# Secure Remote Storage Using Oblivious RAM

Giovanni Malloy

Mentors: Georgios Kellaris, Kobbi Nissim

August 11, 2016

**Abstract**

Oblivious RAM (ORAM) is a protocol that allows a user to access the data she stored on a server while concealing the access pattern. This work attempts to make improvements to the efficiency of Path ORAM, a specific implementation of ORAM when answering range queries. The efficiency is measured as the time to retrieve a set of records. We began our exploration with an off-the-shelf implementation for which the time to retrieve 1,000 records was roughly 50 seconds, and the improvements described herein brought that number down to roughly 7 seconds. This was achieved by streamlining existing code, switching to a simpler database management system, and integrating the use of threading. Ultimately, this work presents a feasible foundation to continue to improve the algorithm for real-world applications.

## 1 Introduction

Assume a scenario where a bank has a database of records where rows represent each client record and columns represent different attributes (i.e. name, age, account information, etc.). A bank can retrieve information about the database by issuing queries on a specific attribute. For example, a point query will retrieve records of individuals age 30, while range queries retrieve the records of those aged 20 to 30. A typical practice is to outsource such databases to a third party, while still allowing efficient querying. However, a third party cannot always be trusted with sensitive information, thus encryption is used. In particular, the bank generates a key and encrypts the data along with an encrypted data structure to efficiently query the outsourced database, attempting to balance system efficiency and security.

Both access pattern and communication volume leaks are possible on these systems [3]. Access pattern leakage allows an adversary to distinguish among records and queries, although the adversary is unable to read the encrypted records or queries. Communication volume leakage only allows an adversary to see the size of the set of records retrieved by a

query. Previous work has shown that an adversary can effectively reconstruct the distribution of records on the indexed attribute domain using either the access pattern or communication volume. Thus, privacy is compromised. The solution to this issue presents itself as two-fold. First, an advanced cryptographic technique, such as oblivious RAM, can be used to disguise access patterns. Similarly, differential privacy is one way to hide the communication volume. This work focuses on exploring the practicality of combining ORAM and differential privacy. We develop and analyze experiments to enhance the understanding of Path ORAM, a specific implementation of oblivious RAM. In this work, we address the question: *how to use Oblivious RAM as a practical solution to concealing the access pattern leakage?*

# 2 Background

## 2.1 Outsourced Database Systems

A database is a standardized collection of data that stores records (rows) and their attributes (columns). A user may retrieve records with specific attributes by querying the database. There are two types of queries relevant to this report: range queries and point queries. Assume a total ordering on the attribute domain $D$ and that $D = 1, ..., N$ for some $N \in \mathbb{N}$. A query $q_{[a,b]}$ that is associated with an interval $[a, b]$ for $1 \leq a \leq b \leq N$, and asks for all the records with attribute values within the range $[a, b]$ is a range query. Point queries degenerate from range queries when $a = b$. For a certain subset of attributes, indices are generated and assigned. Indices act as pointers to different records in a database in order to allow efficient querying.

Outsourced database systems are commonly used in a variety of industries. Some implementations rely on deterministic or order preserving encryption to protect the data that they store. The user generates a key to encrypt the records and indices. The encrypted indices form a data structure that allows the user to efficiently query the outsourced database. The communication efficiency revolves around minimizing the bandwidth cost to both the server and client and the time to retrieve a set of records.

## 2.2 Attacks

After observing a large number of query retrievals, an adversary can aggregate the access pattern and communication volume leaks and begin to reconstruct the attribute domain of the records [3]. An attribute domain of records is the interval of values that corresponds to that attribute. The records are distributed over the domain, and how they are distributed over the domain is private information. Privacy is breached and an attack is successful when the adversary can reconstruct this information [3]. Most current database systems encrypt and store each record only once, and every time that record is retrieved, the same ciphertext is retrieved. The adversary can efficiently reconstruct the distribution of records over the indexed attribute domain with repeated observations, as shown in previous work. For database systems that frequently re-encrypt a subset of records, it is no longer possible

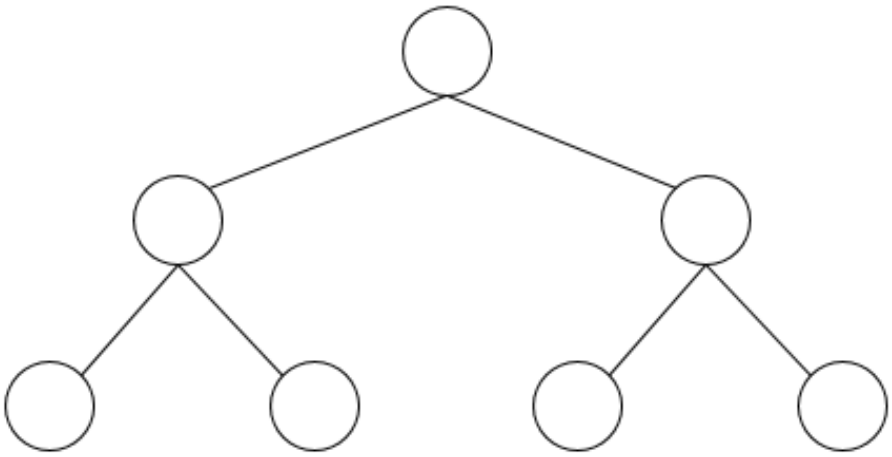

Figure 1: A binary tree structure with $L = 4$ leaves and a height of $H = 3$.

for an adversary to efficiently distinguish between distinct records. However, there is still a communication volume leakage which reveals the bandwidth of both the retrieved records. By observing the size of the set of records retrieved after each query, the adversary is still able to reconstruct the distribution of records over the indexed attribute domain thereby breaching privacy.

## 2.3 Path ORAM

Oblivious RAM is a method of encryption that hides the access pattern leakage by continuously reshuffling and reorganizing the data after each query. The specific implementation of ORAM discussed in this work is Path ORAM. This work was chosen because the implementation is simpler, more practical, asymptotically better than the many known ORAM schemes for small client storage, and the code is readily available [1]. The Path ORAM algorithm used in this work is defined with a binary tree (Figure 1) where, given $n$ records, there will be a number of leaves, $L = 2^{\lceil \log_2 n \rceil}$, and a height of $H = \lceil \log_2(n+1) \rceil$ [1, 4]. The origin node in the tree is called the root. Each node will have two children nodes until the construction of the tree is completed at the leaves. Each record is defined by an encrypted path on the tree from the root to a leaf. After any record or set of records is retrieved, the paths of the records are immediately re-encrypted and replace the previous ones.

Each node on the tree is assigned a bucket, and each bucket can hold a fixed number of blocks, $Z$. The user of the database holds a map of the encrypted paths of each record. The server then retrieves the records corresponding to the user's position map. The server sends the encrypted records to the database, re-encrypts and replaces the paths of those records, and updates the user's map. The user then decrypts the retrieved records.

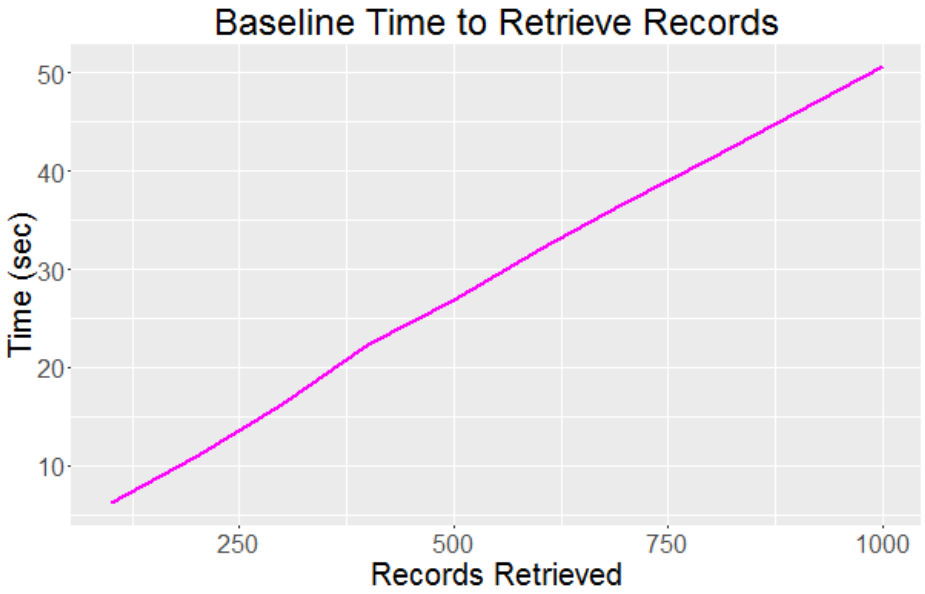


Figure 2: Record retrieval time for off-the-shelf implementation of Path ORAM.

# 3 Experiments and Results

## 3.1 Setup

The experiments were run on a Dell Inspiron 5558 with an Intel Core i7-5500 CPU @ 2.40 GHz processor and 8.00 GB of RAM. The machine runs Windows 7 Professional (64-bit). The Java code was run using NetBeans IDE 8.1 communicating with a PostgreSQL and Redis database management system. Both the client and the DBMS server were hosted locally. The time for the experiments was measured by comparing the current time, as measured by *System.currentTimeMillis()*, immediately before starting the threads and the current time after the threads finish running.

## 3.2 Baseline

The baseline scenario was evaluated to determine the affect on the record retrieval time. Figure 2 shows the time to retrieve 1,000 records before improvements were implemented at 50.680 seconds. This experiment was run without threading using PostgreSQL as the database management system before the code was streamlined.

Path ORAM inherently involves some storage inefficiency because of its binary tree structure. Each node of the tree reserves some space in memory. Given that there are fewer records than nodes, using a PAth ORAM tree structure will require more storage space than simply storing the records alone. Therefore, this work develops methods to minimize the high communication time. These methods can be enumerated as such:

1. Streamline Code

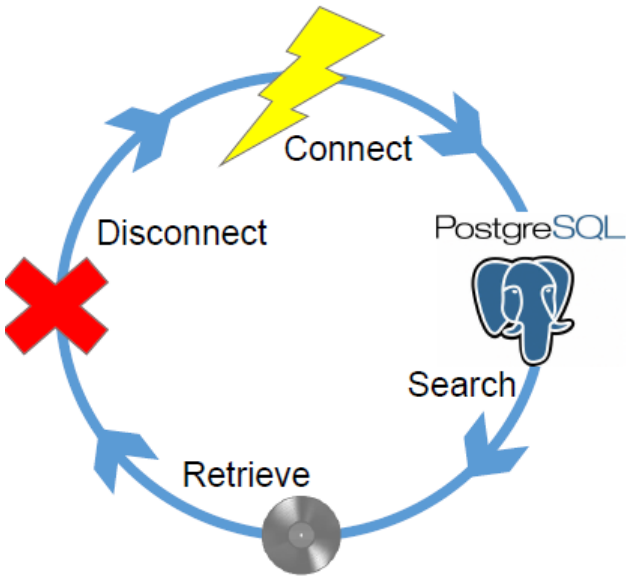


Figure 3: The record retrieval process of the off-the-shelf implementation with PostgreSQL.

2. Database Management System (DBMS) change
3. Threading

## 3.3 Streamlining Code

Step 1, streamlining code, was primarily dedicated to addressing process inefficiencies. The primary bottleneck at this phase was the act of connecting and disconnecting from the database. Initially, the Path ORAM implementation would retrieve a record by connecting to the database held on the local DBMS, searching the database for the appropriate record, returning that record, and finally disconnecting from the database. Yet, the implementation was time intensive for large sets of record retrievals because for every record retrieved, the process of connecting and disconnecting would repeat, as shown in Figure 3. To remedy this inefficiency, we altered the existing code such that when a client wants to retrieve a set of records, they will connect to the database only at the very beginning of the record retrieval and disconnect at the very end (Figure 4). The resulting time savings cut communication time by roughly two-thirds.

## 3.4 Database Management System

After eliminating inefficiencies in the code, the next large bottleneck, as identified in step 2, was the DBMS itself. The two different database management systems compared in this work are both locally-hosted. The first is PostgreSQL, which is an object-relational database management system [5]. The second is a far simpler database system, Redis. Redis is an open

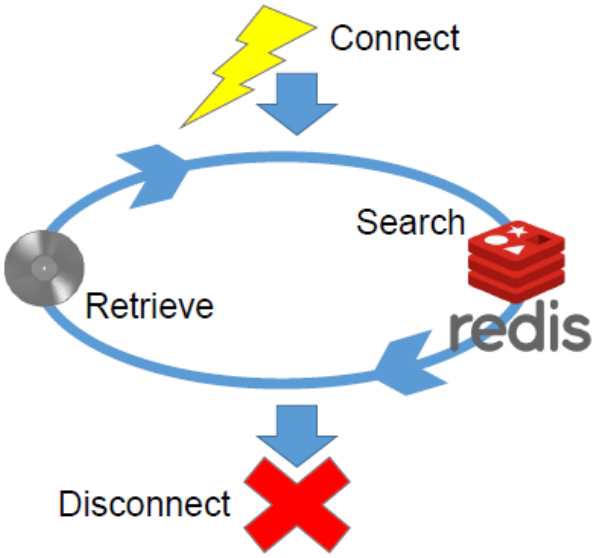


Figure 4: The record retrieval process after streamlining code and switching to Redis.

source key-value database management system [2]. For simply retrieving records, Redis is sufficient to manage the Path ORAM structure despite its lower functionality. This resulted in an additional time savings of about one half of the already reduced record retrieval time.

### 3.5 Threading in Multiple ORAMs

Threading, step 3, is a way for a computer program to run multiple tasks in parallel. Effectively, this manifests itself in Path ORAM as multiple clients querying the database and retrieving records at the same time. In order to implement threading in Path ORAM, this work investigates a multiple ORAM structure. In this case there were ten Path ORAM binary trees of 131,072 records each stored on the database. Thus, the overall database contains 1,310,720 records. Each tree was stored in a different location on the Redis database management system. Each of the ten threads this work creates only queries one of the trees and locations in memory. This ensures that we can experiment with asynchronous threading while still providing privacy guarantees and without any risk to data corruption.

There were twenty-four different experiments run to test the efficiency of record retrieval time. This work also compares compares the retrieval of differently sized sets of records and number of threads. The size of the set of records retrieved was 1,000 records, 10,000 records, or 100,000 records for 1, 2, 5, and 10 threads.

When implementing threading, the amount of time to retrieve 1,000 records drops dramatically. The results of each iteration of the experiment are detailed in Appendix I (Section 6). Depending on the combination of the number of threads and the type of threading used, the time varies between 5.6504 seconds and 7.1960 seconds. Moreover, the time to retrieve 10,000 records varies from 17.5990 seconds to 34.5768 seconds, and the time to retrieve 100,000 records varies from 158.7176 seconds to 284.0788 seconds. By reducing the number

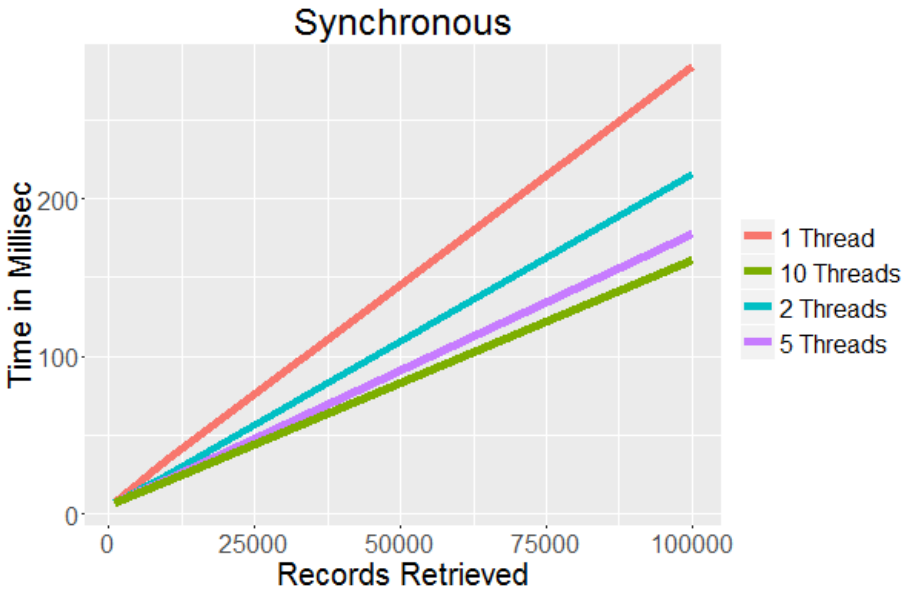


Figure 5: Record retrieval times for synchronous threads.

of records retrieved from 100,000 to 10,000, the time to retrieve the records is decreased by roughly an order of magnitude. This is not the case when going from 10,000 to 1,000 records, though.

The relationship between record retrieval time and the number of threads can be seen in Figures 5 and 6. In general, a greater number of threads will lead to a faster retrieval of a set of records of the same size (Figure 7). This is expected, as a greater number of threads allows a greater number of tasks to be performed simultaneously. The graph also shows that the overall time savings of using a greater number of threads increases as the size of the set of records to be retrieved gets larger. For small numbers of records retrieved there is very little difference in record retrieval time because the act of creating and managing additional threads nullifies the time savings of having those threads.

The relationship between record retrieval time and the type of threading (i.e. synchronous or asynchronous) can be seen in Figures 8 and 9. Asynchronous threading is faster than synchronous threading for the same number of threads (Figure 10). Asynchronous threads should be faster than synchronous ones because they do not wait on each other to finish a task in a location. However, for both synchronous and asynchronous threads, the marginal benefit of increasing the number of threads shrinks as the number of threads used grows larger. The decreasing marginal benefit of more threads is likely do to hardware constraints on the number of processes running in parallel of the local machine.

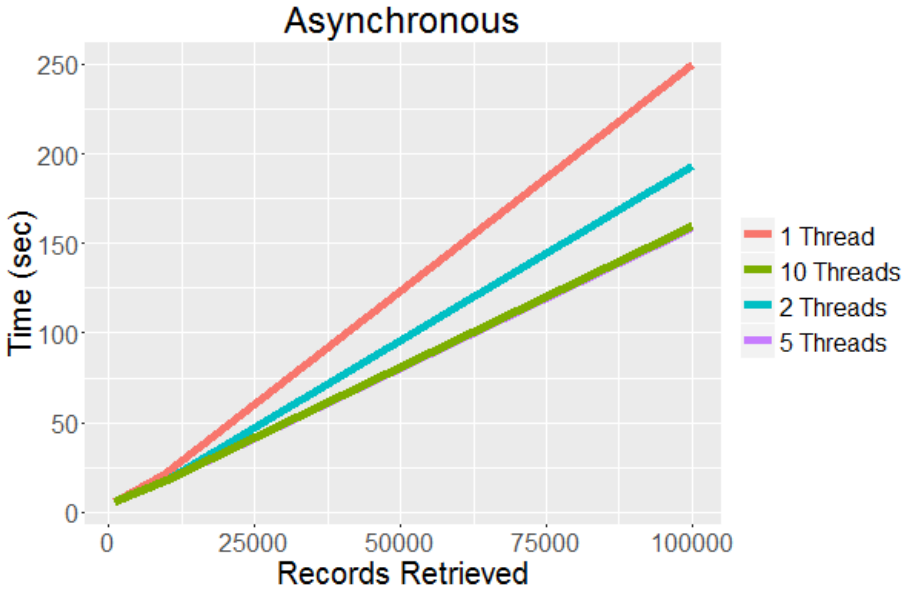


Figure 6: Record retrieval times for asynchronous threads.

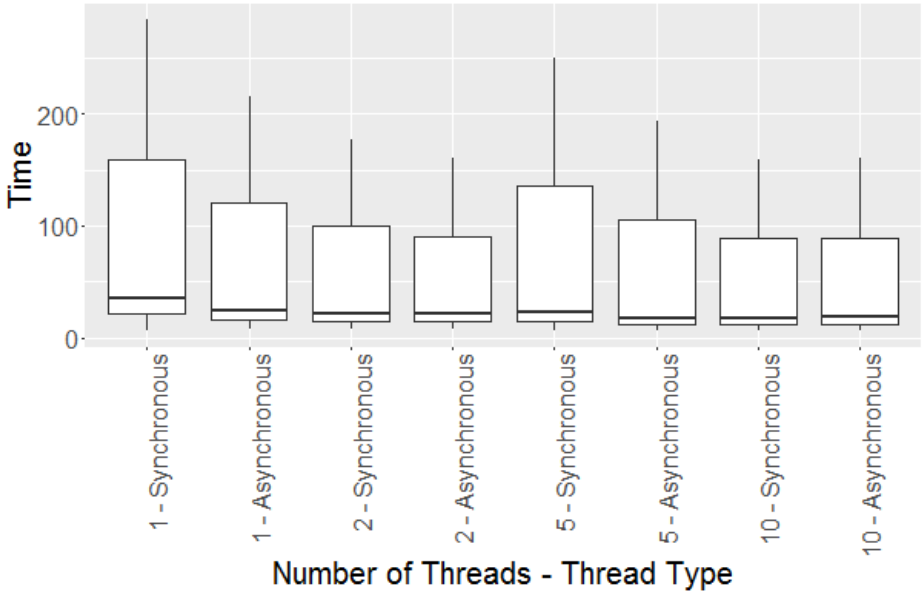


Figure 7: Record retrieval times for different numbers of threads for synchronous and asynchronous threading.

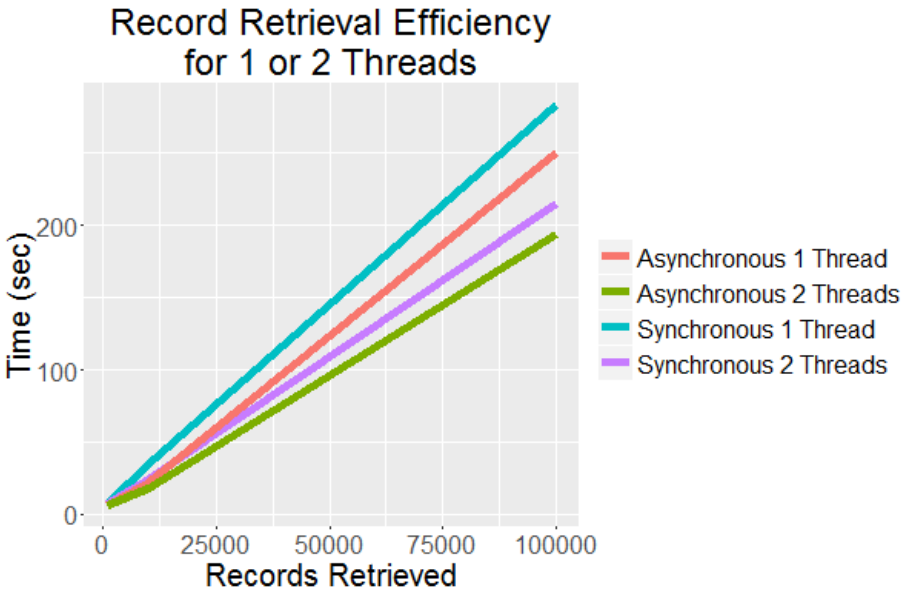


Figure 8: Record retrieval times for 1 and 2 threads.

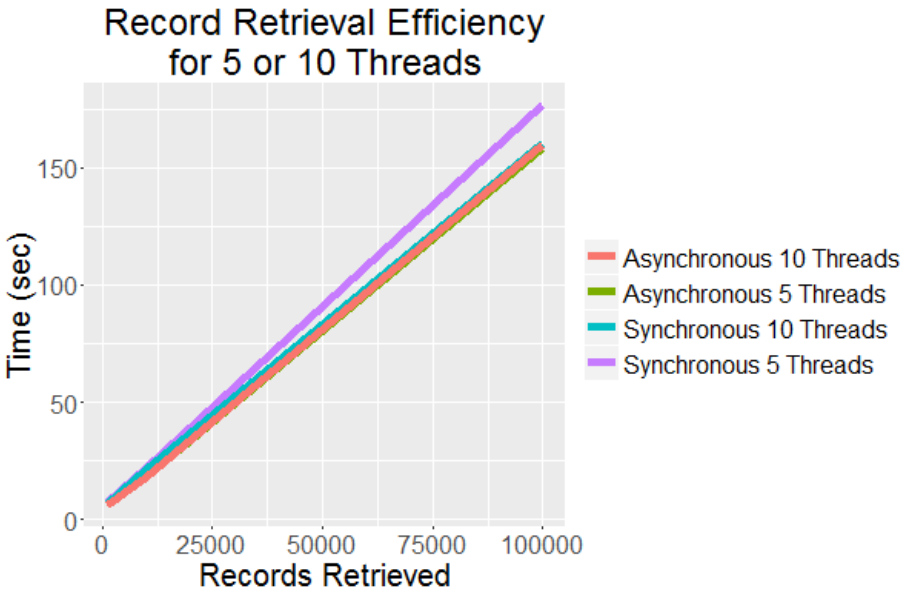


Figure 9: Record retrieval times for 5 and 10 threads.

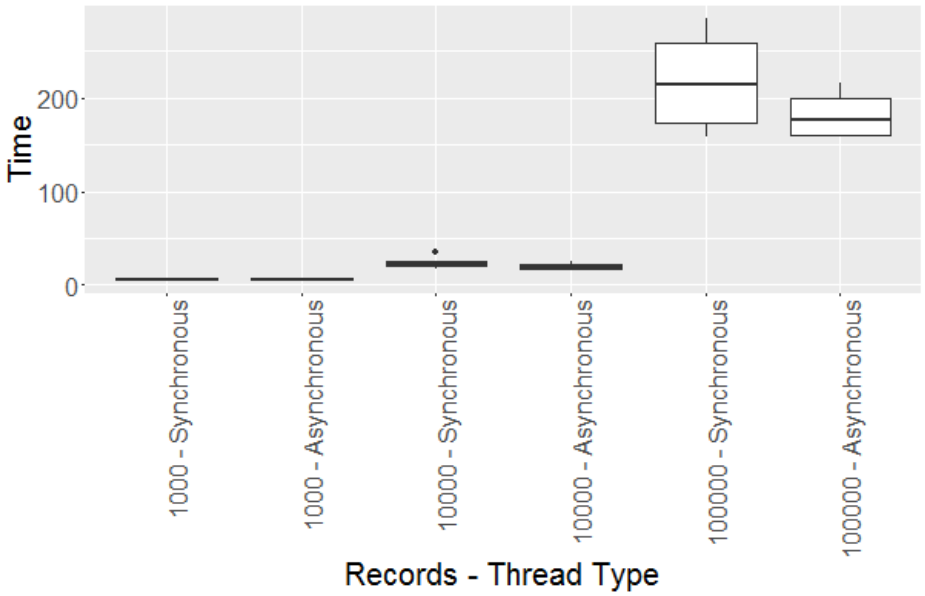


Figure 10: Record retrieval times for different numbers of records for synchronous and asynchronous threading.

# 4 Conclusions

Overall, Path ORAM as an implementation of Oblivious RAM shows promise and progress towards real-life practical usage as a method to conceal the access pattern leakage. This work presents improvements that brought down the record retrieval time by almost an order of magnitude. A simpler database management system provided improvements to the communication efficiency without sacrificing any functionality. The multiple tree Path ORAM approach presented in this work was able to allow for threading in an algorithm that otherwise would not. For best results as the size of the set of records to be retrieved grows, asynchronous threading with a high number of threads should be utilized. Ideally, the time can further be decreased from a few seconds to a few milliseconds for real-world applications.

# 5 Future Work

The goal of this stream of research is to make Path ORAM an efficient and practical method to conceal access pattern leakage. However, this work only utilized local database management systems. Thus, in the future, it will be necessary to consider the communication time between machines on the same network or from a machine to storage in the cloud. The multiple ORAM structures could even be distributed over multiple servers. Furthermore, in order to achieve a greater level of privacy, the communication volume leakage should be concealed. This work imagines that differential privacy will be used to do so by padding the results of queries with duplicate records. There is now a foundation for additional improvements, such

as those described above, to be made to Path ORAM.

# 6 Appendix I

| Thread Type | Number of Threads | Records Retrieved | Average Time (sec) |
|---|---|---|---|
| Synchronous | 1 | 1,000 | 6.8476 |
| Synchronous | 1 | 10,000 | 34.5768 |
| Synchronous | 1 | 100,000 | 284.0788 |
| Synchronous | 2 | 1,000 | 7.1960 |
| Synchronous | 2 | 10,000 | 24.6718 |
| Synchronous | 2 | 100,000 | 215.8094 |
| Synchronous | 5 | 1,000 | 7.0202 |
| Synchronous | 5 | 10,000 | 21.6820 |
| Synchronous | 5 | 100,000 | 177.1836 |
| Synchronous | 10 | 1,000 | 7.1960 |
| Synchronous | 10 | 10,000 | 24.6718 |
| Synchronous | 10 | 100,000 | 160.6152 |
| Asynchronous | 1 | 1,000 | 5.9098 |
| Asynchronous | 1 | 10,000 | 22.4572 |
| Asynchronous | 1 | 100,000 | 250.0488 |
| Asynchronous | 2 | 1,000 | 5.6504 |
| Asynchronous | 2 | 10,000 | 17.5990 |

| | | | |
|---|---|---|---|
| Asynchronous | 2 | 100,000 | 193.4388 |
| Asynchronous | 5 | 1,000 | 5.9034 |
| Asynchronous | 5 | 10,000 | 17.6728 |
| Asynchronous | 5 | 100,000 | 158.7176 |
| Asynchronous | 10 | 1,000 | 5.6868 |
| Asynchronous | 10 | 10,000 | 17.9284 |
| Asynchronous | 10 | 100,000 | 159.9910 |

Table 1: Results of threading experiments' effects on record retrieval efficiency

# References


[1] V. Bindschaedler, M. Naveed, X. Pan, X. F. Wang, and Y. Huang. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. 2015.

[2] J. L. Carlson. Redis in action. *Manning Publications*, 2013.

[3] G. Kellairs, G. Kollios, K. Nissim, and A. O'Neill. Generic attacks on secure outsourced databases. *CCS*, 2016.

[4] E. Stefanov, M. van Dijk, E. Shi, T. H. Chan, Fletcher C., L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. 2012.

[5] M. Stonebaker and L. A. Rowe. The design of postgres. *SIGMOD '86 Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 340 – 355, 1986.