Journal of Theoretical and Applied Information Technology
15th October 2024. Vol.102. No. 19
© Little Lion Scientific

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

OPTIMIZING ENCRYPTED DATA RETRIEVAL: PARALLEL SEARCH TECHNIQUES WITH HASH TABLE INDEXING

WAFA ALDABABAT[1]

[1]Instructor, SE Dept., Prince Abdullah bin Ghazi Faculty of Information and Communication Technology, Al-Balqa Applied University, Jordan
E-mail: [1]w.dababat@bau.edu.jo

ID 55516 Submission	Editorial Screening	Conditional Acceptance	Final Revision Acceptance
07-09-2024	07-09-2024	24-09-2024	09-10-2024

ABSTRACT

String matching is one of the fundamental operations in many applications, it is more challenging today to handle string searching while maintaining the security and privacy of data. In this paper a mechanism for searching in encrypted data is presented, this is done by using index-based searching on encrypted data, with using hash tables as index for searching and encrypting both the keys and values stored in the hash table for security. Also, parallel searching of multiple patterns is used in our implementation by using multiple threads is added to enhance efficiency. The results show that the approach is feasible and has a potential application in secure data retrieval systems, because it improves the searching performance while maintaining the security and privacy of data.

Keywords: *String matching, Encryption, Parallel search, Security, Hash tables, Index-Based searching.*

1. INTRODUCTION

We live in a world of big data, and there is a particularly press need for real-time secured data processing in fields like finance, healthcare, education, etc. Cloud computing is used to solve the problem of storing massive amounts of data and accessing it efficiently , however, data is stored on remote servers, introducing the problem of security and privacy of stored data. If the data stored on remote servers is not protected in a proper way, it will become vulnerable to unauthorized access.

One solution for preserving the privacy of remotely stored data is to encrypt the data before uploading it to the remote server. When we need to search the encrypted data, traditional searching algorithms will decrypt the stored data before searching it, making the data at risk of unauthorized access. So, to maintain its security, we need a way to search the data without decrypting it.

There are many solutions for searching encrypted data while maintaining its privacy, some of these methods will be discussed in Section 2. One of these methods is using Search Index Structures [9, 10, 11].

In Search Index Structures, indexes can be used for retrieving and searching the data efficiently and securely. Using this technique, we have two parts of storage; the data itself and the search index that contains the search keys. Traditional data securing methods have focused on encrypting the stored data to protect it, this will leave the index under danger of potential security attacks. To eliminate such attacks, both the stored data and the search index need to be encrypted.

In this paper, we introduce a solution that is based on search index structures. In our solution, a hash table is used as search index, the values in the hash table are encrypted, the keys used for searching are also encrypted yielding a more robust and secure search.

The efficiency of search is also considered in our solution, searching encrypted data can be time consuming especially when the data set is large, and to reduce the search time we use parallel searching. By distributing the searching load, multiple patterns can be searched simultaneously by using multi-threading, which will significantly speed up the searching process

Advanced Encryption Standard (AES) is considered a secured standard for encryption. In our implementation, we used the BouncyCastle Library in Java with AES for encryption, together with multi-threading in Java, which results in an application where searching can be efficient and secure at the same time.

In summary, we presented a method for enhancing the security of searching encrypted data by using encrypted hash tables as a search index, the efficiency of search in increased by using multi-threading to search multiple patterns and we have experimentally evaluated our work to show its effectiveness.

The rest of this paper is organized as follows: related work is discussed in section 2, our technique including implementation of hash table encryption and parallel search is presented in section 3, experimental results and discussion are presented in section 4 and in section 5 we present the conclusion and potential future developments on our work.

2. RELATED WORK

The work in the field of searching encrypted data has presented many solutions to search the data securely and efficiently, in this section we present some of these key solutions to make a context for our work.

2.1. Homomorphic Encryption

In homomorphic encryption, the search is done directly on encrypted data and the result of search is also encrypted, when the result is decrypted, it will give the same output as if the search was done on the original data without encryption. A full homomorphic encryption FHE is presented in [1]. The limitation of this solution is its high overhead making it inappropriate for real-time systems. Enhancements on FHE have been presented in [2, 3].

2.2. Searchable Symmetric Encryption

Searchable Symmetric Encryption SSE is appropriate for cloud computing, it allows the search to be done on files while they stay encrypted. [4, 5] have presented secured and efficient search based on SSE. However, it is not always scalable for large datasets.

2.3. Oblivious RAM

Oblivious RAM (ORAM) methodology is based on hiding access patterns to encrypted data, in this way unauthorized access cannot be done to these patterns, eliminating the possibility of inferring any information about the data [6]. The limitation is also the associated computational overhead. The work in [7, 8] addresses this limitation by presenting solutions to reduce the latency and overhead to make these protocols more practical.

2.4. Secure Index Structures

A lot of work have been presented based on secure Index Structures. A secure sequential scan technique is proposed in [9], this technique is not very efficient when the data set is large. Bloom filters are used to build indexes as presented in [10], but this solution can give false results affecting the accuracy of the search. A B+ Tree index is presented in [11] for fast and efficient search over encrypted data.

2.5. Encrypted Trie Structures

Encrypted Trie Structures have been presented in [12], and shown to be efficient for queries on encrypted data. In [13], dynamic data is handled efficiently by schemes based on Trie Structures. Efficiency can be compromised in these techniques because they depend on complex encryption and decryption operations.

Based on these foundations, we present a secure method for searching in encrypted data, by using hash table based index and encrypting both the values stored in the hash table index and the keys used for searching, which will result in a secure search mechanism. To insure performance is maintained with security, parallel search using multi-threading is used in our solution to handle the search of multiple patterns at the same time. The presented solution combines security with efficiency and simplicity, resulting in a suitable technique that can be easily adapted by real world applications.

3. METHODOLOGY OF WORK

3.1. Technique of Multi-Threaded Secure Search

The technique presented in this paper relies on two main components: secure hash table and multi-threaded search. The encrypted data itself is stored in a separate file, the search is done using the keys in the hash table, and then the retrieved value from the index is used to directly access the encrypted data in the file. The search is done in parallel using multiple threads to enhance performance of the search.

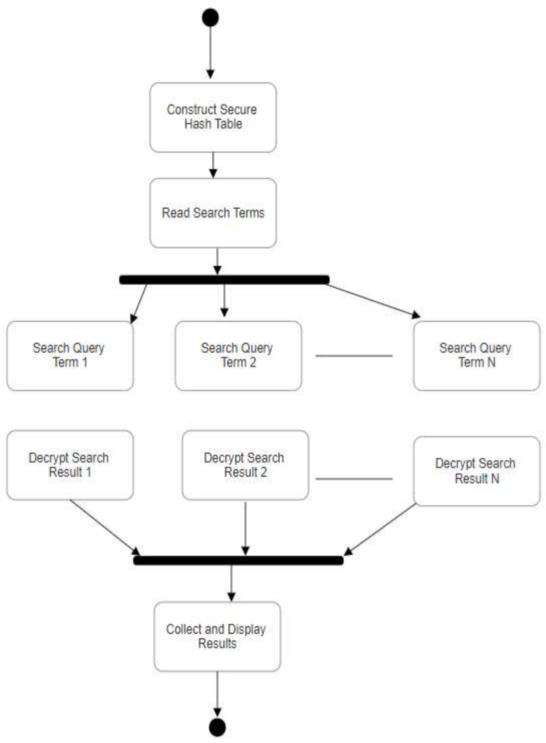


Figure 1: Activity Diagram for Multi-Threaded Secure Search

The technique of searching is done in three steps:
1. Construction of the secure hash index: in this step, the hash table is built, where each key is a hash value that represents encrypted data or some identifier, and each value is a pointer that points to the location of the corresponding data that is in encrypted form in the encrypted data file. Both the key and the value records in the hash table are encrypted.
2. Search Query: When we want to search a term, it is hashed to get a value in the secure hash table, the returned value will give us a pointer to the encrypted data in the data file.
3. Encrypted data Retrieval and Decryption: using the pointer that we got in step 2, we can access the data file to retrieve the required data in encrypted form. The retrieved data is then decrypted to get its original form which we can use as needed.

Using parallel processing, multi-threaded search can be done, we can run multiple threads that preform steps 2 and 3 for different terms simultaneously. Activity diagram for the presented method is shown in Figure 1.

3.2. Secure Hash Table Class Implementation

The hash table contains the search keys and the values. In this implementation, (AES) is used to encrypt the keys and the values. The hash table and the encrypted data are stored in different files. The keys and values in the secure hash table are based on the encrypted data in the data file, the values stored in the hash index are the locations of the encrypted data in the data file.

The main components of the Secure Hash Table class are shown in Figure 2.

3.3. Multi-Threaded Search Class Implementation

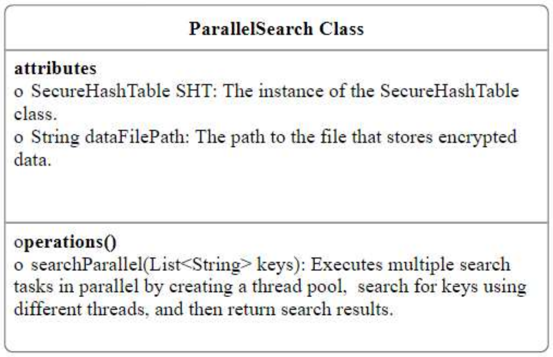
In this implementation, we aim to improve the efficiency of the search by applying it using multiple threads, in this way separate multiple search operations can by multiple threads simultaneously which shall decrease the overall search time.

The main components of the Parallel Search class are presented in Figure 3.

3.4. Implementation Steps

The following steps explain the sequence in which the secure hash search is done based on the SecureHashTable and ParallelSearch classes.

A. Initialize Components:Create instances of **SecureHashTable** and **ParallelSearch**.
B. Store Data:
   - Use putV() to add data to the hash table.
C. Save and Load Data:
   - Use saveToFile() to save encrypted data.
   - Use readFromFile() to load encrypted data.
D. Perform Searches:
   - Use SearchMultiple(keys) to search for multiple patterns in parallel.

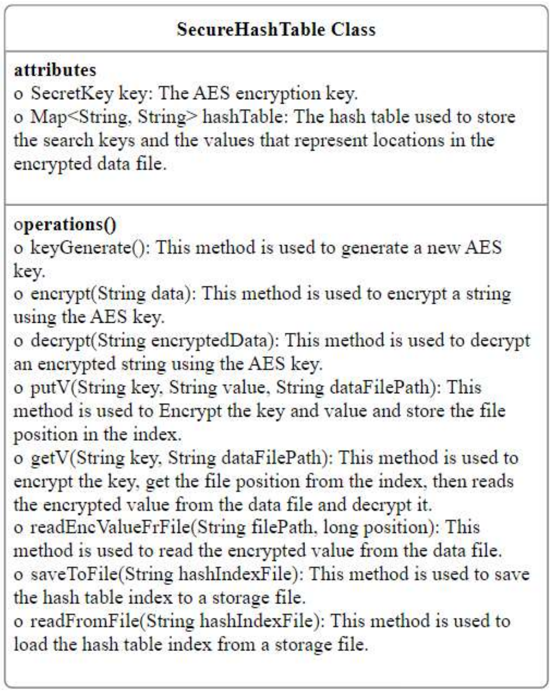


ParallelSearch Class

attributes
o SecureHashTable SHT: The instance of the SecureHashTable class.
o String dataFilePath: The path to the file that stores encrypted data.

operations()
o searchParallel(List<String> keys): Executes multiple search tasks in parallel by creating a thread pool, search for keys using different threads, and then return search results.

Figure 2: Secure Hash Table Class



SecureHashTable Class

attributes
o SecretKey key: The AES encryption key.
o Map<String, String> hashTable: The hash table used to store the search keys and the values that represent locations in the encrypted data file.

operations()
o keyGenerate(): This method is used to generate a new AES key.
o encrypt(String data): This method is used to encrypt a string using the AES key.
o decrypt(String encryptedData): This method is used to decrypt an encrypted string using the AES key.
o putV(String key, String value, String dataFilePath): This method is used to Encrypt the key and value and store the file position in the index.
o getV(String key, String dataFilePath): This method is used to encrypt the key, get the file position from the index, then reads the encrypted value from the data file and decrypt it.
o readEncValueFrFile(String filePath, long position): This method is used to read the encrypted value from the data file.
o saveToFile(String hashIndexFile): This method is used to save the hash table index to a storage file.
o readFromFile(String hashIndexFile): This method is used to load the hash table index from a storage file.

Figure 3: Parallel Search Class

4. EXPERIMENTAL RESULTS AND ANALYSIS

4.1. Implementation Environment

The hardware environment for our implementation is a computer with 16GB of RAM and an Intel core i7 processor. The software we used is Java and for encryption we used BouncyCastle library. We generate random data set with 100,000 elements in our experiments.

4.2. Performance metrics

The following metrics were recorded in our experiments:

- Number of Threads: The number of threads used in the search operaions.
- Number of elements: The size of the dataset.
- Search Time (ns): The average time needed by search operations.
- Encryption Time (ns): The time needed to encrypt the dataset.
- Decryption Time (ns): The time needed to decrypt the dataset.
- Memory Usage (MB): The average memory consumption by threads in megabytes.
- Search Time Speedup: Search time needed by one thread / Search time needed by multiple threads.

4.3. Experimental results of multi-threaded search on encrypted hash table

When we run the tests, we started the search with one thread and then increased the number of threads one thread at a time. We did the same with the dataset, where we started with a dataset with 10,000 entries in the first test, increasing it till we reached 100,000 entries and recording the average search time, encryption time, decryption time and memory usage in each test. The results of our tests are presented in Table 1.

Based on Table 1, we can compute the speedup that we gain by distributing the search load on multiple threads, Table 2 contains the search time needed by multiple threads when searching in encrypted dataset of 10,000 entries together with the calculated speedup.

The effect of increasing the number of threads to handle the search operation on a fixed size data set of 10,000 entries is shown in Figure 4.

The Search time is also affected by the dataset size, the relationship between the parallel search time and varying dataset size is depicted in Figure 5.

Table 1: Results of Parallel Search on Encrypted Data Using Secure Hash Table

Number of Threads	Number of Entries	Average Search Time (Seconds)	Encryption Time (Seconds)	Decryption Time (Seconds)	Memory Usage (MB)
1	10,000	1.50	0.113	0.113	50
1	50,000	7.70	0.59	0.59	50
1	100,000	15.6	1.34	1.34	50
2	10,000	0.79	0.062	0.062	55
2	50,000	3.75	0.314	0.314	55
2	100,000	7.80	0.641	0.641	55
3	10,000	0.63	0.051	0.051	57
3	50,000	3.10	0.257	0.257	57
3	100,000	6.80	0.592	0.592	57
4	10,000	0.47	0.042	0.042	60
4	50,000	2.48	0.224	0.224	60
4	100,000	4.92	0.466	0.466	60
5	10,000	0.41	0.041	0.041	62
5	50,000	2.12	0.21	0.21	62
5	100,000	4.35	0.453	0.453	62
6	10,000	0.37	0.038	0.038	65
6	50,000	1.82	0.159	0.159	65
6	100,000	3.83	0.389	0.389	65
7	10,000	0.31	0.031	0.031	68
7	50,000	1.63	0.156	0.156	68
7	100,000	3.18	0.354	0.354	68
8	10,000	0.26	0.026	0.026	70
8	50,000	1.37	0.144	0.144	70
8	100,000	2.93	0.294	0.294	70

Table 2: Speedup of Multi-Threaded Search

Number of threads	Average Search Time (s)	Speedup
1	1.5	1
2	0.79	1.898734
3	0.63	2.380952
4	0.47	3.191489
5	0.41	3.658537
6	0.37	4.054054
7	0.31	4.83871
8	0.26	5.769231

The usage of memory during search operation is affected by the number of threads, in Figure 6. The relationship between number of threads and used memory is presented.

We noticed that the encryption time and decryption time are nearly the same, both times are affected by the number of threads and the dataset size, the relationship between the encryption time with number of threads and data set size is shown in Figure 7.

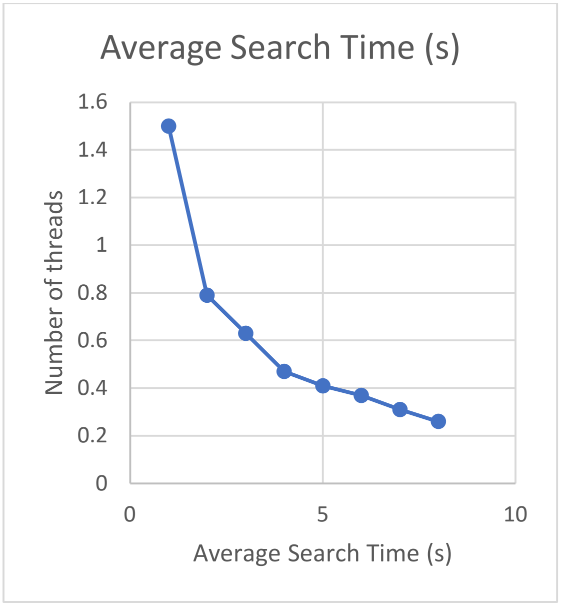


Figure 4: Average Parallel Search Time for 10,000 Entries

The speedup gained by using multi-threaded search is depicted in Figure 8.

4.4. Analysis of Experimental Results

Using a secure hash table for searching encrypted data was successful and gave correct search results in all tests, so it is a sound and accurate way of searching. Searching using this algorithm will maintain the security and privacy of the encrypted data.

Using only one thread for searching will always have the highest search time in all experiments since it represents the sequential version of the algorithm, this can be used in certain cases, such as if we have only one pattern to search for or if we have a single core processor, which is not applicable in most cases.

When reading and analyzing the experimental results we notice that using multiple threads for searching multiple patterns in parallel is a gain, it has a good effect on reducing the overall search time and encryption/decryption time and thus speeding up search operations. In Figure 4. We see that the search time decreases when increasing the number of search thread, the time reduction is not constant because we will have communication and overhead time due to thread management process as the number of threads increase, but in general, using multithreaded search will speed up the search operation.

In Figure 5. The relationship between multiple threads and search time is also shown, but in this time, taking varying dataset size into consideration, it is also clear and expected that the search time will increase as the dataset size increases, and again, using multiple threads for searching will make searching faster in all given dataset sizes, so we conclude that using multi-threaded search will improve and enhance the efficiency of searching.

Figure 7. Presents the relationship between encryption time and varying number of threads with varying dataset size, encryption and decryption times are crucial in our application since all operations deal with encrypted hash table index and encrypted data. It is clear in Figure 4. that using multi-threaded search has a great impact on reducing encryption time and so speeding up searching of encrypted data.

Using multiple threads for searching will clearly reduce search time, but on the other hand, as the number of threads increase, memory requirements will increase, so we must consider that when using this approach. Figure 6. Shows the results of scaling up number of threads on memory requirements.

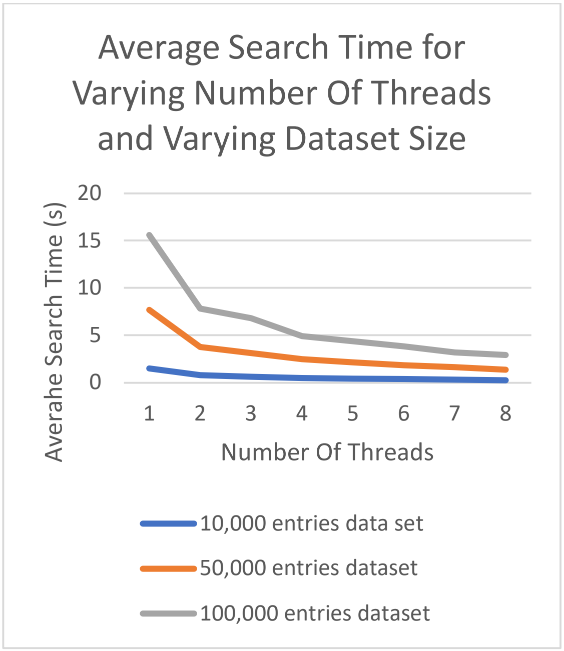


Figure 5: Average Parallel Search Time for Varying Number of Threads and Varying Dataset Size

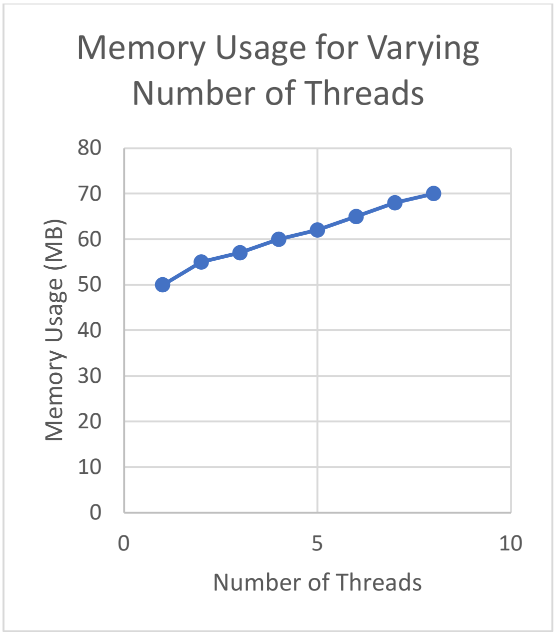


Figure 6: Memory Usage for Varying Number of Threads

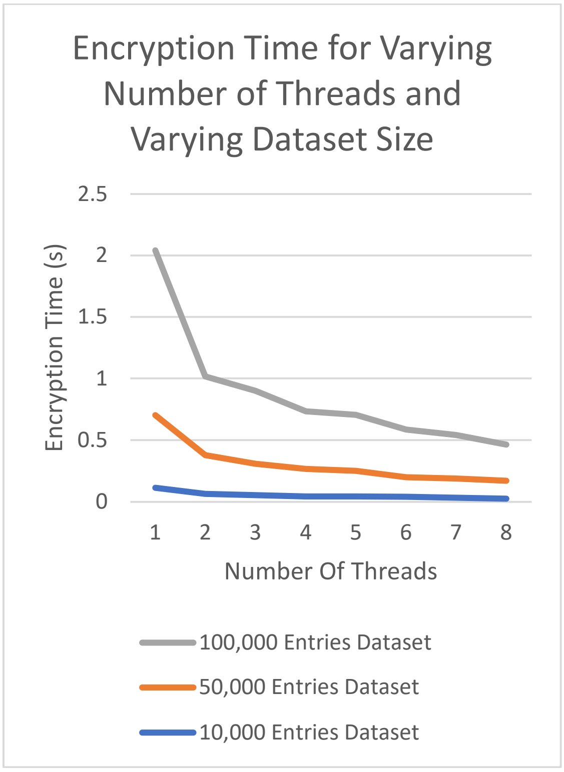


Figure 7: Encryption Time for Varying Number of Threads and Varying Dataset Size

Referring to Figure 8. We can see that using multi-threaded search has a great impact on the performance and efficiency of the search, a direct relationship is present between the number of threads for searching with the speedup gained, the more threads that we use for searching, the less search time will be consumed and a more speedup will be gained. Taking into consideration that we are limited by the number of cores in the processor, in case that the number of threads starts to be higher than the number of cores, search time will start to increase and speedup will start to decrease.

Similar analysis of speedup can be done regarding to encryption and decryption time. As discussed earlier in this section, it is clear in Figure 7. that the encryption time decreases as the number of used threads increase, which clearly implies that speedup will also be gained here by using multiple threads in this process.

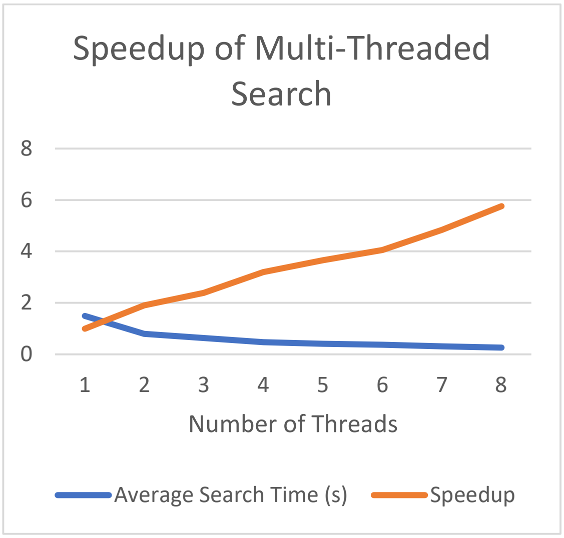


Figure 8: Speedup of Multi-Threaded Search

When comparing these results to other secure index structures such as the B+ tree index presented in [11], we find a number of distinctions. In our work, the focus on hash table index combined with parallel environment led to a rapid search times making an advantage in scenarios where we need to search for multiple patterns. On the other hand, the B+ Tree index is designed for efficient range queries and dynamic updates, so it is suitable in applications with larger datasets and more diverse queries. Both works aim to balance efficiency with security in encrypted data search.

As a results we can say that using of secure hash table index is a success in searching encrypted data, and using of multi-threaded search will enhance the performance of the search, yet adding more memory requirements.

5. CONCLUSION

In this work, a method for fast and secure string matching on encrypted data based on encrypted hash table index and multi-threading for search operations is developed. The presented approach maintains the confidentiality of data by encrypting both the keys and the values in the hash table while providing an efficient search performance by using of parallel search. Data protection is strengthened by integration of symmetric encryption, while fast access of encrypted data is achieved by using a hash table. The using of parallel search will reduce search times, ensuring the effectiveness of parallel processing when dealing with large datasets. The presented experimental results showed that our method has a good scaling with increased number of threads and datasets sizes, though the method has a trade-off with overhead times and memory usage.

The proposed approach provides security and efficiency, making it applicable for systems that needs both of these factors, such as private search engines and cloud computing. Further work may focus on real-world environment testing of this approach and optimizing the index data structure. Overall, this study makes a contribution on secure data searching techniques and gives a framework for searching encrypted data effectively.

REFERENCES

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in Proceedings of the 41st Annual ACM Symposium on Theory of Computing, pp. 169-178, 2009.

[2] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," Designs, Codes and Cryptography, vol. 71, no. 1, pp. 57-81, 2014.

[3] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in Advances in Cryptology – EUROCRYPT, pp. 1-29, 2011.

[4] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," Journal of Computer Security, vol. 19, no. 5, pp. 895-934, 2011.

[5] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in NDSS, pp. 1-32, 2014.

[6] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," Journal of the ACM (JACM), vol. 43, no. 3, pp. 431-473, 1996.

[7] E. Stefanov, E. Shi, and D. Song, "Path ORAM: An extremely simple oblivious RAM protocol," in Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 299-310, 2013.

[8] Z. Ren, J. Yu, Y. Zhang, and C. Wang, "Designated-cloud path ORAM: Efficient outsourcing of RAM with scalability bottleneck," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 957-973, 2017.

[9] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in Proceedings of the 2000 IEEE Symposium on Security and Privacy, pp. 44-55, 2000.

[10] Y. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in International Conference on Applied Cryptography and Network Security, pp. 442-456, 2005.

[11] Z.-F. Wang, A.-G. Tang, and W. Wang, "Fast Query Over Encrypted Data Based on B+ Tree," 2009 International Conference on Apperceiving Computing and Intelligence Analysis, pp. 1-6, 2009.

[12] H. Hu, J. Xu, C. Ren, and B. Choi, "Processing private queries over untrusted data cloud through privacy homomorphism," in Proceedings of the 27th International Conference on Data Engineering, pp. 663-674, 2011.

[13] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 965-976, 2012.