Performance Evaluation of Pattern Matching Encryption

by

John Collier Dvorak

A Project Report Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Supervised by Professor Alan Kaminsky

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences Rochester Institute of Technology Rochester, New York

May 2015
Abstract

Performance Evaluation of Pattern Matching Encryption

John Collier Dvorak

Supervising Professor: Professor Alan Kaminsky

Client-server architectures are increasingly useful in today’s world of cloud computing. One common disadvantage of using an external server is security; server hardware is frequently outside of a user’s control. More complex cryptographic systems have been proposed to mitigate security issues on an untrusted server, universally at a cost to performance. One such proposal, Pattern Matching Encryption presents a system of queryable encryption and estimated performance overhead. This project implements a specific proposed query system and measures performance of the system over a variety of variable parameters.
# Contents

Abstract ................................................................. ii

1 Introduction .......................................................... 1
   1.1 Background ....................................................... 1
   1.2 Motivation ....................................................... 1
   1.3 Solution .......................................................... 2
   1.4 Roadmap .......................................................... 2

2 Design ................................................................. 3
   2.1 Key Generation ................................................... 4
      2.1.1 Client ....................................................... 4
   2.2 Encryption ....................................................... 4
      2.2.1 Client ....................................................... 4
      2.2.2 Server ...................................................... 5
      2.2.3 Communication ............................................ 6
   2.3 Search ............................................................ 6
      2.3.1 Client ....................................................... 6
      2.3.2 Server ...................................................... 7
      2.3.3 Communication ............................................ 7

3 Implementation ....................................................... 9
   3.1 General Design .................................................. 9
   3.2 Assumptions ..................................................... 10
   3.3 Modules .......................................................... 10
      3.3.1 Encryption Module ........................................ 10
      3.3.2 PreprocessEncryptor ..................................... 11
      3.3.3 PatternMatchingClient / PatternMatchingServer ....... 11
4 Analysis .......................................................... 12
  4.1 Testing Environment ...................................... 12
  4.2 Parameters .................................................. 13
  4.3 Measurements .............................................. 13
    4.3.1 Processing Time ...................................... 13
    4.3.2 Ciphertext Size ....................................... 14
    4.3.3 Query Time ........................................... 15

5 Conclusions .................................................... 17
  5.1 Current Status ............................................. 17
  5.2 Future Work ................................................ 17
    5.2.1 General Optimizations ............................... 17
    5.2.2 Extensibility ......................................... 18
    5.2.3 Threat Models ....................................... 19
  5.3 Lessons Learned .......................................... 19

Bibliography ..................................................... 20
List of Tables

4.1 Hardware Specifications of m3.xlarge .......................... 12
4.2 Measured Data Size vs Processing Time .......................... 13
4.3 Measured Data Size vs Processing Time .......................... 15
4.4 Query Length vs Query Time - $k = 10$ .......................... 16
4.5 Query Count vs Query Time - $m = 10$ .......................... 16
List of Figures

2.1 Client-Server Architecture ........................................... 3
4.1 Polynomial Fit of Encryption Time ..................................... 14
4.2 Polynomial Fit of Ciphertext Size .................................... 15
Chapter 1

Introduction

1.1 Background

Cloud services are becoming more and more popular for their ease-of-use, high availability, and elasticity [8]. Client-server architectures work particularly well with cloud services, as they allow users to offload processing and memory requirements to a remote server with high accessibility. These servers can exist anywhere on a connected network, limited only by the bandwidth and latency of the connection. In such a system, many thin clients with limited resources and a network connection can offload memory-intensive tasks to more centralized servers. This is particularly useful in data-driven systems where medical records, legal documents, or financial data can be stored in a single location.

1.2 Motivation

The problem with existing cloud services is security. Cloud computing platforms have a multitude of risks, some of which can be mitigated with encryption [6]. The server in a client-server architecture is untrusted, as it is typically in a remote location with an unknown or untrusted security model protecting it. Data stored on an untrusted cloud platform must be encrypted if it contains sensitive data such as medical records, legal documents, or financial data [5]. To prevent data breaches, the decryption key must be stored somewhere other than the untrusted server itself.

This presents a challenge for querying such a data set using traditional encryption methods. If the server cannot decrypt the data it holds, then the entire data set must be transferred
back to the client for each query, and decrypted locally. This makes the cloud service much less useful, as "thin" clients must be able to store and process the data themselves, rather than the scalable server.

1.3 Solution

One proposed solution to this problem is *queryable encryption*, a method of pattern matching on an encrypted ciphertext constructed by Melissa Chase and Emily Shen [2] [9]. Chase and Shen formalize queryable encryption, a system that provides a query function within rigidly defined security models. This system consists of client and server software, and the communication protocol they use to exchange data. The client software encrypts a known data set and transfers the encrypted text to the untrusted server. The client can also initiate an interactive query protocol that, in conjunction with the server, returns correct query results. The untrusted server stores said encrypted data and participate in an interactive query protocol to return search results with limited information leakage.

However, the additional security of pattern matching encryption comes at the cost of system performance.

1.4 Roadmap

This paper outlines the queryable encryption protocol at a high level, to provide sufficient background knowledge to understand the issue of performance. Next it describes an implementation of queryable encryption, along with design details found during the creation of the software. Lastly the results of this implementation are compared against the performance metrics presented in *Pattern Matching Encryption*.
Chapter 2

Design

The architecture for the queryable encryption system is relatively simple; one or more thin clients can contact an untrusted server (perhaps through a cloud service) and perform queries on the server’s encrypted data.

![Client-Server Architecture](image)

Figure 2.1: Client-Server Architecture

There are three discrete operations that can be performed on this architecture: **Key Generation**, **Encryption**, and **Search**. For each operation, functionality is described for both of the software components **Client** and **Server**. The contents of the **Communication** for relevant stages are also described at a high level.
2.1 Key Generation

Before any encryption, keys must be generated. This occurs only on the client, as only the client has the authority to use the key for decryption in this security model. No server activity or network communication is required for this stage. Key generation is implemented as a method of an Encryption module, described in 2.2.

2.1.1 Client

The key generation function requires a security parameter $\lambda$ and returns a cryptographic key $K$ of suitable security. In actuality, $K$ consists of 7 cryptographic keys but the design for this project abstracts those details. Keys are never distributed to the server, and are distributed to clients based on individual client architecture needs.

2.2 Encryption

Before any search queries can be performed, the client must process the plaintext and deliver it to the server in an encrypted form. The client encrypts a data string with an encryption key and sends it to the server. This step can be performed independently of any other encryption or search method.

Client encryption is implemented as a method of the Encryption module, to allow different encryption methods to be swapped into the rest of the system with little modification. Encryption modules include methods for key generation, encryption, and decryption. Key generation is conflated with encryption to support an extension encryption method with non-string encryption keys.

2.2.1 Client

To encrypt, the client must have a key $K$ and a message $M$ to encrypt. $K$ is created during the key generation stage and may be client-specific, shared between select clients, or match any other key distribution scheme that fits the user’s requirements. Multiple messages can
be encrypted with the same key, though this loosens the guarantees of plaintext security proved by Chase and Shen [2]. Broadly, the message $M$ is inserted into a suffix tree data structure and each substring of $M$ is encrypted with metadata about the location in $M$.

The encrypted substrings and associated metadata are added to a dictionary along with fictitious entries to prevent a malicious server from gaining knowledge of the text. More specific details of the encryption method are located in Section 4.3 Final Construction of Pattern Matching Encryption [2]. The resulting ciphertext $CT$ is sent to the server for storage and future queries.

The entire client processing stage is a method of an PreprocessEncryptor module, so separate preprocessing methods can be seamlessly inserted into the system. For example, another PreprocessEncryptor module could be utilized if the encryption system required native support for non-string data types.

### 2.2.2 Server

The server receives a ciphertext $CT$ from a client and stores it locally. As the ciphertext sent over the network, the communication protocol serializes the data structure using a shared serialize functionality that the client and server share. Depending on the requirements of the system, a server may support many clients and store $CT$ according to a unique client identifier.

Ciphertext storage is left up to the PatternMatchingServer module because the client has no knowledge of storage and retrieval methods. This abstraction allows PatternMatchingClients and PatternMatchingServers interactions to be completely interface-based. A PatternMatchingServer may be running on a different operating system, database schema, or programming language without affecting its ability to serve clients. The server does no further processing of the ciphertext, and can support search queries from this point forward.
2.2.3 Communication

The only contents transferred between client and server during the encryption stage is \( CT \), the encrypted message. This communication can be done over an arbitrary network medium. After the initial transfer the connection is terminated and no further communication takes place until a search query is initiated.

2.3 Search

The most common stage is the search, where a client queries for the existence and count of a string \( q \) as a substring of \( M \). There are three rounds of communication between the client and server. At the end of the three rounds, the client will know the count of substrings \( q \) that exist in \( M \). The client will also learn the indicies within \( M \) where \( q \) is located, if they exist.

2.3.1 Client

The client initiates the query by encrypting a query string \( q \) with a portion of the encryption key \( K \) and sends this to the server. After receiving the server’s response for each of the query rounds, the client verifies and decrypts the data it receives and continues to the next round.

At the end of the three rounds, the client learn exactly one of three things:

- No substrings of \( q \) exist in \( M \),
- The count of instances of \( q \) within \( M \) and their relative indices, or
- The server they are in contact with is malicious and has not provided honest output.

The third option is guaranteed to be returned if the server modifies the queries or ciphertext to a non-negligible degree. This ensures that the client will not unknowingly receive incorrect answers. More specific details of the search method are located in Section 4.3 Final Construction of Pattern Matching Encryption [2]. All functionality of the search
protocol is contained in a PatternMatchingClient module, to allow clients to be interoperable regardless of implementation details.

2.3.2 Server

The server receives a query from the client if a search is to be performed. To serve the query request for each step of the search stage, the server retrieves the ciphertext $CT$ from a database, disk, or memory. The server then performs lookups from $CT$ and returns a subset of the ciphertext based on the contents of the client query. More specific details of the search method are located in Section 4.3 Final Construction of Pattern Matching Encryption [2].

These retrieval and data manipulations can be done independently of other operations occurring on the individual server, and do not require any stored sessions or saved state. This allows query requests to be served regardless of previous or future operations, as long as $CT$ is present for the given query. For example, Client$_A$ could interact with independent servers Server$_X$, Server$_Y$, and Server$_Z$ during the three stages of a search, invisibly to a user. The stateless design was chosen to allow for greater scalability and redundancy of servers; a set of $n$ servers is fault-tolerant to $n - 1$ hardware failures, and can linearly scale to service $n$ times as many clients as a single server.

2.3.3 Communication

The client and server communicate during the three steps of the search stage. Six communication packets are transferred between the requesting client and a chosen server, a request and response for each of the three steps. The communication methods are the interface endpoints for the PatternMatchingClients and PatternMatchingServers. Implementing these interfaces and possessing the appropriate $CT$ is all that is required to act as a Pattern-MatchingServer.

The data is sent between client and server as a serialized byte stream, written for the
data types used. Using a custom SerializerFunctions module allows for minimal data overhead when packing objects specific to the encryption query process. Additional SerializerFunctions modules may be useful if the size of queried data sets becomes significantly larger. If more efficient serialization methods are implemented, the modular nature of SerializerFunctions would allow for version negotiation between client and server. As the SerializerFunctions is a module, any client and server can communicate if they share a SerializerFunctions module.
Chapter 3

Implementation

My project is an implementation and analysis of another’s system, and is the very definition of building off of the results of others. Without Chase and Shen’s excellently presented system, clear descriptions, and detailed background I would have never been able to implement their novel technique of queryable encryption. All of the credit for this novel queryable encryption scheme goes to them.

Because this project is purely an implementation and analysis, design details focus more on performance and less on security. I verified the security of the Pattern Matching Encryption model to the best of my ability, and focused my efforts on performance. Following is a description of modules touched upon in 2. Design, as well as a rationale for several design decisions made during implementation.

3.1 General Design

I chose C# to implement the queryable encryption system, as I needed a multi-platform language with support for highly component-based code. C# meets all those requirements and has extensive cryptography libraries as well.

I decided to develop on and target Linux systems, using the Mono framework to create cross-platform executables. The only officially supported and tested environment is a Linux system, but the multi-platform nature of C# and Mono would make migration to other systems straightforward.
3.2 Assumptions

For the purposes of data collection, and without loss of generality, I have made several assumptions that simplify my implementation. First, I assume that all data to be queried has a data type of string, and limit both the plaintext and query to string types. If this implementation is to be extended to queryable document that is not a string, I assume a mapping of document to strings is readily available.

Furthermore, the initial alphabet parseable by my implementation is limited to lowercase English alphabetic characters. Both the initial data set and query strings are stripped of non-alphabetic characters and lower-cased when parsed. This has the additional benefit of allowing case-insensitive search queries.

These assumptions were made to ease data collection and testing, and to speed up development time. None of the simplifications should have an impact on running time or ciphertext size, apart from the obviously larger suffix tree. Asymptotic behavior should be unaffected.

3.3 Modules

3.3.1 Encryption Module

The first encryption method I chose for the queryable encryption scheme is $\mathcal{E}_{\text{xor-auth}}$. The $\mathcal{E}_{\text{xor-auth}}$ method was constructed by Chase and Shen to meet the security criteria for their threat model. These criteria are necessary to prove correctness against malicious servers and satisfies a series of security games to test the indistinguishability of malicious servers and minimum information leakage.

The requirements for the encryption method in Shen’s system is that the encryption method be both authenticated and which-key-concealing. The authentication step is simple, by applying an encrypt-then-MAC step to the standard encryption scheme. If the Message Authentication Code does not match during decryption, no decryption is attempted. The which-key-concealing criteria is more complex, essentially asserting that from a set of
ciphertexts, there is no method of distinguishing which ciphertexts were encrypted with the same key, with non-negligible probability. Shen proved both the authenticated and which-key-concealing properties of $E_{\text{xor-auth}}$ encryption method, so it was an easy choice to implement the given encryption algorithm.

### 3.3.2 PreprocessEncryptor

The encryption process consists of tokenizing the plaintext (medical record, legal document, or financial record), inserting the plaintext into a suffix tree data structure, and encrypting elements of the suffix tree along with metadata to facilitate future searches. The resulting ciphertext can be transferred to the untrusted server for future queries. One crucial element of the linear encryption time is the construction of a suffix tree.

A naive approach to generating a suffix tree can have a time complexity of $O(n^2)$ where $n$ is the length of the input string. The Ukkonen algorithm [10] has a linear $O(n)$ complexity for the construction of a suffix tree. It is also on-line, meaning a suffix tree can be constructed on a stream of data without needing the entire text at once. As the asymptotic performance of the encryption and preprocessing stage is linear, the suffix tree construction must necessarily be linear as well.

I also used an open source suffix tree library text-indexing [4] provided by S. Costin. They provide a fast suffix tree construction, implementing the same on-line linear time Ukkonen algorithm.

### 3.3.3 PatternMatchingClient / PatternMatchingServer

For each query the clients wish to perform, there are three rounds of communication between client and server. These rounds are a negotiation of data agreement, including checks to narrow down correct query results, verify the server is not sending maliciously incorrect data, and finally return the valid results that match the initial query string. The entire search protocol occurs within the PatternMatchingClient and PatternMatchingServer modules.
Chapter 4

Analysis

4.1 Testing Environment

All tests are run on a Linux environment, as that is the platform it is implemented and designed to run on. To standardize the results, tests were run on hardware with known performance specifications[1], on the Amazon Elastic Compute Cloud (EC2) platform. I created an m3.xlarge instance type for each of the tests, which has specifications listed in Table 4.1.

I ran tests independently from one another to avoid previous tests’ memory footprints from affecting the performance of future tests.

Testing data was drawn largely from a Lorem ipsum filler text generator [11]. Lorem ipsum data was chosen because of its ease of collection and its similarity to English in word size and letter distribution.

As the experiments are heavily data-driven, the results are summarized with variable correlation and data visualization of the data involved. The graphed results are shown below and expanded upon.

<table>
<thead>
<tr>
<th>Table 4.1: Hardware Specifications of m3.xlarge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>Virtual CPUs</td>
</tr>
<tr>
<td>Clock Speed</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Disk</td>
</tr>
<tr>
<td>Network</td>
</tr>
</tbody>
</table>
4.2 Parameters

For each of the following metrics, $\lambda$ is the security parameter, a measure of the cryptographic strength of an encryption algorithm. $\lambda$ is selected for the data being encrypted, and is chosen to minimize information leakage for several cryptographic operations. The parameter $n$ is the length of the plaintext data that is originally encrypted. $m$ is the length of a query string that occurs as a substring of the data $k$ times.

4.3 Measurements

4.3.1 Processing Time

For a variety of plaintext sizes I measured the total time spent processing the data, including constructing a suffix tree, collecting metadata, and encryption time. This data can be compared against the $O(n)$ expected runtime of encrypting the data. However, I was unable to match the $O(n)$ encryption time presented in Pattern Matching Encryption. My results plotted in Table 4.2, and asymptotic running time is estimated in Figure 4.1.

<table>
<thead>
<tr>
<th>Data size (bytes)</th>
<th>Processing time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>165.9</td>
</tr>
<tr>
<td>50</td>
<td>1008.2</td>
</tr>
<tr>
<td>100</td>
<td>3659.3</td>
</tr>
<tr>
<td>200</td>
<td>12897.2</td>
</tr>
<tr>
<td>300</td>
<td>28849.6</td>
</tr>
<tr>
<td>400</td>
<td>50221.9</td>
</tr>
<tr>
<td>500</td>
<td>77633.9</td>
</tr>
<tr>
<td>600</td>
<td>110004.9</td>
</tr>
<tr>
<td>700</td>
<td>148096.9</td>
</tr>
<tr>
<td>800</td>
<td>194293.3</td>
</tr>
<tr>
<td>900</td>
<td>245863.5</td>
</tr>
<tr>
<td>1000</td>
<td>304427.2</td>
</tr>
</tbody>
</table>

The least-squares best fit curve polynomial is represented by the function $0.29234n^2 +$
5.78232n + 117.3, with an $R^2$ goodness-of-fit coefficient of 0.999975. The extremely high coefficient of determination, along with the obvious upwards slant on the graph, makes it clear that my current implementation does not meet the linear time encryption efficiency proposed by Chase and Shen. The cause of this is explored in Section 5.2.1. This function has a relatively small highest-order constant (0.29234), but can not compete with the linear time method at larger data sizes.

### 4.3.2 Ciphertext Size

I also ran the processing stage for a variety of security parameters, to validate the $O(n\lambda)$ ciphertext size prediction. The results of the data collection are presented in 4.3 and asymptotic impact is measured in 4.2.

Note that these ciphertext size estimates are taken from serialized ciphertexts, and does not reflect the actual data size stored on a server. I made the decision to serialize, (and in some cases re-serialize) the data to provide consistent measurements for this specific metric. Improvements to the SerializerFunctions module or more space-efficient data structures would have a dramatic impact on the ciphertext sizes reflected here.

As mentioned above, the encryption method is impacted by some performance issues detailed above and in Section 5.2.1. Because of this, the ciphertext size function is also limited to $O(n^2)$ space requirements until the corresponding optimizations \(^\text{1}\) are implemented. As currently implemented, the ciphertext size is measured by $4.16351n^2 +$

---

\(^1\)Levin’s trick and efficient batched PRFs
Table 4.3: Measured Data Size vs Processing Time

<table>
<thead>
<tr>
<th>Input Size (bytes)</th>
<th>Ciphertext Size (megabytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8261</td>
</tr>
<tr>
<td>50</td>
<td>29667</td>
</tr>
<tr>
<td>100</td>
<td>69292</td>
</tr>
<tr>
<td>200</td>
<td>201695</td>
</tr>
<tr>
<td>300</td>
<td>465989</td>
</tr>
<tr>
<td>400</td>
<td>808210</td>
</tr>
<tr>
<td>500</td>
<td>1194233</td>
</tr>
<tr>
<td>600</td>
<td>1578254</td>
</tr>
<tr>
<td>700</td>
<td>2392489</td>
</tr>
<tr>
<td>800</td>
<td>2845028</td>
</tr>
<tr>
<td>900</td>
<td>3894079</td>
</tr>
<tr>
<td>1000</td>
<td>4359713</td>
</tr>
</tbody>
</table>

343.97n – 8793.84. This fitness function is accurate to a coefficient of determination of $R^2 = 0.995999$.

Figure 4.2: Polynomial Fit of Ciphertext Size

4.3.3 Query Time

Lastly I ran data string queries, with known number of substrings in the ciphertext data, and collected data on run time versus number of results returned. Chase and Shen provide a method of querying search strings in $O(m+k)$ time, within three rounds of communication. When testing the time taken per number of results, I constructed simple repeated strings and searched for substrings with known counts and lengths. I performed two measurements,
alternately fixing the query length \( m \) (Table 4.5) and the query count \( k \) (Table 4.4) to 10, and measuring the impact of changing the other variable. Both the query string length and query count have a linear impact on the query time. Therefore my implementation of the search method conforms to the presented \( \mathcal{O}(m + k) \).

Table 4.4: Query Length vs Query Time - \( k = 10 \)

<table>
<thead>
<tr>
<th>Search String Size (bytes)</th>
<th>Query Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>22.5</td>
</tr>
<tr>
<td>20</td>
<td>29.1</td>
</tr>
<tr>
<td>30</td>
<td>34.9</td>
</tr>
<tr>
<td>40</td>
<td>41.8</td>
</tr>
<tr>
<td>50</td>
<td>48.1</td>
</tr>
<tr>
<td>60</td>
<td>55.5</td>
</tr>
<tr>
<td>70</td>
<td>63.6</td>
</tr>
</tbody>
</table>

Table 4.5: Query Count vs Query Time - \( m = 10 \)

<table>
<thead>
<tr>
<th>Search String Count</th>
<th>Query Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>22.2</td>
</tr>
<tr>
<td>20</td>
<td>23.1</td>
</tr>
<tr>
<td>30</td>
<td>24.8</td>
</tr>
<tr>
<td>40</td>
<td>25.1</td>
</tr>
<tr>
<td>50</td>
<td>27.2</td>
</tr>
<tr>
<td>60</td>
<td>26.8</td>
</tr>
<tr>
<td>70</td>
<td>30.3</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusions

The queryable encryption model presented is a very interesting system with a great deal of potential. The initial construction and definitions[2] provide a foundation for future research into the subject. I hope that my contributions to this topic are found useful in the computer science community.

5.1 Current Status

The queryable encryption implementation in this project is fully functional. There is plenty of room for improvement, iterated below, but the base functionality is working.

5.2 Future Work

There are several areas in which the given implementation could be improved. Two cryptographic shortcuts could speed up encryption performance by orders of magnitude. Additional functionality can be added using the modular design presented in 3.

5.2.1 General Optimizations

The most significant future work that could be applied to my implementation would be performance optimizations. One notable optimization would be the introduction of ”Levin’s trick” to increase performance in the Encryption and client Search stages. When applying a pseudorandom function family to a large input, the input can be hashed with a specific
set of hash functions\(^1\) to speed up computation without sacrificing security [7]. This optimization, also known as Levin’s trick, would give lower-order speedups but no asymptotic improvements.

Another notable addition could be the **Efficient batch implementation of PRFs** described in 4.4 Efficiency Pattern Matching Encryption. By applying a (polynomial) rolling hash function, the initial client encryption can further increase the encryption performance. Rolling hash functions on a set of strings with many shared prefixes (such as the entire ciphertext in a suffix tree) requires much less computation than a set of distinct strings [3]. This optimization in conjunction with Levin’s trick would give an asymptotic improvement to the encryption speed, reducing this implementation’s encryption from \(O(n^2)\) to \(O(n)\) when hashing all suffix tree entries.

### 5.2.2 Extensibility

The software implementation described in this paper was designed to be easily extensible to added functionality. The module system permits simple replacement of a type of function including Encryption, SerializerFunctions, PatternMatchingClient, PatternMatchingServer, and a PreprocessEncryptor. Further work could be spent on optimizing one or several of these modules to increase performance, lower data overhead, or add native cross-platform support. This also allows for the seamless addition of new features that were not included due to constraints on development time.

Potential new features could include alternative encryption methods or database wrappers for the server’s ciphertext retrieval. The ability for a single server to support multiple users or multiple ciphertexts per user would give a large boost to usability in real world systems. If a specific module improvement breaks backwards compatibility, an additional communication handshake should precede an Encryption or Search method call. Once the client and server negotiate which modules they support and prefer, communication can resume as described in 2. Design.

\(^1\)\(\epsilon\)-almost-universal, to which the Rabin-Karp, polynomial, and rolling hash functions generally belong
5.2.3 Threat Models

Another interesting piece of future work is presented directly by Chase and Shen; the threat modeling of honest, honest-but-curious, and malicious servers. *Pattern Matching Encryption* details the leakage of information that is inherent in delivering results, and a formal definition for this leakage is provided. However, this formal definition only proves that the information leaked is *negligible* with regard to the security parameter $\lambda$. Exploring the real world impact of negligible data leakage, under normal conditions and reasonable $\lambda$ values is an avenue of future research.

5.3 Lessons Learned

I failed to meet all of the time and space performances detailed by Shen and Chase in my initial implementation. I underestimated the performance impact of some shortcuts presented in *Pattern Matching Encryption*, and took a significant performance hit because of it. While I made space and time saving optimizations such as a linear suffix tree construction, I was unable to implement all provided methods in the time period. Specifically, using Levin’s trick and a polynomial rolling hash function (Detailed in 5.2.1) would allow the encryption process to be done in linear time.

I did implement an easily extensible and modular software project I plan on maintaining in the future. The modular aspect of functions in my client and server provide an easy test bed for implementing new encryption or data compression methods. The testing framework and recorded results will also make it very easy to determine the efficiency of any new modules as they are developed.
Bibliography


