.

In contrast, realDPRFMM introduces padding to conceal volume leakage but does so using a more practical approach than DPRFMM. Rather than padding to the maximum bucket size B, it pads to a smaller bound P_{real}, chosen based on expected (rather than worst-case) occupancy. This allows it to achieve a degree of volume-hiding while keeping the number of padded ciphertexts per label small and predictable.

What makes realDPRFMM's resolve time nearly match that of EMM is the tight calibration of P_{real} relative to typical list sizes in the dataset. As a result, the average number of ciphertexts per label returned to the client is not much higher than in EMM, especially for queries covering many labels. Since the client must still perform decryption and authenticity checks on these padded entries, some overhead is introduced, but this overhead remains marginal and grows slowly with query size.

As for the AHEMM and VHAHEMM variants, the AHEMM curve is not visible in figure 6.5 because it nearly overlaps with VHAHEMM due to the scale of the other schemes. To highlight this, we present figure 6.6, where only AHEMM and VHAHEMM are shown.

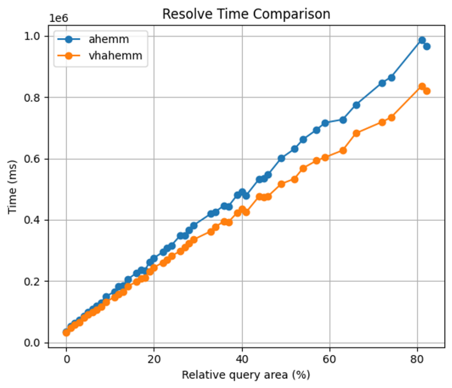


Figure 6.6: Resolve Time Comparison of ahemm and vhahemm on dataset 1

Both AHEMM and VHAHEMM are designed to hide access patterns, ensuring that the server cannot determine which specific entries were returned. However, they differ in their treatment of volume leakage. AHEMM leaks the exact number of matching records, while VHAHEMM introduces volume padding by extending each label's result list to a fixed bound P, derived from the largest bucket in its layout.

In terms of resolve time, AHEMM generally requires slightly more client-side processing than VHAHEMM, as reflected by its consistently higher curve in figure 6.6. This might seem counterintuitive, given that VHAHEMM applies padding while AHEMM does not. However, this difference can be explained by the following factors. First of all, in VHAHEMM, the number of ciphertexts per label is fixed (up to P), allowing the client to precompute the expected ciphertexts to decrypt. In contrast, AHEMM may deal with a more variable set of ciphertexts, potentially introducing slight inefficiencies in memory access patterns, buffer allocation, or multithreaded execution depending on how the client is implemented. In addition, while both schemes rely on symmetric-key decryption, VHAHEMM's uniform ciphertext sizes and layout can lead to better CPU-level cache utilization and batching opportunities during decryption. AHEMM's variable-length results, meanwhile, may slightly increase per-ciphertext overhead. Finally, since VHAHEMM returns a predictable number of ciphertexts per label, it is easier to optimize the client code for performance, e.g. via parallel processing or fixed-size loops. AHEMM's less regular output sizes may prevent such optimization in certain scenarios.

Despite these differences, the performance gap remains small. Both constructions show linear growth in resolve time as the query area increases. This is expected, as larger queries cover more labels and return more ciphertexts. However, VHAHEMM's slightly lower curve suggests that volume padding can be implemented efficiently and does not necessarily imply a significant computational burden at the client.

6.6. Result Count Comparison Dataset 1

Figure 6.7 presents the mean result counts of all EMM variants. In our evaluation, result count refers to the number of encrypted entries returned by the server in response to a given query. This metric provides insight into the volume of data exposed per query and reflects a key dimension of leakage in encrypted search schemes. In constructions without volume-hiding mechanisms, the result count closely approximates the actual number of matching records, potentially revealing query selectivity to the server. In contrast, volume-hiding schemes intentionally obscure this count by introducing dummy values or padding, resulting in uniform or capped output sizes. For the sake of measurement, our reported result count includes all returned entries, whether real or dummy, to reflect the total communication and client-side filtering cost incurred by the scheme.

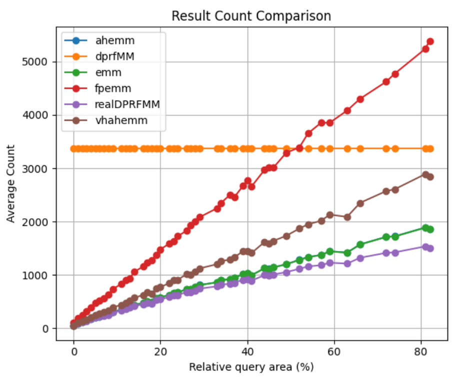


Figure 6.7: Result Count Comparison of all EMM variants on dataset 1

The baseline EMM and the access-hiding variant AHEMM do not apply any padding to the results. As a result, they leak the exact number of matching records for every query. This is reflected in Figure 6.7 by their linear curves, where the result count increases proportionally with the number of documents matched. Since the baseline EMM and AHEMM yield identical result counts, their curves completely overlap, which is why the curve for the AHEMM is not visible in this figure.

In contrast, FPEMM provides full volume-hiding by padding each keyword's posting list to the global maximum list size, denoted by L_{max}. Consequently, the result count for FPEMM grows linearly with query coverage but at a much steeper rate, since each label contributes L_{max} ciphertexts regardless of how many actual matches it has. This results in high communication overhead, especially for queries that cover sparse keywords.

VHAHEMM adopts a more moderate approach by padding each posting list to a fixed bound P, which corresponds to the largest label bucket size in its underlying hash-based construction. This leads to a curve that also grows linearly with query coverage but with a lower slope than FPEMM. The server learns only the upper bound P per label, rather than exact result counts.

DPRFMM goes further by using cuckoo hashing and stash padding to conceal volume leakage. Each queried label contributes a fixed number of ciphertexts derived from two buckets (each of size B) and a stash of size S. This results in a consistent per-label contribution of $2B + S$, making the DPRFMM curve parallel to VHAHEMM's but with a slightly higher slope due to the stash. Importantly, this scheme provides computational volume-hiding, meaning the server cannot distinguish between queries based on their selectivity.

Finally, realDPRFMM offers a tunable trade-off. Instead of padding buckets to the full size B, it pads them to a more practical bound P_{real}, which is based on expected occupancy rather than the worst case. As a result, its curve lies between those of AHEMM and DPRFMM. It leaks approximate volume information by revealing how many entries fall within the typical bound, but not the exact counts. This makes it suitable for scenarios where some leakage is acceptable in exchange for lower bandwidth.

6.7. Trapdoor Time Comparison Dataset 2

To set the stage for the next set of experiments, we briefly highlight the primary distinction between Dataset 1 and Dataset 2. While Dataset 1 was constructed with multiple values per coordinate to simulate a denser, more information-rich setting, Dataset 2 mirrors the structure of the original California dataset, containing only a single value per coordinate.

This sparser format has direct implications for volume leakage. When each coordinate stores only one value, the number of results per query is inherently small and less variable. As a result, volume-based inference attacks pose a lower practical threat in this setting. Because of this reduced leakage risk, several of the volume-hiding mechanisms implemented by the more advanced EMM variants may appear excessive for this data scenario.

It is precisely for this reason that we evaluate our EMM constructions on Dataset 2. By testing in a setting where volume leakage is naturally constrained, we can observe how each construction performs when its defensive mechanisms are arguably overengineered. This allows us to assess whether the additional complexity and performance overheads introduced by volume-hiding techniques are justified in lower-risk contexts, and how each variant compares to the baseline EMM under these new conditions.

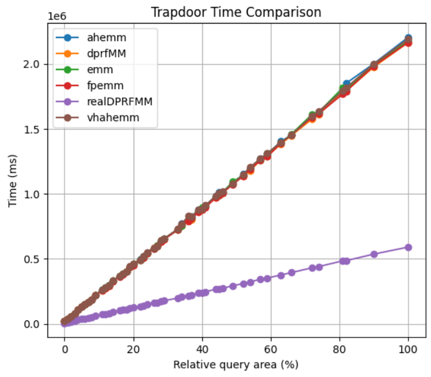


Figure 6.8: Trapdoor Time Comparison of all EMM variants on dataset 2

Figure 6.8 presents the average trapdoor generation time for each EMM variant on Dataset 2. Similar to the results observed on Dataset 1 in figure 6.1, we find that most EMM variants show nearly identical performance, with the one notable exception being realDPRFMM consistently outperforming all others in terms of trapdoor generation time.

This repeated trend across both datasets suggests that the format and density of the underlying data do not significantly influence trapdoor generation performance for most schemes. The process of creating a trapdoor primarily depends on the number of labels derived from a query and the cryptographic primitives used to encode them, rather than on the volume or structure of the underlying data associated with those labels.

The superior performance of realDPRFMM can be attributed to its use of a more efficient key derivation and hashing pipeline tailored to its lightweight padding strategy. Since it avoids the heavier operations required by constructions like FPEMM or DPRFMM, and does not require per-label structure padding during trapdoor generation (as that is handled during retrieval), realDPRFMM is able to generate trapdoors faster.

In contrast, the remaining variants (including EMM, AHEMM, FPEMM, and VHAHEMM) share nearly identical trapdoor generation costs. This is expected, as they all involve similar steps in label extraction and symmetric encryption to some extent. These operations scale with the number of labels per query, which remains relatively constant regardless of the data format.

These findings indicate that trapdoor generation time is largely unaffected by the sparsity of Dataset 2. The similarity in performance across EMM variants reinforces the observation that trapdoor generation is dictated more by scheme design than by data characteristics. RealDPRFMM's faster performance makes it particularly attractive in applications requiring low-latency query issuance, especially when its leakage profile is acceptable for the use case.

6.8. Search Time Comparison Dataset 2

Figure 6.9 illustrates the average search time per query for all EMM variants on Dataset 2. Before analyzing the relative behavior of the schemes, it is important to note the scale of the y-axis: search times are significantly lower overall compared to the corresponding results on Dataset 1. This is expected, as Dataset 2 contains only single-valued tuples per coordinate. With fewer results returned per query, the total amount of data processed is reduced, leading to faster search operations across all variants. Readers interested in comparing raw execution times across both datasets on a consistent scale may refer to Figure B.1 in the Appendix, which reproduces this plot using the same y-axis range as that of the Dataset 1 version. Rather than focusing on absolute time values, our analysis centers on the relative performance trends between variants and how they compare to those observed in Dataset 1.

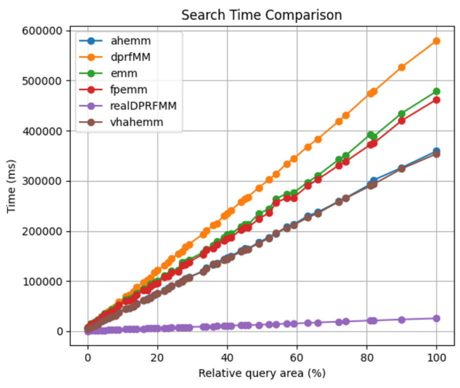


Figure 6.9: Search Time Comparison of all EMM variants on dataset 2

In contrast to Figure 6.2 on Dataset 1, where FPEMM clearly showed the highest search time cost due to its aggressive global padding strategy, it is now DPRFMM that shows the highest search time on Dataset 2. This shift is driven by the data format with only a single value per coordinate. The relative burden of DPRFMM's padding and filtering operations becomes more pronounced. The scheme's fixed-size bucket and stash padding leads to large amounts of dummy data being returned and filtered, regardless of how small the true result set is.

Meanwhile, FPEMM now performs comparably to the baseline EMM. Because every coordinate contains only one entry, the per-label padding enforced by FPEMM is less excessive than in Dataset 1 in Figure 6.2. As a result, the number of dummy ciphertexts returned is closer to the number of actual matches, reducing the overall padding overhead and bringing its performance closer to unpadded schemes.The other EMM variants (AHEMM, realDPRFMM, and VHAHEMM) continue to show behavior consistent with their performance on Dataset 1. AHEMM remains similar to VHAHEMM in search time, while maintains the lowest search time among the volume-hiding schemes due to its efficient design. These results confirm that padding-heavy constructions like DPRFMM have a significant performance cost even when volume leakage is naturally low, whereas schemes like FPEMM adapt better to sparse settings.

6.9. Resolve Time Comparison Dataset 2

Figure 6.10 shows the average resolve time per query for all EMM variants on Dataset 2. As was the case with Dataset 1 in Figure 6.4, DPRFMM continues to show the highest resolve time by a considerable margin. This is expected due to its strong volume-hiding guarantees, which require returning and processing a large number of padded ciphertexts per queried label. Each of these ciphertexts must be decrypted or checked at the client, regardless of whether it contains real data. However, a key difference compared to the previous dataset is that all other EMM variants now demonstrate resolve times that closely match that of the baseline EMM. RealDPRFMM slightly outperforms the baseline.

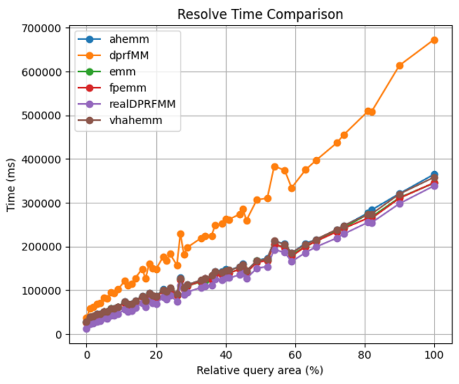


Figure 6.10: Resolve Time Comparison of all EMM variants on dataset 2

This convergence in performance is largely a result of the sparse data format in Dataset 2. Since each coordinate contains only a single value, the number of ciphertexts per label is minimal. As a consequence FPEMM has far less overhead than in Dataset 1, as its global padding bound L_{max} becomes closer to the true result count. VHAHEMM applies per-label padding to a small fixed bound P, which in this setting is not much larger than the actual list size. AHEMM and EMM apply no padding, thus naturally retain their low overhead. RealDPRFMM, with its practical padding bound P_{real}, avoids worst-case overhead while still hiding volume patterns approximately. In this sparse context, the actual number of dummy ciphertexts added is minimal, allowing it to slightly outperform even the baseline EMM due to possible batching or lower decryption overhead per ciphertext.

These results illustrate that when data is inherently sparse and result lists are short, the overhead of most volume-hiding mechanisms becomes negligible. In such cases, the primary driver of resolve time is simply the number of ciphertexts that must be decrypted, which is something that varies little between padded and unpadded schemes under these conditions.

6.10. Result Count Comparison Dataset 2

Figure 6.11 presents the average number of ciphertexts returned per query for each EMM variant on Dataset 2. Compared to the result count plot for Dataset 1 in Figure 6.7, the biggest difference is observed in the behavior of FPEMM. In Dataset 1, FPEMM returned the most ciphertexts due to its global padding strategy, which padded every postings list to the global maximum list size L_{max}. However, on Dataset 2, the FPEMM curve now overlaps completely with those of the baseline EMM and AHEMM. This convergence occurs because, in this dataset, each coordinate holds only a single value. As a result, the global maximum list size L_{max} is effectively identical to the typical list size, eliminating the need for excessive padding. Thus, all three schemes return the same number of results, exposing the exact selectivity of each query.

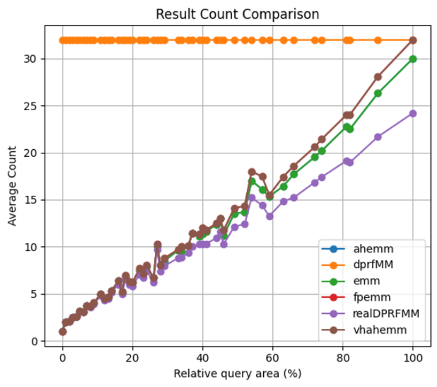


Figure 6.11: Result Count Comparison of all EMM variants on dataset 2

DPRFMM, on the other hand, continues to enforce strict volume-hiding and exhibits the highest, constant result count across all query areas. This is a direct consequence of its use of fixed-size bucket and stash padding per queried label. Even in sparse datasets like this one, DPRFMM still pads each result to the worst-case bound $2B + S$, ensuring uniform ciphertext counts per label and complete volume obfuscation.

VHAHEMM maintains a moderate padding strategy. It pads each postings list to a fixed per-label bound P, which is typically smaller than L_{max} but larger than the actual list size in sparse settings. As a result, its result count is higher than EMM, AHEMM, and FPEMM, but lower than DPRFMM. Its curve increases more steeply with query area since it returns P ciphertexts for every label covered. RealDPRFMM pads up to a practical bound P_{real}, chosen based on typical occupancy rather than the worst case. Consequently, its result count curve lies below that of VHAHEMM and significantly below DPRFMM.

6.11. Quantitative Feature Analysis

While visual plots of latency and result curves provide intuitive insight into the behavior of each EMM scheme, relying solely on graphical interpretation can be misleading, especially when differences between schemes are subtle or distributed across multiple metrics. To provide a more rigorous and reproducible basis for comparison, we extracted five quantitative features from the performance curves of each scheme: slope, intercept, coefficient of determination (R^2), end-point value, and normalized area under the curve (AUC). These features offer interpretable summaries of each curve's behavior and allow for cross-scheme evaluation.

Quantitative Feature Definitions:

1. **Slope (ms/%):** The slope represents the incremental cost in milliseconds per percent increase in query coverage. It captures the efficiency of a scheme as query area grows. A lower slope indicates that the scheme scales better with query size.
2. **Intercept (ms):** This value represents the fixed overhead of the scheme when query coverage is near zero. It may reflect setup costs, token generation time, or encryption-related overheads. Lower intercepts are preferable for applications involving many small queries.
3. **R^2 (Coefficient of Determination):** R^2 measures how well a linear model fits the performance curve. Values near 1.0 confirm that the behavior is nearly linear, validating the appropriateness of linear summaries. Deviations suggest non-linearities or inconsistencies in scaling.
4. **End-Point (ms):** This is the total time required for the largest query size evaluated (typically the maximum coverage). It gives a sense of worst-case performance and practical latency bounds.
5. **Normalized AUC (ms·%):** The normalized area under the curve integrates performance across all coverage levels and divides by the coverage range. It approximates the average latency per unit coverage and is useful for comparing overall efficiency.

In this work, we deliberately focus on average features rather than features derived from the combined total time curve. While summing trapdoor, search, and resolve times into a single curve can yield a practical latency measure, it can also obscure the behavior of individual components. For instance, trapdoor time often dominates the total due to its scale, overshadowing the contributions and optimizations in the search or resolve phases. By averaging features computed separately on each sub-operation, we can ensure that each component is given equal analytical weight, regardless of magnitude. It also avoids biases introduced by the disproportionately larger values in one metric overwhelming the others in a combined curve.

Table 6.1: Average Curve Features for Each EMM construction on Dataset 1

Construction	Slope (ms/%)	Intercept (ms)	R^2	End-Point (ms)	Norm. AUC (ms·%)
EMM	0.1668	0.2982	0.9975	13.6186	7.1550
AHEMM	0.0127	0.0183	0.9953	1.0796	0.5386
VHAHEMM	0.0121	0.0259	0.9959	0.9861	0.5207
FPEMM	0.2801	0.8047	0.9974	23.7477	12.2607
DPRFMM	1.2083	3.1622	0.9969	99.7598	52.7668
RealDPRFMM	0.1146	0.2593	0.9969	9.5518	4.9642

Table 6.2: Average Curve Features for Each EMM construction on Dataset 2

Construction	Slope (ms/%)	Intercept (ms)	R^2	End-Point (ms)	Norm. AUC (ms·%)
EMM	0.0099	0.0183	0.9889	1.0065	0.5168
AHEMM	0.0095	0.0178	0.9943	0.9652	0.4927
VHAHEMM	0.0095	0.0178	0.9940	0.9755	0.4952
FPEMM	0.0100	0.0180	0.9909	1.0413	0.5188
DPRFMM	0.0115	0.0216	0.9933	1.1554	0.5976
RealDPRFMM	0.0032	0.0047	0.9744	0.3870	0.1662

Looking at tables 6.1 and 6.2 allows us to make several observations based on the quantitative features. First of all, for the slope on Dataset 1, the slopes vary widely: DPRFMM (1.2083 ms/%) scales worst, followed by FPEMM at 0.2801 ms/%. realDPRFMM (0.1146 ms/%), EMM (0.1668 ms/%), AHEMM (0.0127 ms/%), and VHAHEMM (0.0121 ms/%) lie in between. This ordering aligns with our qualitative analysis that padding-heavy schemes have a larger incremental costs. On Dataset 2, slopes converge around 0.01 ms/%, except for realDPRFMM, which drops to 0.0032 ms/%. The sparse data format neutralizes most padding overhead, leaving only the minimal per-label cost. RealDPRFMM's tiny slope reflects its highly optimized padding bound relative to actual list sizes.

Dataset 1 intercepts range from $\sim 0.02ms$ for AHEMM/VHAHEMM to 3.16 ms for DPRFMM. This indicates substantial setup or per-query initialization costs for DPRFMM, likely due to key derivation or padding metadata preparation. Dataset 2 intercepts drop sharply (0.017–0.022 ms for most schemes), confirming that fixed costs are dominated by data-independent operations (e.g., trapdoor token setup), which remain constant across datasets. realDPRFMM's intercept (0.0047 ms) is especially low, benefiting low-latency small-query scenarios.

All variants achieve $R^2 > 0.99$ on Dataset 1, validating the linear model assumption. On Dataset 2, R^2 remains high (> 0.99) for all but realDPRFMM (0.9744), suggesting minor non-linear effects in its padding strategy when lists are extremely small.

The end-point on Dataset 1 spans from $\sim 0.99ms$ (VHAHEMM) to $\sim 99.76ms$ (DPRFMM). This feature illustrates contrast between lightweight schemes, which complete even the most expansive queries in under one millisecond, and DPRFMM, which has nearly 100× higher latency due to its extensive padding and decryption workload. On Dataset 2, end-points compress to ~ 0.4–$1.15ms$, further reinforcing that all schemes become practical for latency-sensitive applications in sparse settings, with DPRFMM still at the high end. The dramatic compression of worst-case latency in this sparse setting highlights that when actual result lists are small, even heavy padding schemes become feasible for latency-sensitive tasks. This suggests that, in practice, choosing a volume-hiding construction can be guided by the acceptable upper bound on query latency and the expected query size distribution.

On Dataset 1, normalized AUC values span from around 0.52 ms·% (VHAHEMM) to 52.77 ms·% (DPRFMM). VHAHEMM's low AUC indicates that it combines minimal fixed cost with little scaling, making it highly efficient across varied query sizes. In contrast, DPRFMM's large AUC highlights that its average per-unit cost remains high owing to heavy padding. On Dataset 2, normalized AUC compresses into a tighter band of approximately 0.17 ms·% (realDPRFMM) to 0.60 ms·% (DPRFMM). This area metric shows that, even when worst-case latency diverges, the cumulative work done across small-to-large queries remains bounded and low for most schemes. realDPRFMM's exceptionally low normalized AUC emphasizes its consistent, low-latency behavior, making it ideal for workloads with diverse query sizes.

To summarize:

- Data sparsity dramatically reduces both fixed and incremental costs for all schemes, making even padding-heavy constructions feasible in low-leakage settings.
- DPRFMM, while providing the strongest volume-hiding guarantee, remains the most expensive in both metrics, suggesting its use only when leakage risk is otherwise unacceptable.
- realDPRFMM offers the best performance, with moderate overhead on dense data and minimal overhead on sparse data, thanks to its pragmatic padding bound.
- VHAHEMM offers near-baseline efficiency on both datasets while providing coarse-grained volume protection.
- AHEMM and the baseline EMM are indistinguishable in sparse settings, revealing that access-pattern hiding alone adds negligible cost.

7 Verification and Validation

In this chapter, we present a series of empirical tests showing the correctness of our schemes, alongside a formal security proof that verifies their theoretical soundness. We also include a complexity analysis to validate their practical efficiency.

7.1. Correctness Testing

To demonstrate the correctness and reliability of our implementations, a comprehensive suite of unit tests was developed and executed using the `pytest` framework. These tests validate that the system performs as expected under both typical and edge-case conditions, while also reflecting its formal security goals: namely, hiding the access pattern while allowing volume leakage under an honest-but-curious threat model.

Our ”Round-Trip Tests” verify that the encrypted multi-map correctly retrieves the original data values associated with a given label. This is done by building an index from a known plaintext map; computing a trapdoor for each label and retrieving the corresponding encrypted values; decrypting the ciphertexts back into plaintext; comparing the output against the original values. This confirms that the `build_index`, `trapdoor`, `search`, and `resolve` functions form a coherent pipeline that reliably preserves the intended mapping.

The next tests were to ensure internal consistency we showed that generating trapdoors from the same key-label pair are identical across runs. Additionally, the results of multiple searches using the same trapdoor are shown to return identical ciphertext sets as expected. This proves that the system is deterministic with respect to key and label, as required for repeatable and cacheable trapdoor-based access.

The scheme’s reliance on cryptographic keys was tested by comparing two independently initialized EMM engines where identical plaintext data was encrypted using two different keys, which resulted in the trapdoors and ciphertexts for the same label to differ completely. This behavior validates the cryptographic binding of the trapdoor and ciphertexts to the secret key, as to be expected from a secure HMAC and encryption scheme.

In case that the EMM construction leaked the volume pattern, tests were run to explicitly verify this property. Labels with different numbers of associated values were indexed. For each query, the number of ciphertexts retrieved matched the number of inserted values. This demonstrates that while the system hides which label was queried, it leaks the number of matching values.

The following non-standard cases were tested to ensure implementation robustness: Labels with empty value lists (no ciphertexts should be returned); empty string labels (which should still function correctly); very long strings (to check serialization and encryption integrity); unicode and non-ASCII values (to confirm proper character encoding).

Finally, security was further validated by testing behavior under key mismatch, where ciphertexts encrypted under one key were presented to a different EMM engine. In this case, the resolve either failed or produced incorrect data. This confirms that ciphertexts cannot be meaningfully decrypted without the correct key, upholding confidentiality guarantees.

7.2. Formal Security Proof

In this section we will provide a simulation-based security proof for AHEMM. We used the formal SSE definitions of Curtmola et al. [40] and others [41] and account for volume-pattern leakage as discussed in recent Structured Encryption (STE) literature [42, 43, 44].

7.2.1. Scheme Syntax and Real Worlds

We model AHEMM as a structured encryption scheme $\Sigma = (Setup, Query, Eval, Result)$ for a multi-map MM. Formally, $Setup(1^\lambda, MM)$ is a Probabilistic Polynomial-Time (PPT) algorithm [45, 46] that takes a security parameter and the plaintext multi-map MM and outputs a secret key K and an encrypted multi-map EMM. $Query(K, l)$ (client-side) takes the key K and a label/query $l \in L$ and produces a query token τ (this corresponds to the "Token" algorithm in standard STE). $Eval(EMM, \tau)$ (server-side) takes the encrypted structure EMM and token τ and returns an encrypted response c (analogous to "Query" in STE). Finally, $Result(K, c)$ (client-side) takes K and the ciphertext c and outputs the result $r = MM(l)$. Correctness requires that, for any sequence of queries $l_1, ..., l_q$, the decrypted results satisfy $r_i \equiv MM(l_i)$.

We describe the real-world protocol from the (semi-honest) server's perspective. Intuitively, the server sees the encrypted multi-map and all query tokens and encrypted answers. Formally:

1. The adversary (server) chooses a multi-map MM and gives it to the challenger.
2. The challenger runs $(K, EMM) \leftarrow Setup(1^\lambda, MM)$ and sends EMM to the adversary.
3. For each client query l_i (chosen adaptively by the adversary or environment), the challenger computes $\tau_i \leftarrow Query(K, l_i)$ and $c_i \leftarrow Eval(EMM, \tau_i)$, then sends (τ_i, c_i) to the adversary
4. The adversary outputs a bit indicating whether it believes this is the real or ideal execution.

In the ideal-world simulation, a PPT simulator $\mathcal{S}$ has access only to specified leakage. Let L_S and L_Q be the setup- and query-leakage functions (defined in subsection 7.2.2). $Ideal_L^{\Sigma,\mathcal{S}}(1^\lambda)$ proceeds as follows:

- The adversary chooses MM and sends it to the challenger. The challenger computes the setup leakage $\ell_0 = L_S(MM)$ and gives ℓ_0 to $\mathcal{S}$. Using only ℓ_0, the simulator outputs a simulated encrypted structure EMM' and gives EMM' to the adversary.
- For each client query l_i (chosen by the adversary), the challenger computes the query leakage $\ell_i = L_Q(MM, (l_1, ..., l_i))$ and gives ℓ_i to $\mathcal{S}$. The simulator then outputs a token τ_i' and a ciphertextc_i' and gives (τ_i', c_i') to the adversary.
- Finally the adversary outputs a bit.

This matches the standard simulation-based SSE model in which the adversary (playing the role of the honest-but-curious server) cannot distinguish whether it is interacting with the real protocol or with the simulator $\mathcal{S}$ that only sees leakage.

7.2.2. Leakage Profile

Because AHEMM hides which records are accessed, the only nontrivial leakage is the *volume pattern*. Concretely, we define:

- **Setup leakage** $L_S(MM)$**:** the adversary learns the database "dimensions" (e.g. number of labels/values). At minimum we let L_S reveal the total number of labels m, the total number of values $N = |MM|$, and an upper bound on the maximum tuple size (so that EMM' can be appropriately sized). This is analogous to the usual $(N, t)-$leakage in SSE.
- **Query leakage** $L_Q(MM, (l_1, ..., l_i))$**:** for each query l_i, L_Q reveals only the *volume* $|MM(l_i)|$ of the response. In particular, no information about which label is queried or which values are returned is leaked. Thus $L_{Q_i} = |MM(l_i)|$, and the adversary only learns the sequence of lengths of responses.

In summary, the leakage profile $L = (L_S, L_Q)$ for AHEMM reveals only the size parameters of the database and the lengths of each query's result (*the volume pattern*). In particular the query/equality/search pattern and the access pattern (which values are accessed) remain hidden. This is consistent with AHEMM's goal of access-pattern hiding: the server sees *how many* items are returned, but not which ones or whether a query repeats.

7.2.3. Security Definition (Simulation-Based)

We say AHEMM is $L-$secure (in the adaptive setting) if no PPT adversary can distinguish the real and ideal experiments except with negligible advantage. Formally, let $Real_{\Sigma}^{A}(1^{\lambda})$ and $Ideal_{L}^{A,\mathcal{S}}(1^{\lambda})$ be the two experiments defined above. Then Σ is $L-$secure if there exists a PPT simulator $\mathcal{S}$ such that for every PPT adversary A, $Prob[A$ outputs 1 in $Real_{\Sigma}^{A}]$ - $Prob[A$ outputs 1 in $Ideal_{L}^{A,\mathcal{S}} = negl(\lambda)$. Equivalently, the view of A in the real world (which consists of the actual encrypted multi-map EMM and the real tokens/ciphertexts) is computationally indistinguishable from the view simulated by $\mathcal{S}$ given only the leakage L.

7.2.4. Simulator Construction

We now describe a PPT simulator $\mathcal{S}$ that, on input the leakage $L = (L_S, L_Q)$ (and the public parameters), produces a view that is indistinguishable from the real protocol transcript. Let MM be the adversary's chosen database, $\mathcal{S}$ only sees $L_S(MM)$ and then L_Q for each query. The simulator works as follows:

- **Setup simulation:** $\mathcal{S}$ receives $\ell_0 = L_S(MM)$, which includes the total number of labels m and total values N (and max tuple size). Based on this, $\mathcal{S}$ constructs a dummy encrypted multi-map EMM' of the same "shape" as a real EMM. For example, if the real scheme uses an encrypted dictionary of size N, $\mathcal{S}$ can simulate this by generating N ciphertext slots filled with random bits. Concretely, $\mathcal{S}$ generates random strings (of the same length as real ciphertexts) to populate EMM' so that $|EMM'|$ matches L_S. By the IND-CPA security of the encryption scheme [46, 47] (and PRF security of any token generation), these random ciphertexts are indistinguishable from encryption of real data. The simulator gives EMM' to the adversary.
- **Query simulation:** For each query i: $\mathcal{S}$ receives $\ell_i = |MM(l_i)|$ (the volume v_i). First, $\mathcal{S}$ generates a random token τ_i' (uniformly from the token space) to simulate $Query(K, l_i)$.Because $Query$ tokens in the real scheme are pseudorandom and independent of l_i (access hiding), the adversary cannot distinguish τ_i' from a real token. Next, to simulate the encrypted response, $\mathcal{S}$ produces v_i "dummy" ciphertexts. For each of the v_i values, $\mathcal{S}$ outputs a random string of the appropriate ciphertext length. These serve as the fake encrypted values that would decrypt to the actual result. The server (adversary) learns that $|MM(l_i)| = v_i$ values were returned, but sees only random-looking ciphertexts. In effect, $\mathcal{S}$ outputs (τ_i', c_i') with c_i' consisting of v_i random ciphertexts. This completes the simulation for query i.

Throughout, $\mathcal{S}$ maintains no additional state except ℓ_0 and the leaked volumes. By construction, the distribution of $(EMM', (\tau_i', c_i')_i)$ produced by $\mathcal{S}$ depends only on the leaked sizes and not on the actual data. Importantly, since the access pattern is hidden, there is no additional linkage between queries that $\mathcal{S}$ must enforce beyond consistent lengths.

7.2.5. Indistinguishability Proof

Theorem (Security of AHEMM). Under the standard cryptographic assumptions that the encryption scheme is IND-CPA secure and the query token function is a secure PRF, the AHEMM scheme is L-secure (with leakage functions $L = (L_S, L_Q)$ as defined above). Equivalently, for any PPT adversary $\mathcal{A}$; the real-world execution Real$\mathcal{A}, \Sigma(1^{\lambda})$; and the ideal execution Ideal$^{L}\mathcal{A}, \mathcal{S}(1^{\lambda})$ are computationally indistinguishable.

Proof. Consider an arbitrary PPT adversary $\mathcal{A}$. We compare the view of $\mathcal{A}$ in the real execution of the AHEMM protocol with its view in the simulated (ideal) execution produced by the simulator $\mathcal{S}$ described above. We show that each component of $\mathcal{A}$'s real-world view is indistinguishable from the corresponding component of the simulated view:

1. **Encrypted multi-map (Setup):** In the real protocol, the server holds an encrypted multi-map EMM that is an encryption of the true database under the secret key K. In the simulated view, $\mathcal{S}$ generates a "dummy" encrypted multi-map EMM$'$ of the same size, filling each entry with a random ciphertext. By the IND-CPA security of the encryption scheme, the distribution of EMM$'$ (random ciphertexts) is computationally indistinguishable from the distribution of EMM (encryptions of the real data). Therefore, $\mathcal{A}$ cannot distinguish whether it has received the real EMM or the simulated EMM$'$.
2. **Query tokens:** Each real query token τ_i = Query(K, ℓ_i) is generated by a keyed pseudorandom function on the query label ℓ_i. In simulation, $\mathcal{S}$ simply chooses each token τ_i' uniformly at random from the token space. Since the PRF-based token generator is secure, τ_i is indistinguishable from

a uniform random string. Hence the tuple of real tokens $(\tau_1, \dots, \tau_t)$ is indistinguishable from the simulated tokens $(\tau'_1, \dots, \tau'_t)$. In other words, $\mathcal{A}$ cannot tell whether the tokens came from the real PRF or were chosen at random.

3. **Encrypted responses:** In the real execution, each query for label ℓ_i returns a ciphertext response c_i consisting of $v_i = |\mathsf{MM}(\ell_i)|$ ciphertexts, each encrypting an actual matching value. In the ideal execution, $\mathcal{S}$ outputs for each query the same number v_i of ciphertext slots, each filled with an independent random string. By IND-CPA security, a batch of fresh random ciphertexts is indistinguishable from encryptions of actual messages. Thus the combination of v_i ciphertexts in the real response is computationally indistinguishable from the combination of v_i random ciphertexts in the simulated response.
4. **No additional leakage:** By construction, the simulator leaks only the specified leakage: the database size (number of labels m and total values N) and each query's volume v_i. The actual values of the labels and the exact access pattern remain hidden. Since AHEMM hides the access pattern, $\mathcal{A}$ cannot correlate which specific entries are returned across queries beyond what is revealed by the volumes v_i. Apart from the components above, the real and simulated views reveal exactly the same information (that information being the leakage defined in Section 7.2.2).

Since each part of the real view is computationally indistinguishable from the corresponding part of the simulated view, the overall views are indistinguishable. Formally, if there existed a PPT distinguisher $\mathcal{D}$ that tells apart the real and ideal executions with non-negligible advantage, we could construct a reduction to break either the IND-CPA security or the PRF security. For example, one could replace the real encrypted multi-map or response ciphertexts one-by-one with random ciphertexts; if $\mathcal{D}$ notices any change, it would break IND-CPA. Similarly, replacing real tokens with random strings would break the PRF security if distinguishable. Each such replacement is indistinguishable by assumption, so no distinguisher can have more than negligible advantage.

Therefore, by a hybrid argument and cryptographic reductions, no PPT adversary can distinguish the real and ideal experiments except with negligible probability. This implies that Σ (the AHEMM scheme) satisfies the L-security definition given above. In other words, all information learned by the server in the real protocol is limited to the prescribed leakage profile (L_S, L_Q), and AHEMM is secure in the simulation-based sense.

7.2.6. Security Proof of other EMM variants

Each EMM variant re-uses the same simulation-based proof skeleton we used for AHEMM, but differs only in what the simulator must hide (the leakage functions) and how it fills in dummy data or random tokens to match each scheme's structure. For VHAHEMM, the simulator only learns the public parameters (total database size and fixed pad length), and on every query returns exactly that fixed number of ciphertexts, mixed with real data (as counted by the setup leakage) and dummies.

For dprfMM and realDPRFMM, the leakage function tells the simulator which past queries were for the same key but never how many items each returned. The core difference is that, in Setup, the simulator must build and randomly fill a cuckoo-hash table of the correct size (as prescribed by the setup leakage), inserting real-value slots and dummy slots so that every key's bucket is the same "shape". On each query, knowing only the equality class and the fixed pad length, it pulls exactly that many entries from the simulated buckets. The rest of the proof follows the same IND-CPA/PRF hybrids: replacing real PRF outputs with random tokens and real encryptions with dummy ciphertexts remains indistinguishable.

Finally, FPEMM sits between VHAHEMM and dprfMM in complexity. The simulator does know the identity of each queried key (because equality is leaked), so it arranges a simple table or array of fixed-length slots per key in Setup, populates with the correct number of "real" ciphertexts and dummies, and on each query releases that entire slot. From the adversary's view, the only difference from AHEMM's proof is that every reply is padded to the same length, so the simulator's dummy-filling mechanism is identical, but keyed to per-label arrays rather than to multi-range covers.

In all cases the proof structure remains unchanged, i.e. defining (L_S, L_Q), building a simulator that produces random tokens and dummy encryptions, and reducing any distinguisher to breaking IND-CPA or PRF security. Only the leakage functions and the dummy-filling strategy differ:

1. VHAHEMM must pad every response to a global constant, hiding both which and how many items leaked.

2. dprfMM/realDPRFMM must simulate a cuckoo hash table of the correct size and reveal buckets for each equality class.
3. FPEMM pads per-key arrays of fixed length, leaking which key but not how many real entries.

7.3. Complexity Analysis

We analyze each scheme in terms of server storage (number of 16-byte ciphertext blocks stored) and query cost (blocks fetched for a key with n_k values). Let K = number of keys, N = total number of values, and t = maximum values for any key (i.e. $t =$ max kn_k). (All ciphertexts are 16-byte blocks.)

- **AHEMM (no padding):** No dummy padding is used, so the server stores exactly one ciphertext per value. Storage = N blocks. A query for a key with n_k values returns exactly those n_k ciphertexts (cost $\Theta(n_k)$). Thus AHEMM has storage $\Theta(N)$ and query cost $\Theta(n_k)$. (It leaks the volume n_k by design.)
- **FPEMM (fixed-length padding):** This scheme pads each key's value list to the global maximum length t. Every key has t entries (including real + dummy values) of fixed size. Thus storage = $K \times t$ blocks, and every query must fetch t blocks. In big-O terms, storage is $\Theta(K, t)$ and query cost is $\Theta(t)$. This completely hides volume, at the cost of a potentially large padding overhead.
- **VHAHEMM (volume-hiding AHEMM):** Like FPEMM, it uses fixed-length padding to hide volumes. We assume it also pads each key's list to length t. So the server stores $K \cdot t$ blocks and each query retrieves t blocks (identical complexity to FPEMM). In other words, storage $\Theta(K, t)$ and query $\Theta(t)$ (independent of n_k). Any additional "access-pattern hiding" overhead (e.g. ORAM) would only increase cost, but the block counts remain as above.
- **dprfMM (hash-based, volume-hiding):** This scheme uses a two-bucket hashing approach with fixed bucket size. Because the number of hash functions (2) and bucket size are constant, each value is stored once (plus a constant number of dummy slots). Server storage is $\Theta(N)$ up to a constant factor (asymptotically optimal). To answer a query for a key with n_k values, the client fetches at most n_k real values plus the two fixed-size buckets (a constant extra). Thus query cost is $\Theta(n_k)$ with only constant overhead. Patel et al. [29] prove that dprfMM achieves optimal storage and query $\Theta(volume)$.
- **realDPRFMM (practical DPRF):** This variant also uses two fixed buckets per key. Its storage and query costs are essentially the same as dprfMM's: $\Theta(N)$ storage and $\Theta(n_k)$ query (with constant extra from the two buckets). In effect, both dprfMM and realDPRFMM touch at most two small buckets (of constant size) per query, so their overhead beyond the actual volume is constant.

Each conclusion above follows directly from how each scheme pads (or not) and the constant-bucket hashing assumption. In particular, padding to length t yields $O(Kt)$ storage and $O(t)$ per-query cost, whereas hashing into two fixed-size buckets yields $O(N)$ storage and $O(n_k)$ cost.

Scheme	Server Storage (ciphertext blocks)	Query Cost (blocks fetched)	Notes
AHEMM	N (one block per actual value)	n_k (all real values)	No padding (leaks volume)
FPEMM	$K \times t$	t	Pad to *t* values each
VHAHEMM	$K \times t$	t	Pad to *t* (hides volume)
dprfMM	$\Theta(N)$ (≈ one block per value)	$\Theta(n_k)$ (plus constant)	2-bucket hashing
realDPRFMM	$\Theta(N)$	$\Theta(n_k)$ (plus constant)	Similar to dprfMM

Table 7.1: Complexity analysis of encrypted multi-map schemes.

To derive the values in Table 7.1 we used the padding and hashing assumptions that were given. Without padding (AHEMM), storage is just the number of values and queries return exactly the real values. With fixed padding (FPEMM/VHAHEMM), each key is padded to length t, so storage = $K \times t$ and each query fetches t blocks (as in naive pad-to-max [48, 49]). For the hash-based schemes (dprfMM/realDPRFMM), fixing 2 locations per key means each query touches ≤2 constant-size buckets. Thus storage is linear in N and query cost is linear in the number of real values n_k (optimal as shown by Patel et al. [29]). These formulas directly follow from counting how many 16-byte ciphertexts are stored and retrieved under each scheme's rules.

8 Discussion

In this chapter, we summarize our key findings, compare the results prior literature, evaluate their theoretical and practical significance, and address the study's limitations. We also reflect on how the research evolved and what lessons were learned along the way.

8.1. Key Findings

Based on the extensive evaluation presented in Chapter 6, this section summarizes the key experimental findings that emerged from benchmarking the performance and leakage profiles of the various EMM constructions. These results were derived across multiple metric (trapdoor generation, search time, resolve time, and bandwidth) on both dense (multi-valued) and sparse (single-valued) datasets. The following points highlight the most significant insights.

The realDPRFMM construction achieved the best performance in trapdoor generation, significantly outperforming other schemes by re-using the derived $HMAC$ key across queries. This optimization avoids redundant key derivation and reduces latency. Other schemes (EMM, AHEMM, VHAHEMM, FPEMM, and DPRFMM) showed nearly identical trapdoor timings due to the shared $KDF + HMAC$ structure, indicating that their cryptographic token generation costs are dominated by this common computation.

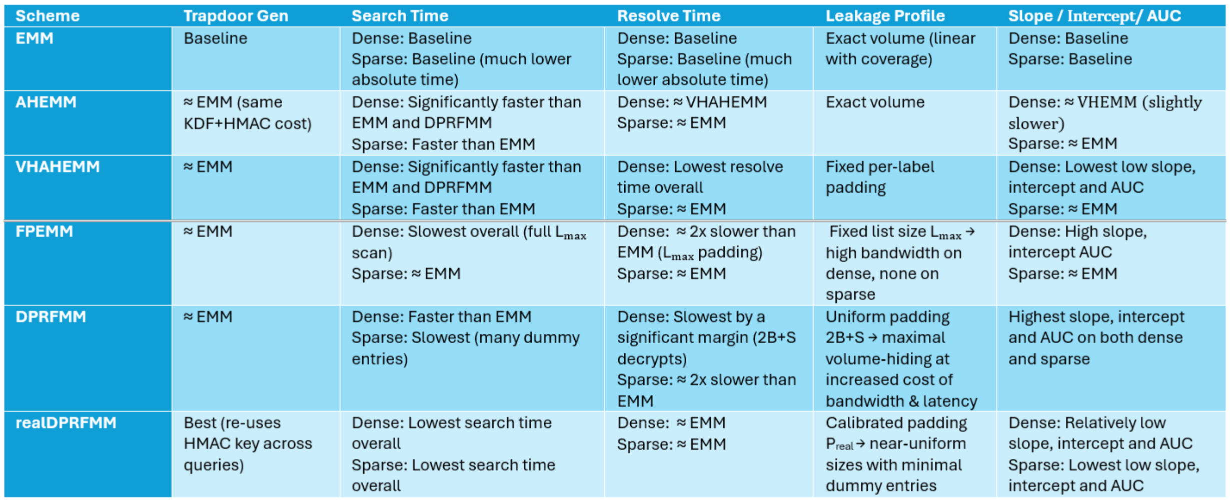
On the dense Dataset 1, the baseline EMM and FPEMM variants showed the highest search times. FPEMM's fixed global padding forces the server to scan a larger padded database without optimizations, making it consistently slower than EMM. By contrast, AHEMM, VHAHEMM, DPRFMM, and realDPRFMM all outperformed the baseline. realDPRFMM achieved the lowest search time overall, reflecting its design focus on efficiency, while AHEMM and VHAHEMM ran at nearly identical speeds despite VHAHEMM's extra per-label padding. Interestingly, DPRFMM, even though it explores a larger cuckoo-hash search space and applies padding to both buckets and query responses, still beat EMM and FPEMM thanks to its use of delegatable PRFs. This gain comes with additional search pattern leakage, which is outside our current analysis. On the sparse Dataset 2, all schemes benefited from single-valued data and showed much lower absolute times (see Appendix B.1 for a unified scale). Here, DPRFMM became the slowest variant because its fixed bucket and stash padding returns many dummy entries even when only one real result exists. Meanwhile, FPEMM's per-label padding matched the single-value format closely, making its performance comparable to EMM. AHEMM and VHAHEMM remained similar to one another, and realDPRFMM continued to lead among volume-hiding schemes, confirming that constructions with heavy padding suffer most in sparse settings while adaptive designs adapt better.

On the dense Dataset 1, schemes with heavier padding (DPRFMM, FPEMM) suffered the most in client-side resolve time. DPRFMM's worst case resolve time approached 100 ms due to the requirement to decrypt $2B + S$ ciphertexts per label. FPEMM also experienced significant decryption time as every label was padded to the dataset-wide maximum list length L_{max}. On the sparse Dataset 2, resolve times for most schemes (including EMM, AHEMM, VHAHEMM, FPEMM, and realDPRFMM) were smaller, as minimal or no padding was needed. DPRFMM remained the slowest due to its fixed padding.

EMM and AHEMM leaked the volume pattern, with result counts growing linearly with query coverage. FPEMM enforced a fixed list size L_{max} for every label, resulting in significant bandwidth overhead on dense datasets but negligible impact on sparse datasets. VHAHEMM padded each label to a fixed per-label bound, striking a balance between leakage and efficiency. DPRFMM enforced strict uniform padding $2B + S$ per label, ensuring maximal volume-hiding at the cost of increased bandwidth and client latency. realDPRFMM achieved near-uniform response sizes with minimal dummy entries by padding to realistic bounds.

Slope, intercept, R², end-points, and normalised area-under-curve (AUC) analyses supported our qualitative findings. Padding-heavy schemes (DPRFMM, FPEMM) showed high slopes and AUCs on dense datasets, showing a steep performance decline. AHEMM and VHAHEMM maintained slope and intercept values nearly identical to the baseline EMM, indicating minimal additional performance cost. realDPRFMM achieved the lowest slope and intercept values among volume-hiding schemes, which showed its efficiency across both datasets.

Table 8.1: Performance and leakage comparison of EMM constructions on dense and sparse datasets.



Scheme	Trapdoor Gen	Search Time	Resolve Time	Leakage Profile	Slope / Intercept/ AUC
EMM	Baseline	Dense: Baseline Sparse: Baseline (much lower absolute time)	Dense: Baseline Sparse: Baseline (much lower absolute time)	Exact volume (linear with coverage)	Dense: Baseline Sparse: Baseline
AHEMM	≈ EMM (same KDF+HMAC cost)	Dense: Significantly faster than EMM and DPRFMM Sparse: Faster than EMM	Dense: ≈ VHAHEMM Sparse: ≈ EMM	Exact volume	Dense: ≈ VHEMM (slightly slower) Sparse: ≈ EMM
VHAHEMM	≈ EMM	Dense: Significantly faster than EMM and DPRFMM Sparse: Faster than EMM	Dense: Lowest resolve time overall Sparse: ≈ EMM	Fixed per-label padding	Dense: Lowest low slope, intercept and AUC Sparse: ≈ EMM
FPEMM	≈ EMM	Dense: Slowest overall (full L_{max} scan) Sparse: ≈ EMM	Dense: ≈ 2x slower than EMM (L_{max} padding) Sparse: ≈ EMM	Fixed list size L_{max} → high bandwidth on dense, none on sparse	Dense: High slope, intercept AUC Sparse: ≈ EMM
DPRFMM	≈ EMM	Dense: Faster than EMM Sparse: Slowest (many dummy entries)	Dense: Slowest by a significant margin (2B+S decrypts) Sparse: ≈ 2x slower than EMM	Uniform padding 2B+S → maximal volume-hiding at increased cost of bandwidth & latency	Highest slope, intercept and AUC on both dense and sparse
realDPRFMM	Best (re-uses HMAC key across queries)	Dense: Lowest search time overall Sparse: Lowest search time overall	Dense: ≈ EMM Sparse: ≈ EMM	Calibrated padding P_{real} → near-uniform sizes with minimal dummy entries	Dense: Relatively low slope, intercept and AUC Sparse: Lowest low slope, intercept and AUC

Notes:

- "≈ EMM" indicates performance very close to the baseline EMM.
- "Dense" refers to Dataset 1; "Sparse" refers to Dataset 2.
- Dense dataset timings are worst-case values from Chapter 6; sparse timings reflect minimal padding scenarios.

Table 8.1 summarizes our key findings by evaluating each EMM construction compared to the baseline.

8.2. Comparison with Prior Work

Prior multi-attribute encrypted range search schemes have typically offered either fixed trade-offs between performance and leakage or targeted protection for only one form of leakage, usually volume, along a single dimension. For instance, the EMM proposed by Falzon et al. [1] supports multi-dimensional range queries using canonical range-cover labels, but leaks volume patterns and it leaks more access pattern compared to our constructions. Similarly, Patel et al.'s dprfMM [29] achieves strong volume hiding for single-attribute multimap search by employing cuckoo hashing and distributed PRFs, but incurs high resolve costs due to worst-case padding. Kamara and Moataz's VLH and AVLH constructions [28] instead rely on computational assumptions to achieve tunable volume hiding, but their schemes are also limited to single-dimensional queries. Finally, Markatou et al. [26] showed how structural and volume leakage in multi-dimensional settings can be exploited for data reconstruction, further highlighting the importance of managing both access and volume leakage.

Our work extends and generalizes these prior contributions by introducing a tunable framework that supports secure multi-dimensional range search with control over both volume and access leakage. We implement five scheme variants each offering a distinct point on the leakage-performance spectrum. Our DPRFMM construction mirrors the strong volume-hiding guarantees of Patel et al.'s dprfMM [29], achieving constant-size responses through full padding. However, consistent with their findings, we observe that this incurs significant latency: approaching 100 ms per label in our multi-dimensional context.

To address this, our realDPRFMM variant introduces a padding bound P_{real}, significantly reducing the amount of dummy data and hence lowering average query latency by an order of magnitude. While this entails a controlled amount of volume leakage, it mirrors the tunability goal by Kamara and Moataz [28], now extended to support multiple attributes. Meanwhile, our access-hiding schemes (AHEMM and VHAHEMM) use tuple-level encryption to suppress per-item access leakage without the severe overheads of ORAM, achieving latency costs of approximately 0.01 ms per tuple, a negligible increase over the non-access-hiding variants.

The novelty of our contribution lies in providing a suite of five EMM variants that all support encrypted multi-dimensional range queries, yet each variant features a different combination of volume and access leakage controls and performance characteristics. This approach enables practitioners to select the variant that best aligns with their security and efficiency requirements. No prior work offers multiple co-existing schemes achieving the same functionality with such a spectrum of security-performance trade-offs.

We compare our proposed schemes to prior work in Table 8.2, highlighting shared attributes as well as distinctive features.

Scheme	**Dims**	**Volume**	**Access**	**Latency (ms/label)**	**Tunable**
EMM (base) [1]	Multi	None	None	0.1668ms	No
dprfMM [29]	1D	Full pad (strong)	None	≈1.2ms	No
VLH/AVLH [28]	1D	Comp. (tunable)	None	≈0.01 ms	Partial (vol. only)
ORAM Base [11]	Any	ORAM pad (strong)	ORAM acc. (strong)	$10-100$ms	No
FPEMM	Multi	Fixed pad	None	0.2801ms	Yes (access vs. perf.)
DPRFMM	Multi	Full pad (strong)	None	1.2083ms	No
realDPRFMM	Multi	Bounded pad	None	0.1146ms	Yes (vol. vs. lat.)
AHEMM	Multi	None	Tuple-level enc.	0.0127ms	Yes (access vs. perf.)
VHAHEMM	Multi	Fixed pad	Tuple + pad	0.0121ms	Yes (access vs. perf.)

Table 8.2: Comparison of encrypted multi-map schemes.

The values listed in the “Latency” column of Table 8.2 report the average processing time, in milliseconds per matched record (or “label”), for each scheme under our standard linear□cover query workload. Concretely, for each scheme we plotted total query execution time against the number of returned labels, and then computed the slope of that line, which directly gives the mean cost to fetch, decrypt and materialize a single label. For the prior, 1-dimensional schemes and ORAM, we have used approximate values based on their published performance characteristics.

8.3. Implications

The findings presented in this thesis have meaningful implications for both theoretical research and practical deployment of EMM systems. From a theoretical perspective, the observed performance–privacy trade-offs highlight the need for formal models that go beyond binary leakage assumptions and instead account for intermediate and tunable leakage levels. Our padding approach, as demonstrated through realDPRFMM, introduces a middle ground between worst-case uniform padding and exact leakage, which invites further formal analysis. This suggests new opportunities to redefine existing leakage models by incorporating realistic occupancy distributions and to explore novel complexity metrics that reflect the cumulative cost across all stages of query execution, namely, trapdoor generation, server-side search, and client-side decryption.

In practical terms, the experiments offer actionable guidance to system designers and implementers. The clear distinction between dense and sparse datasets shows that one-size-fits-all solutions are suboptimal: in environments with sparse or low-selectivity data, lightweight schemes such as EMM and AHEMM are sufficient and cost-effective. In contrast, dense multi-valued datasets often expose more exploitable volume leakage, motivating the use of padding schemes. While realDPRFMM shows strong performance and offers a practical compromise between leakage resistance and efficiency, it should be recognized that its relaxed volume-hiding guarantees result in slightly more observable leakage than stricter schemes like DPRFMM. This trade-off, introduces certain risks in adversarial settings where even minor variations in response sizes could be exploited.

Nevertheless, the low overhead of realDPRFMM makes it a compelling choice for systems prioritizing performance alongside reasonable privacy guarantees. It also highlights the potential of adaptive padding techniques, which dynamically adjust based on empirical data properties. This insight opens up the potential for implementing dynamic padding mechanisms that optimize performance while preserving privacy, guided by realistic bounds. Finally, the methodology used in this thesis, especially the separation of performance into discrete operational phases, offers a reproducible framework for benchmarking future constructions.

8.4. Limitations

While the results presented in this thesis offer valuable insights into the performance and leakage trade-offs of various EMM constructions, several limitations must be acknowledged to contextualize the scope and applicability of the findings.

First, the threat model adopted throughout this work assumes a passive, honest-but-curious adversary. That is, the server is assumed to faithfully execute the protocol without deviation, while attempting to learn as much as possible from the observable leakage. This model excludes more powerful adversaries that may behave maliciously, e.g. by modifying data, crafting adaptive queries, or deviating from the protocol, which would necessitate additional cryptographic mechanisms such as verifiable computation or integrity and consistency checks. As such, the security guarantees discussed herein do not extend to adversarial settings involving active or Byzantine behavior [50].

Second, this work operates under a static data model, where the encrypted index is generated once and remains unchanged during querying. Many practical applications require dynamic operations such as insertions, deletions, and updates, which introduce additional leakage channels (e.g. update patterns or versioning metadata) and performance trade-offs. Extending these constructions to dynamic settings remains an important direction for future work.

Third, the analysis in this thesis is restricted to two primary leakage dimensions: access and volume. While these are widely studied in evaluating privacy in encrypted search, certain constructions, such as DPRFMM and realDPRFMM, can also leak structural information. For instance, the positional behavior of elements within cuckoo hash tables or stash usage patterns may serve as side channels, particularly when adversaries observe repeated queries. Other forms of leakage, such as intersection patterns or co-occurrence frequencies, are also not modeled in the current analysis.

Fourth, the evaluation is based on two synthetic datasets representing sparse and dense multi-map structures. While these datasets offer meaningful contrasts and expose performance boundaries, real-world data often consists of more complex and skewed distributions. For instance, keyword frequency might follow a Zipfian distribution [51], or access patterns may exhibit temporal locality. These nuances can influence both leakage characteristics and performance outcomes in ways not fully captured in the current benchmarks.

Finally, the experimental setup assumes a single-client model. In practice, systems often support multiple clients issuing queries concurrently, which raises new challenges. Cross-client correlation attacks [52], contention-induced timing variations [53], and query pattern interference [54] are all factors that could influence leakage and system performance. Exploring these multi-client dynamics requires further investigation.

8.5. Reflection on the Research Process

My interest in encrypted search began last year during a seminar where I wrote an essay titled *Machine Learning Optimization of Homomorphic Encryption*. In that essay, I explored the fundamentals of homomorphic encryption and began sketching a basic search scheme for identifying optimal parameter sets. This initial exploration sparked my curiosity in the broader space of privacy-preserving computation, and it led me to contact Professor Markatou to ask if she would consider supervising my Master's thesis. Fortunately for me she agreed, a decision for which I remain very grateful.

Professor Markatou proposed that I explore directions that build on her previous work, and I started reading into the topic of leakage in encrypted databases. I quickly realized this was an area I had little prior exposure to, as most of my coursework had not focused on this particular threat model. The first few weeks of the project were therefore heavily focused on background reading, understanding leakage types, survey papers, and how different constructions attempt to mitigate information exposure. This literature review clarified core concepts and gave me concrete ideas on how I could structure my own work.

Initially, I used the same datasets employed by Falzon et al. [1] in their earlier experiments. However, these turned out to be single-valued datasets, which limited the behavior I was able to observe during evaluation. The results I was seeing didn't align with my expectations, and I realized the dataset wasn't giving me the insights I needed. To address this, I created a synthetic dataset with multiple values per key, better suited to testing volume-hiding constructions.

Of course, this decision came with its own challenges. Much of the benchmarking code written by Falzon et al. [1] was designed for single-valued datasets, and it became incompatible with the new format I introduced. At first, I tried to adapt the existing code-base incrementally, fixing issues one by one, but this quickly became cumbersome. Eventually, I realized it would be more efficient to design and implement my own benchmark class from scratch. This process taught me that while theoretical understanding is essential, a practical mindset and flexibility are just as important in applied research.

After completing my thesis proposal, I needed to find a co-supervisor. I was fortunate that Professor Liang and his PhD student Huanhuan agreed to take on this role. I began having weekly meetings with Huanhuan, whose feedback was incredibly helpful in shaping my project. These regular check-ins helped me stay focused, track my progress, and improve the clarity and precision of my work. In hindsight, I believe that if I had started this collaboration earlier, the quality of my work would have improved even further.

Overall, this project taught me a great deal, not just about encrypted search and leakage profiles, but also about what it takes to carry out a research project from start to finish. I learned how to navigate unfamiliar technical terrain, adapt tools to suit new requirements, and seek out guidance and collaboration. These are lessons I will carry with me into future academic and professional work.

Part IV

Closure

9
Conclusion

9.1. Answers to Research Questions

This section addresses the research questions formulated in Section 1.3, in their original order.

Research Question 1

How do the EMM leakage profiles influence the average trapdoor generation time, search time, resolve time, and result count under identical dataset and query workloads?

The influence of EMM leakage profiles on performance was evaluated by measuring four key metrics: trapdoor generation time, search time, resolve time, and result count, across identical datasets and query workloads. The results revealed that while all EMM variants share the same basic structure, their leakage characteristics significantly affect performance in distinct ways.

Trapdoor generation time was largely uniform across most variants, with the exception of realDPRFMM. This variant benefits from caching its derived HMAC key, thereby avoiding repeated KDF operations and achieving faster trapdoor generation than its counterparts. Other schemes such as EMM, AHEMM, VHAHEMM, FPEMM, and DPRFMM showed nearly identical and higher latencies in this phase.

Search time was strongly influenced by the amount of padding introduced to hide volume leakage. FPEMM, which pads all labels to the global maximum length, resulted in the highest search time on dense datasets. On sparse datasets, DPRFMM became the slowest due to the inefficiency of processing many padded dummy entries. VHAHEMM offered a more moderate performance, padding each label to a fixed bound. AHEMM and baseline EMM, which perform no volume hiding, achieved lower search times. The realDPRFMM variant consistently returned the fastest search times by using a realistic padding bound that reduces unnecessary dummy data while offering partial volume hiding.

Resolve time, the client-side decryption and filtering cost, also varied by leakage profile. The baseline EMM performed best here, decrypting only the exact set of relevant ciphertexts. realDPRFMM came close, thanks to its tight padding bounds. VHAHEMM introduced slight overhead due to fixed-size padding, while AHEMM remained comparable to the baseline in sparse settings. In contrast, FPEMM and DPRFMM had the highest resolve times, especially on dense data, due to the large volume of padded entries requiring decryption.

Finally, the result count represents both the bandwidth overhead and degree of volume leakage. This directly reflected the padding strategy. EMM and AHEMM returned exact match counts, leaking full volume information but having minimal overhead. FPEMM always returned the global maximum number of ciphertexts per label, offering complete volume hiding at the cost of significant bandwidth. VHAHEMM and DPRFMM applied fixed or formulaic padding, balancing leakage and efficiency. realDPRFMM stood out by padding to realistic occupancy levels, which reduced bandwidth while maintaining a meaningful level of volume obfuscation.

Overall, the evaluation showed a clear trade-off between privacy and efficiency. Schemes like EMM and AHEMM provide optimal performance but full volume leakage, while FPEMM and DPRFMM achieve strong privacy guarantees at high computational cost. VHAHEMM and especially realDPRFMM offer a flexible middle ground, enabling system designers to tune leakage according to application-specific constraints.

Research Question 2

What quantitative relationship exists between specific leakage metrics and each performance metric?

The relationship between leakage and performance was quantified by modeling how leakage mitigation strategies impact the core performance metrics. Specifically, we observed near-linear scaling between the extent of volume-hiding padding and the search, resolve, and bandwidth costs.

Search time showed a strong positive correlation with the amount of padding introduced to hide result volumes. Fully padded schemes, such as DPRFMM and FPEMM, had the steepest search time increases with respect to query coverage, showing the computational overhead of processing large padded lists. Schemes like realDPRFMM and VHAHEMM had gentler slopes, with realDPRFMM notably reducing both fixed overhead and per-unit cost by selecting realistic padding bounds.

Resolve time and bandwidth, which was measured by result count, also scaled linearly with the padding size per label. Schemes with minimal leakage, such as EMM and AHEMM, returned only the true matches, leading to optimal decryption efficiency. In contrast, FPEMM and DPRFMM returned significantly more ciphertexts per query due to their aggressive padding strategies, directly inflating client-side decryption cost. realDPRFMM achieving savings by limiting padding without reverting to full leakage.

By computing normalized AUC metrics, we captured the average per-percent-coverage cost across the full query range. These showed that schemes enforcing complete volume hiding (DPRFMM, FPEMM) had the highest total overhead, while constructions like realDPRFMM and VHAHEMM significantly reduced cost without entirely compromising on leakage control. These findings establish a quantitative trade-off where more aggressive leakage mitigation imposes proportionally higher computational and bandwidth costs, while approaches like realDPRFMM or VHAHEMM can offer a flexible balance suited to practical deployment.

Research Question 3

How does varying the dataset configuration impact the performance metrics of each EMM variant?

Varying dataset configuration, specifically, the density of keyword-to-document mappings, affects EMM variants performance primarily by altering the effectiveness and overhead of padding strategies. We compare a “dense” dataset (many documents per keyword) with a “sparse” dataset (few documents per keyword) to observe how each scheme’s padding and access-pattern hiding mechanisms respond to changing true result sizes.

Since trapdoor creation relies on per-label cryptographic operations (KDF and HMAC), its cost remains essentially invariant across dataset configurations for all variants except realDPRFMM. The caching optimization in realDPRFMM yields consistently lower trapdoor latency, independent of density. All other schemes show similar trapdoor times whether the dataset is dense or sparse.

In dense datasets, schemes with aggressive padding (FPEMM and DPRFMM) suffer pronounced penalties in which processing large padded lists increase server-side latency. FPEMM’s global padding to the maximum list length causes the greatest slowdown, while DPRFMM’s fixed $2B + S$ padding similarly decreases its performance. By contrast, in sparse datasets, DPRFMM suffers most. Its padding overhead (dummy entries) overwhelms the few real matches. realDPRFMM, AHEMM and VHAHEMM adapt better with their padding bounds reducing unnecessary dummy processing in both dense and sparse settings, returning relatively stable search times.

Client-side decryption cost scales with the number of ciphertexts returned, so FPEMM and DPRFMM show the largest resolve times on dense data and maintain high overhead even on sparse data. VHAHEMM and AHEMM decrypt the fewest results by hiding the access pattern, thus having fewer encrypted entries in the first place. As a result they show the fastest resolve times on dense data, where other schemes will have proportionally more encrypted data.

The total number of ciphertexts returned per query directly reflects the padding strategy's interaction with dataset density. In dense datasets, full-padding schemes transmit maximal volumes, drastically increasing bandwidth. In sparse configurations, DPRFMM's constant padding yields excessive dummy traffic, whereas FPEMM's padding to a global maximum similarly overshoots. realDPRFMM scales padding in line with realistic occupancy, reducing dummy traffic in sparse datasets while still masking volume sufficiently in dense datasets.

Research Question 4

Which EMM variants achieve the most favorable security–performance trade-off across different datasets and query selection scenarios?

Across both evaluated datasets and a range of query coverage scenarios, the EMM schemes that offer lightweight leakage protections achieve the most favorable balance between security and performance. In particular, schemes that incorporate efficient access- and volume-hiding mechanisms without resorting to full padding demonstrate superior throughput while still mitigating key leakage vectors.

On the dense dataset (Dataset 1), the AHEMM and VHAHEMM variants perform especially well, achieving the lowest per-label latency among all evaluated schemes. With measured average costs of just 0.0127 ms/label (AHEMM) and 0.0121 ms/label (VHAHEMM), both schemes outperform even the unprotected baseline in high-volume query scenarios. These schemes also maintain low fixed overheads and normalized area-under-curve (AUC) values (0.52 ms·%) across a wide range of query sizes, highlighting their scalability. AHEMM offers tuple-level access pattern hiding, while VHAHEMM extends this with bounded volume padding, offering improved leakage profiles at negligible performance cost. It is also worth noting that schemes which hide the access pattern often result in fewer ciphertext entries, as encrypted tuples are fetched and processed in bulk rather than individually. This not only strengthens security by concealing access behavior, but also reduces the number of decryption operations required per query, thereby improving overall efficiency.

In contrast, realDPRFMM proves most advantageous on the sparse dataset (Dataset 2) and for workloads composed predominantly of small queries. With a latency slope of just 0.0032 ms/label and minimal setup overhead, realDPRFMM delivers excellent responsiveness by tuning its padding to realistic occupancy bounds. This makes it ideal for latency-sensitive applications that can tolerate moderate, bounded leakage.

Schemes like baseline EMM and FPEMM provide less favorable trade-offs. While the baseline EMM is simple and performs moderately (0.1668 ms/label), it offers no protection against volume leakage. FPEMM achieves full volume-hiding through fixed-size padding, but this comes at a substantial latency cost (0.2801 ms/label), making it less suitable for high-throughput or real-time use cases.

Finally, DPRFMM achieves the strongest leakage protection through aggressive padding, but incurs the highest performance penalty. With an average cost of 1.2083 ms/label and a normalized AUC of over 50 ms·%, it is only suitable in scenarios where leakage resilience outweighs performance constraints.

In conclusion, AHEMM and VHAHEMM offer the most favorable security–performance trade-off for general-purpose deployments requiring strong leakage mitigation with high efficiency. For scenarios emphasizing low-latency retrieval of small results, realDPRFMM provides a compelling alternative. Conversely, schemes with fixed, heavy padding (FPEMM, DPRFMM) should be reserved for contexts where maximal leakage resistance is of great importance and latency is a secondary concern.

9.2. Closing Remarks

This thesis set out to investigate the problem of designing EMM schemes that offer a meaningful balance between leakage resilience and query performance. While prior work has addressed one-dimensional search or resorted to heavy-weight protections such as full ORAM, our focus was on constructing and evaluating multi-dimensional EMM variants that enable efficient range queries while controlling the extent of access and volume leakage.

To that end, we introduced five new EMM variants (AHEMM, VHAHEMM, FPEMM, DPRFMM, and realDPRFMM) and evaluated them alongside a baseline EMM under varied dataset densities and query coverage levels. Using both latency slope and AUC as performance indicators, we quantified the cost of each leakage-hiding strategy. Our results show that AHEMM and VHAHEMM offer the best overall balance for dense datasets and large queries, while realDPRFMM excels in sparse or small-query scenarios. On the other hand, schemes with heavy padding such as DPRFMM and FPEMM incur significantly higher latencies but provide stronger leakage protection, making them appropriate only in highly sensitive applications.

The practical impact of these findings lies in offering clear guidance to system designers that by selecting or tuning an EMM variant based on workload characteristics and security requirements, it is possible to achieve strong leakage mitigation without sacrificing performance. Theoretically, the thesis contributes a more nuanced understanding of how leakage-resilience techniques affect system efficiency and reveals the potential of intermediate approaches.

Key Takeaways:

1. AHEMM, VHAHEMM and realDPRFMM strike a practical middle ground between leakage resistance and performance, making them suitable for a broad range of applications.
2. The choice of scheme should be informed by workload characteristics: large-range queries benefit from lightweight tuple-based protection, while small-result queries are best served by bounded-volume padding.
3. Full leakage hiding is costly and not always necessary; selecting just enough padding to meet a specific threat model can drastically improve latency without compromising essential security goals.

10 Future Work

This chapter provides a brief overview of the primary short and long term recommendations for the future continuation of this research.

10.1. Short Term Directions

There are a number of ways this work could be extended in the near future. One improvement is to consider a different threat model. This thesis assumed that the server is honest-but-curious, i.e. the adversary follows the rules but tries to learn from whatever it can observe. In future work, one could add basic protections like integrity checks to detect if the server tries to return incorrect results.

Another short-term step is to move beyond a fixed, unchanging dataset. Most real-world systems need to support updates (adding, deleting or changing entries). A good starting point could be experimenting with limited forms of updates, like append-only data, to see how updates affect performance and leakage.

It would also be useful to look at more than just access and volume leakage. For example, certain constructions (DPRFMM and realDPRFMM) might leak structural information, such as where elements end up in a hash table or whether stash entries are used. Measuring things like stash size or how often labels appear together in queries could uncover new leakage patterns.

Future experiments could also try to use more realistic datasets. The two datasets used in this thesis (dense and sparse) are helpful, but real data, such as document collections or keyword logs, could behave very differently. Testing on real workloads might show new performance issues or uncover different leakage risks.

Another useful short-term direction is to look at how the system performs with multiple clients. So far, this work assumes only one client. But in real deployments, many users might send queries at the same time. Testing how this affects timing, contention, and leakage would help make the system more practical.

Lastly, it would be interesting to compare DPRFMM to a much simpler baseline: an EMM that just returns the entire database on every query. While this naive approach leaks nothing and avoids padding entirely, it is obviously very inefficient. Since DPRFMM uses more advanced search techniques like cuckoo hashing, it should be faster than returning everything. Measuring how big the performance difference is would help justify DPRFMM's design.

10.2. Long Term Directions

In the longer term, there are several bigger goals that could move this research forward. One direction is to build systems that adjust padding automatically. Instead of using a fixed value for P_{real}, future systems could learn from the data or the query workload and change the padding size on the fly. This would allow the system to stay efficient while still protecting privacy.

It would also be worth looking at other kinds of leakage that weren't covered in this thesis. For example, things like timing differences [55], memory access patterns [56], or network message sizes [57] might leak information. Protecting against these side channels might require writing parts of the system in constant-time code or changing how data is laid out in memory.

Making the system scalable is another long-term goal. So far, everything was tested on a single-threaded setup. In real systems, one might want to run the server on multiple cores or even across several machines. Figuring out how to do this while still protecting privacy, especially when servers need to coordinate their actions, would be a big step forward.

Lastly, key management [58, 59] is an important step in real-world deployments. This includes rotating keys, revoking old keys, and distributing new ones securely. Future work could explore how to support key changes without having to rebuild or re-pad the whole index from scratch.

10.3. Final Thoughts

When I first began exploring encrypted search, I was drawn by the tension between usability and secrecy. Working through the experiments in this thesis has given me a deeper appreciation for how even small design choices ripple out into performance and privacy in unexpected ways. Ultimately, I hope this work helps move us closer to systems where every user, even those with limited technical background, can query their data in the cloud without sacrificing confidentiality. To the reader, thank you for taking the time to read this work. Your interest means a great deal, and I hope these ideas spark new projects, conversations, and innovations in the years ahead.

Bibliography

[1] Francesca Falzon et al. "Range search over encrypted multi-attribute data". In: *Proc. VLDB Endow. 16, 4, 587–600* (2022).

[2] European Commission – FISMA Newsroom. *European Commission updates on emphFISMA developments*. https://ec.europa.eu/newsroom/fisma/items/632027/en. July 2018.

[3] George Lawton. *The pros and cons of big data outsourcing*. https://www.techtarget.com/searchdatamanagement/feature/The-pros-and-cons-of-big-data-outsourcing. Aug. 2021.

[4] Narendra Sahoo. *Role of Encryption in GDPR Compliance*. https://www.tripwire.com/state-of-security/role-of-encryption-in-gdpr-compliance. Mar. 2021.

[5] Andrada Coos. *GDPR Data Encryption Requirements*. https://www.endpointprotector.com/blog/gdpr-data-encryption-requirements/. Apr. 2021.

[6] HiComply. *The world's largest data breaches and the financial cost to businesses revealed*. https://www.hicomply.com/blog/the-worlds-largest-data-breaches-and-the-financial-cost-to-businesses-revealed.

[7] New York Post. *Major data hack nabs 184 M passwords for Google, Apple and more: 'Cybercriminal's dream'*. https://nypost.com/2025/05/28/tech/major-data-hack-nabs-184m-passwords-for-google-apple-more/. May 2025.

[8] Kiuwan. *Data Breaches Are More Expensive Than They Seem*. https://www.kiuwan.com/blog/most-expensive-security-breaches. Jan. 2022.

[9] IBM. *Surging data breach disruption drives costs to record highs – Cost of a Data Breach Report 2024*. https://www.ibm.com/think/insights/whats-new-2024-cost-of-a-data-breach-report. July 2024.

[10] Francesca Falzon et al. *Encrypted Range Search Repository*. https://github.com/cloudsecuritygroup/ers. Accessed: November 2024. 2022.

[11] Oded Goldreich et al. "Software Protection and Simulation on Oblivious RAMs". In: *Journal of the ACM (JACM)* 43.3 (1996), pp. 431–473. doi: 10.1145/233551.233552. url: https://doi.org/10.1145/233551.233552.

[12] Craig Gentry. "A fully homomorphic encryption scheme". AAI3382729. PhD thesis. Stanford, CA, USA, 2009.

[13] Zvika Brakerski et al. "Efficient Fully Homomorphic Encryption from (Standard) LWE". In: *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. 2011, pp. 97–106. doi: 10.1109/FOCS.2011.12.

[14] Emil Stefanov et al. "Path ORAM: An Extremely Simple Oblivious RAM Protocol". In: *J. ACM* 65.4 (Apr. 2018). doi: 10.1145/3177872. url: https://doi.org/10.1145/3177872.

[15] S Goldwasser et al. "The knowledge complexity of interactive proof-systems". In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. STOC '85. Providence, Rhode Island, USA: Association for Computing Machinery, 1985, pp. 291–304. doi: 10.1145/22145.22178. url: https://doi.org/10.1145/22145.22178.

[16] Reza Curtmola et al. "Searchable symmetric encryption: improved definitions and efficient constructions". In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. CCS '06. Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 79–88. doi: 10.1145/1180405.1180417. url: https://doi.org/10.1145/1180405.1180417.

[17] Mihir Bellare et al. “Deterministic and efficiently searchable encryption”. In: *Proceedings of the 27th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO’07. Santa Barbara, CA, USA: Springer-Verlag, 2007, pp. 535–552.

[18] Alexandra Boldyreva et al. “Order-Preserving Symmetric Encryption”. In: vol. 5479. Apr. 2009, pp. 224–241. doi: `10.1007/978-3-642-01001-9_13`.

[19] David Cash et al. “Highly-scalable searchable symmetric encryption with support for boolean queries”. In: *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Springer. 2013, pp. 353–373.

[20] Dawn Xiaoding Song et al. “Practical techniques for searches on encrypted data”. In: *Proceeding 2000 IEEE Symposium on Security and Privacy. SP 2000*. 2000, pp. 44–55. doi: `10.1109/SECPRI.2000.848445`.

[21] Feng Li et al. “A Survey on Searchable Symmetric Encryption”. In: *ACM Comput. Surv.* 56.5 (Nov. 2023). doi: `10.1145/3617991`. url: `https://doi.org/10.1145/3617991`.

[22] Seny Kamara et al. “Dynamic Searchable Symmetric Encryption”. In: (Oct. 2012). doi: `10.1145/2382196.2382298`.

[23] Debrup Chakraborty et al. *Making Searchable Symmetric Encryption Schemes Smaller and Faster*. Cryptology ePrint Archive, Paper 2024/1483. 2024. doi: `doi.org/10.1007/s10207-024-00915-y`. url: `https://eprint.iacr.org/2024/1483`.

[24] Stefania Nita et al. “Searchable Encryption”. In: Sept. 2023, pp. 89–134. doi: `10.1007/978-3-031-43214-9_4`.

[25] Laura Blackstone et al. “Revisiting Leakage Abuse Attacks”. In: *NDSS* (Jan. 2020). doi: `10.14722/ndss.2020.23103`.

[26] Evangelia Anna Markatou et al. “Attacks on encrypted response-hiding range search schemes in multiple dimensions”. In: *Proceedings on Privacy Enhancing Technologies* (2023).

[27] Georgios Kellaris et al. “Generic attacks on secure outsourced databases”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1329–1340.

[28] Seny Kamara et al. “Computationally Volume-Hiding Structured Encryption”. In: *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1326–1343. doi: `10.1109/SP46215.2022.00076`. url: `https://doi.org/10.1109/SP46215.2022.00076`.

[29] Anson Patel et al. “Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019, pp. 215–232. doi: `10.1145/3319535.3329901`. url: `https://doi.org/10.1145/3319535.3329901`.

[30] Alexandra Boldyreva et al. *Encrypted Multi-map that Hides Query, Access, and Volume Patterns*. Cryptology ePrint Archive, Paper 2024/2091. 2024. url: `https://eprint.iacr.org/2024/2091`.

[31] John W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. 4th. Thousand Oaks, CA: SAGE Publications, 2014.

[32] Fred N. Kerlinger et al. *Foundations of Behavioral Research*. 4th. Fort Worth, TX: Harcourt College Publishers, 2000.

[33] A. P. Bradley. “The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms”. In: *Pattern Recognition* 30.7 (1997), pp. 1145–1159. doi: `10.1016/S0031-3203(96)00142-2`.

[34] J. A. Hanley et al. “The Meaning and Use of the Area under a Receiver Operating Characteristic (ROC) Curve”. In: *Radiology* 143.1 (Apr. 1982), pp. 29–36. doi: `10.1148/radiology.143.1.7063747`.

[35] Janez Demšar. “Statistical Comparisons of Classifiers over Multiple Data Sets”. In: *J. Mach. Learn. Res.* 7 (Dec. 2006), pp. 1–30.

[36] Evangelia Anna Markatou et al. “Reconstructing with Less: Leakage Abuse Attacks in Two Dimensions”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2243–2261. doi: `10.1145/3460120.3484552`. url: `https://doi.org/10.1145/3460120.3484552`.

[37] Viraj Biharie. *Tunable-EMM-schemes: Implementations and benchmarks of tunable-leakage encrypted multi-map schemes*. `https://github.com/vaabiharie/Tunable-EMM-schemes`. 2025.

[38] Chang Liu et al. “Search pattern leakage in searchable encryption: Attacks and new construction”. In: *Inf. Sci.* 265 (May 2014), pp. 176–188. doi: `10.1016/j.ins.2013.11.021`. url: `https://doi.org/10.1016/j.ins.2013.11.021`.

[39] Mohammad Saiful Islam et al. “Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation”. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. Briefing paper, 6 Feb 2012. San Diego, CA: The Internet Society, Feb. 2012. url: `https://www.ndss-symposium.org/ndss2012/ndss-2012-programme/access-pattern-disclosure-searchable-encryption-ramification-attack-and-mitigation`.

[40] Reza Curtmola et al. “Searchable symmetric encryption: improved definitions and efficient constructions”. In: *Proceedings of the 13th ACM conference on Computer and communications security*. 2006, pp. 79–88.

[41] Megumi Ando et al. “On the Cost of Suppressing Volume for Encrypted Multi-maps”. In: *Proceedings on Privacy Enhancing Technologies* 2022 (Oct. 2022), pp. 44–65. doi: `10.56553/popets-2022-0098`.

[42] Archita Agarwal et al. “Concurrent Encrypted Multimaps”. In: *Advances in Cryptology – ASIACRYPT 2024: 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9–13, 2024, Proceedings. Part IV*. Kolkata, India: Springer-Verlag, 2024, pp. 169–201. doi: `10.1007/978-981-96-0894-2_6`. url: `https://doi.org/10.1007/978-981-96-0894-2_6`.

[43] Seny Kamara et al. “Structured Encryption and Leakage Suppression”. In: *Advances in Cryptology – CRYPTO 2018*. Vol. 10991. Lecture Notes in Computer Science. Springer, 2018, pp. 339–370. doi: `10.1007/978-3-319-96884-1_12`.

[44] Seny Kamara et al. “Structured Encryption and Dynamic Leakage Suppression”. In: *Advances in Cryptology – EUROCRYPT 2021*. Vol. 12696. Lecture Notes in Computer Science. Springer, 2021, pp. 346–375. doi: `10.1007/978-3-030-77883-5\_13`.

[45] John Gill. “Computational Complexity of Probabilistic Turing Machines”. In: *SIAM J. Comput.* 6.4 (Dec. 1977), pp. 675–695. doi: `10.1137/0206049`. url: `https://doi.org/10.1137/0206049`.

[46] Shafi Goldwasser et al. “Probabilistic Encryption”. In: *Journal of Computer and System Sciences* 28.2 (Apr. 1984), pp. 270–299. doi: `10.1016/0022-0000(84)90070-9`.

[47] Jonathan Katz et al. *Introduction to Modern Cryptography*. 2nd. Boca Raton, FL: Chapman & Hall/CRC, 2014.

[48] Viet Vo et al. “ShieldDB: An Encrypted Document Database With Padding Countermeasures”. In: *IEEE Transactions on Knowledge and Data Engineering* 35.4 (2023), pp. 4236–4252. doi: `10.1109/TKDE.2021.3126607`.

[49] Ghous Amjad et al. *Dynamic Volume-Hiding Encrypted Multi-Maps with Applications to Searchable Encryption*. Cryptology ePrint Archive, Paper 2021/765. 2021. doi: `10.56553/popets-2023-0025`. url: `https://eprint.iacr.org/2021/765`.

[50] Leslie Lamport et al. “The Byzantine Generals Problem”. In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. doi: `10.1145/357172.357176`. url: `https://doi.org/10.1145/357172.357176`.

[51] George Kingsley Zipf. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Cambridge, MA: Addison-Wesley Press, 1949.

[52] Simon Oya et al. *Hiding the Access Pattern is Not Enough: Exploiting Search Pattern Leakage in Searchable Encryption*. 2020. arXiv: 2010.03465 [cs.CR]. url: https://arxiv.org/abs/2010.03465.

[53] Tuo Li et al. *Fast Selective Flushing to Mitigate Contention-based Cache Timing Attacks*. 2022. arXiv: 2204.05508 [cs.CR]. url: https://arxiv.org/abs/2204.05508.

[54] Zhiwei Shang et al. *Obfuscated Access and Search Patterns in Searchable Encryption*. 2021. arXiv: 2102.09651 [cs.CR]. url: https://arxiv.org/abs/2102.09651.

[55] Jean-Luc Danger et al. "High-order timing attacks". In: *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*. CS2 '14. Vienna, Austria: Association for Computing Machinery, 2014, pp. 7–12. doi: 10.1145/2556315.2556316. url: https://doi.org/10.1145/2556315.2556316.

[56] Dag Arne Osvik et al. "Cache-Collision Timing Attacks Against AES". In: *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES) Workshop*. Vol. 4249. Lecture Notes in Computer Science. Yokohama, Japan: Springer, 2006, pp. 201–215. doi: 10.1007/11894063_16.

[57] Saman Feghhi et al. "An Efficient Web Traffic Defence Against Timing-Analysis Attacks". In: 14.2 (Feb. 2019), pp. 525–540. doi: 10.1109/TIFS.2018.2855655. url: https://doi.org/10.1109/TIFS.2018.2855655.

[58] Ievgeniia Kuzminykh et al. "Comparative Analysis of Cryptographic Key Management Systems". In: *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*. Springer International Publishing, 2020, pp. 80–94. doi: 10.1007/978-3-030-65729-1_8. url: http://dx.doi.org/10.1007/978-3-030-65729-1_8.

[59] Sofia Anna Menesidou et al. "Cryptographic Key Management in Delay Tolerant Networks: A Survey". In: *Future Internet* 9.3 (2017). doi: 10.3390/fi9030026. url: https://www.mdpi.com/1999-5903/9/3/26.

A
Algorithms

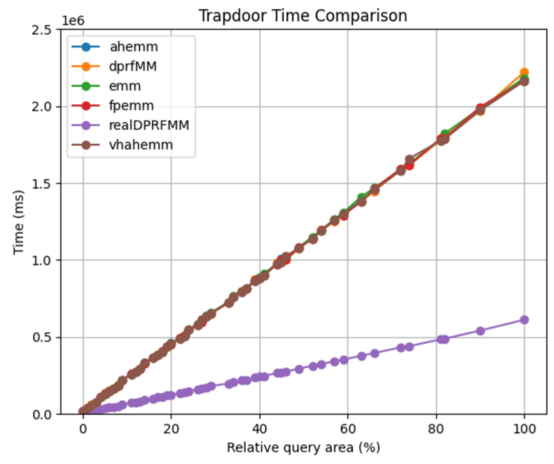
Algorithm Synthetic Data Generation

```
procedure SynthesizeData($n, K, V, m$)
                                   ▷ $n$: number of entries; $K = (k_{min}, k_{max})$; $V = (v_{min}, v_{max})$; $m$= max tuple size
    Clear existing data
    all_keys $\leftarrow$ CartesianProduct$(K)$
    if $n >$ length(all_keys) then
        raise Error("Too many entries")
    end if
    chosen_keys $\leftarrow$ random sample of $n$ keys from all_keys
    buckets $\leftarrow [\,]$
    for $e$ from $\lfloor \log_{10}(v_{min}) \rfloor$ to $\lfloor \log_{10}(v_{max}) \rfloor$ do
        $\ell \leftarrow$ max $(v_{min}, 10^e)$
        $h \leftarrow$ min$(v_{max}, 10^{e+1} - 1)$
        if $h > \ell$ then
            Append $(\ell, h)$ to buckets
        end if
    end for
    for each $k$ in chosen_keys do
        $\ell \leftarrow$ random integer in $[1, m]$
        $v \leftarrow [\,]$
        for $i = 1$ to $\ell$ do
            $(low, high) \leftarrow$ random choice from buckets
            Append random integer from $[low, high]$ to $v$
        end for
        data$[k] \leftarrow$ tuple$(v)$
    end for
end procedure
```

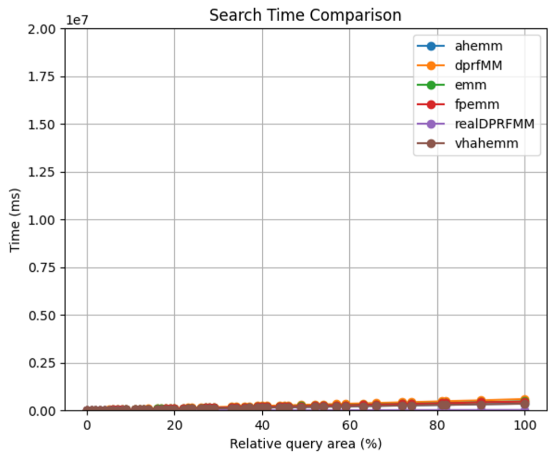
Algorithm Original EMM Build Encrypted Index

```
1:  procedure BuildIndex($key, plaintext\_mm$)
2:      $hmac\_key \leftarrow$ HashKDF($key$, PURPOSE_HMAC)
3:      $enc\_key \leftarrow$ HashKDF($key$, PURPOSE_ENCRYPT)
4:      $encrypted\_db \leftarrow \emptyset$
5:      for each $(label, values) \in plaintext\_mm$ do
6:          $token \leftarrow$ HMAC($hmac\_key, label$)
7:          for $index, value \in values$ do
8:              $ct\_label \leftarrow$ Hash($token +$ bytes($index$))
9:              $ct\_value \leftarrow$ SymmetricEncrypt($enc\_key, value$)
10:             $encrypted\_db[ct\_label] \leftarrow ct\_value$
11:         end for
12:     end for
13:     return $encrypted\_db$
14: end procedure
```

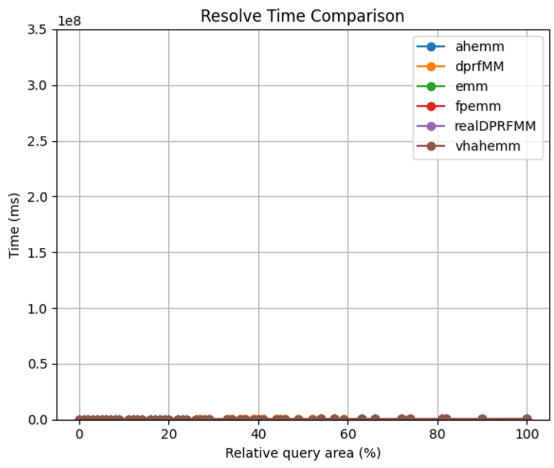
B
Additional Figures



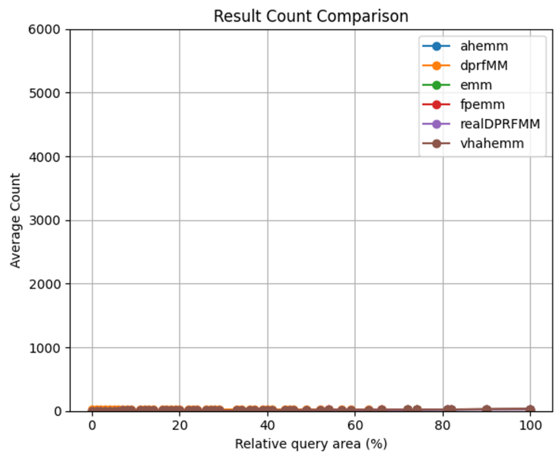
(a) Trapdoor Time Comparison (no scaling necessary)



(b) Search Time Comparison (scaled)



(c) Resolve Time Comparison (scaled)



(d) Result Count (scaled)

Figure B.1: Scaled comparison of evaluation metrics across all EMM variants on Dataset 2 (scaled using the same range as Dataset 1)