EPiC Series in Computing

Volume 98, 2024, Pages 200–206

Proceedings of 39th International Conference on Computers and Their Applications

Data Security in the Cloud Using pTree-based Homomorphic Intrinsic Data Encryption System (pHIDES)

Mohammad Hossain[1]* and Vinayak Sharma[2]†

[1]University of Minnesota Crookston, MN, USA

[2]University of Windsor, ON, Canada

hossain@crk.umn.edu, sharm981@uwindsor.ca

Abstract

Cloud usage for storing data and performing operations has gained immense popularity in recent times. However, there are concerns that uploading data to the cloud increases the chances of unauthorized parties accessing it. One way to secure data from unauthorized access is to encrypt it. Even if the data is hacked, the hackers will not be able to retrieve any information from the data without knowing the 'Key' to decrypt it. But when data needs to be used for services such as data analytics, it must be in its original, non-encrypted form. Decrypting the data makes it vulnerable again, which is why Homomorphic Encryption could be the solution to this problem. In this encryption method, the analytical engine can use the encrypted data to perform analysis, where the analysis result will also be in decrypted form. Only authorized users can access the results using the 'Key.' This research proposal proposes a method called pHIDES to enhance data security in the cloud. The pHIDES (pTree-based Homomorphic Intrinsic Data Encryption System) represents data in pTree (Predicate tree) format, a data mining-ready data structure proven to manipulate a large volume of data effectively. The concept of Homomorphic Encryption (HME) along with pHIDES is discussed in our research, along with the algorithmic execution to analyze the effectiveness of the algorithm used to encrypt data in the cloud.

1 Introduction

Uploading data in the cloud increases the chances of the data being hacked and misused by unauthorized users. It is crucial to have safeguards to protect the data to ensure secure and widespread usage of cloud services. This research paper proposes a method called pHIDES, which uses homomorphic encryption to encrypt data, enhancing data security in the cloud. The pHIDES stands for

* Corresponding Author

† Most work was done when the first author was at the University of Minnesota Crookston.

pTree-based **H**omomorphic **I**ntrinsic **D**ata **E**ncryption **S**ystem. It represents data in pTree (Predicate tree) format, a data mining-ready data structure proven to manipulate a large volume of data effectively. The concept of Homomorphic encryption (HME) along with pHIDES is discussed in our research.

The paper is structured in a clear and organized manner, consisting of the following sections: abstract, introduction, literature review, introduction to pHIDES, a working example, and conclusion. The abstract provides a brief summary of the paper's main points and findings. The introduction provides context and background information on the topic. The literature review discusses relevant previous studies, theories, and findings related to the topic. The introduction to pHIDES explains the concept and key features of this topic. The working example provides a practical application of pHIDES. The conclusion summarizes the paper's findings and conclusions and may include recommendations for future research or practical applications.

2 Literature Review

Homomorphic encryption allows computations to be performed on encrypted data without decrypting it. This capability offers significant potential for enhancing privacy and security in various applications, particularly in cloud computing and data outsourcing scenarios.

Gentry's seminal work in 2009 introduced the concept of fully homomorphic encryption (FHE), enabling arbitrary computations on encrypted data [1]. However, early FHE schemes were impractical due to high computational overhead. Over the years, researchers have made remarkable progress in improving the efficiency and practicality of homomorphic encryption schemes.

Several notable schemes have emerged, such as the BGV scheme proposed by Brakerski, Gentry, and Vaikuntanathan in 2011, which significantly reduced the computational complexity of FHE [2]. Furthermore, recent advancements like the TFHE (Fast Fully Homomorphic Encryption over the Torus) scheme have demonstrated promising performance improvements, making homomorphic encryption more viable for real-world applications [3].

In 2017, Rangasami and Vagdevi explained the aspect of security to the data involves various segments such as network security, strategies of control and access to the service, and storage of data [4]. The paper also mentions the lack of enthusiasm among users for the data security technology offered by various service providers. The paper describes Homomorphic encryption as a way of providing data security to the data on which operations could be performed on the encrypted data itself. The paper also further presents an analysis report of homomorphic encryption methods in terms of security provided to data and their usefulness in cloud computing.

In 2017, Mohan, Devi, and Prakash [5] conducted a detailed survey of the various homomorphic encryption schemes based on the parameters involved, the encryption-decryption mechanism, their homomorphic properties, security considerations, etc. The paper also showcases that encryption methods with homomorphic properties are suitable for applications requiring data privacy and security.

Patil and Biradar [6] presented parallel processing of fully homomorphic encryption on encrypted cloud data in their paper in 2018. The paper showcases that parallel processing produces better performance than performing operations on a sequential process. It also shows work on the Data partitioning method, which is used to improve the security of data on the cloud.

Mahmood and Ibrahem [7] showcased an overview of security issues in cloud computing and utilization of the fully homomorphic encryption technique has drawbacks of large key size and low calculations efficiency, and it is not practical for the secure cloud computing.

Oladunni and Sharad Sharma wrote a survey paper [8] in 2019 regarding different types of homomorphic encryptions, explaining the different techniques of this technology.

3 Introduction to pHIDES

pTree based Homomorphic Intrinsic Data Encryption System (pHIDES) is an encryption mechanism where arithmetic operation on the encrypted data produces the same result as the arithmetic operation on original data. pHIDES uses pTree technology to encode the data ready for encryption. The following subsections describe the pTree technology, the pHIDES mechanism, and the algorithms of pHIDES. The next section describes a working example of pHIDES.

3.1 Review of pTree Technology

pTree stands for Predicate tree. pTree stores data vertically, capturing the bit slices of the data [9]. It is lossless and is also data mining ready. In order to convert a set of data using pTree technology, we first convert the values of the data set into binary. Each vertical bit slice of the data set produces one level-0 pTree. The set of all level-0 pTrees represents the data set as a pTree set.

The following example explains it. Assume that we are constructing our data set using 8 values, as shown in Table 1. Each value in the data set is in the range of 0 to 15, so we represent it using 4-bit binary numbers. After converting the values into binary, we get four bit slices of P_1 through P_4. Each of the bit slices represents one level-0 pTree.

Table 1: Example of level-0 pTree and pTree set.

Index	*Data*	P_1	P_2	P_3	P_4
0	12	1	1	0	0
1	03	0	0	1	1
2	10	1	0	1	0
3	09	1	0	0	1
4	02	0	0	1	0
5	14	1	1	1	0
6	15	1	1	1	1
7	11	1	0	1	1

The length of these pTree, as well as the length of the pTree set is the number of values in the data set which is 8 in this example. To access individual values in the data set, we can use an index system that starts from 0 and goes up to one less than the length of the data set, which will be 7 in this example. So, *Data[0]* = 12, *Data[3]* = 09, *Data[7]* = 11, and so forth. Similarly, using the same indexing system, we access the individual bits of the pTrees. For instance, $P_1[3] = 1$, $P_3[6] = 1$, $P_4[5] = 0$, and so forth. If *pT* represents all four level-0 pTrees as a pTree set, *pT[0]* = 1100, *pT[3]* = 1001, *pT[7]* = 1011, etc. If we want to store this data set in a storage device as a pTree set, we will start storing from the rightmost pTree (P_4) and work our way to the leftmost pTree (P_1) by capturing 8 bits at a time as one byte. Thus, the hexadecimal representation will look like this: **53 6F 86 B7.** This representation will take only 4 bytes of space in the storage device, whereas the original data would take 8 bytes of space in the storage device. Moreover, because of the intrinsic encryption capability of pTree technology, the original data is hard to retrieve from this pTree set data.

3.2 Working Mechanism of pHIDES

As shown in the previous section's example, pHIDES encrypts the data intrinsically and compresses it in most cases. All the arithmetic operations can be performed directly without decrypting the data, and it can be easily configured as FHE.

The pHIDES uses a private key *S,* to create an encrypted pTree set from the input pTree set generated from the given dataset. The length of the encrypted pTree set will be double that of the input pTree set. That is, if the input pTree set has a length of *L*, the encrypted pTree will be the length of *2L*. Using the private key *S*, pHIDES generates a list of *L* indices between 0 to *2L* – 1. Then these indices of the encrypted pTree set are filled with random numbers, while the rest of the indices are filled with the values taken from the input pTree set.

Table 2: Encryption process using pHIDES

Length of the input pTree set: 8 *Length of the output pTree set: 16* *Indices generated: [2, 4, 5, 7, 9, 10, 13,14]*					
Index	*Data*	P_1	P_2	P_3	P_4
0	12	1	1	0	0
1	03	0	0	1	1
2	09	1	0	0	1
3	10	1	0	1	0
4	08	1	0	0	0
5	15	1	1	1	1
6	09	1	0	0	1
7	07	0	1	1	1
8	02	0	0	1	0
9	14	1	1	1	0
10	01	0	0	0	1
11	14	1	1	1	0
12	15	1	1	1	1
13	00	0	0	0	0
14	15	1	1	1	1
15	11	1	0	1	1

When this encrypted pTree set is saved in the storage device, it will be as follows in hexadecimal format: **67 55 85 BE 2B DB 5A 5B.** To retrieve the original pTree set from this encrypted pTree, we need to apply a decrypting process, which starts with producing the same list of *L* indices. For that, it needs to use the same key and the same function used in the encryption process. Next, pHIDES will simply ignore the values of the encrypted pTree set that are stored in those indices and copy the remaining values in the decrypted pTree set. The following subsection shows the details of these algorithms.

If an attempt is made to retrieve the data using the wrong key, it will generate a wrong list of *L* indices. As a result, the system will ignore some of the valid values in the data set while including some of the invalid values in the output pTree set. Therefore, the original data set will remain hidden. The next subsection describes the encryption and decryption methods concisely.

3.3 Algorithm for pHIDES:

Algorithm 1: Encrypt Using pHIDES

Input:
In_pT, a pTree set of *n* pTrees representing the input data set.
Key, a string holding the encryption key.
Output:
Out_pT, a pTree set of *n* pTrees representing the encrypted data set.

Encrypt(*In_pT, Key, Out_pT*)
10. Generate a seed *S* from *Key* using a Hash function
11. Create *Out_pT* of length of *2*L* where *L* = length of *In_pT*
12. Using the seed *S,* Generate a list of indices (*SI*) of *L* distinct indices using a function *indexFunction,* such that *SI=indexFunction(S)*
13. *j*← 0
14. **foreach** *i* **in** *(0:2*L – 1)* **do**
15. **if** *i* **in** *SI* **then**
16. *Out_pT[i]* = Random()
17. **else**
18. *Out_pT[i] = In_pT[j]*
19. *j*← *j+1*
20. **Endfor**

Algorithm 2: Decrypt Using pHIDES

Input:
In_pT, a pTree set of *n* pTrees representing the encrypted data set.
Key, a string holding the encryption key.
Output:
Out_pT, a pTree set of *n* pTrees representing the decrypted data set.

Decrypt(*In_pT, Key, Out_pT*)
1. Generate a seed *S* from *Key* using a Hash function
2. Create *Out_pT* of length of *1/2*L* where *L* = length of *In_pT*
3. Using the seed *S,* Generate a list of indices (*SI*) of *L* distinct indices using a function *indexFunction,* such that *SI=indexFunction(S)*
4. *j*← 0
5. **foreach** *i* **in** *(0: L – 1)* **do**
6. **if** *i* **not in** *SI* **then**
7. *Out_pT[j] = In_pT[i]*
8. *j*← *j+1*
9. **Endfor**

4 Example of an Application Using pHIDES

Assuming we have a data set, as shown in Table 1, that was uploaded to the cloud after encrypting it using pHIDES. That is, we have the encrypted pTree set, as displayed in Table 2, in the cloud. Now, let's say our application needs to know which values in the data set are greater than or equal to 8. The expected result for the input data set should appear as shown under the column "Expected output" in Table 3(A). However, since the application works on the encrypted pTree set instead of the original dataset, it will produce the result as shown under the column "Produced output" in Table 3(B). At this point, the user will apply the pHIDES with the same key to decrypt the “Produced output” column of Table 3(B) and get the expected result as shown in Table 3(C).

Table 3: The Example Application of pHIDES

A. Input		
Index	*Data*	*Expected output*
0	12	1
1	03	0
2	10	1
3	09	1
4	02	0
5	14	1
6	15	1
7	11	1

B. Encrypted Data		
Index	*Data*	*Produced output*
0	12	1
1	03	0
2	09	1
3	10	1
4	08	1
5	15	1
6	09	1
7	07	0
8	02	0
9	14	1
10	01	0
11	14	1
12	15	1
13	00	0
14	15	1
15	11	1

C. After Decryption
1
0
1
1
0
1
1
1

The decryption process of pHIDES generates the same indices L as long as the user uses the same key that was used in the encryption process. In this example, the $L = \{2, 4, 5, 7, 9, 10, 13, 14\}$. Next, pHIDES will produce the decrypted output pTree set, considering only those values that are not from the indices L and ignoring other values. Any attempt to retrieve the result with the wrong key will generate a list of indices different from L. Thus, the system will not be able to distinguish between valid and invalid indices within the encrypted output column and will produce an incorrect result.

5 Conclusion

As a summary, we have demonstrated the success of employing pHIDES for data encryption in cloud applications along with data analytics algorithms. Our research indicates that pHIDES can be utilized as an alternative to current Homomorphic Encryption algorithms, offering a more appropriate setting for enhanced data security along with operational advantages over conventional techniques. However, we would like to mention that pHIDES will work best if the data is already in pTree format.

There is a lot of literature that describes the effective use of pTree in big data analytics. In those applications, pHIDES will provide another level of security used in the cloud [9, 10].

.

References

[1] Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In Proceedings of the forty-sixth annual ACM symposium on theory of computing (pp. 169–178). https://dl.acm.org/doi/10.1145/1536414.1536440

[2] Brakerski, Zvika, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping." ACM Transactions on Computation Theory (TOCT) 6.3 (2014): 1-36.

[3] Chillotti, Ilaria, et al. "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds." Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22. Springer Berlin Heidelberg, 2016.

[4] K. Rangasami and S. Vagdevi, "Comparative study of homomorphic encryption methods for secured data operations in cloud computing," 2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT), 2017, pp. 1-6, doi: 10.1109/ICEECCOT.2017.8284566.

[5] M. Mohan, M. K. K. Devi and V. J. Prakash, "Homomorphic encryption-state of the art," 2017 International Conference on Intelligent Computing and Control (I2C2), 2017, pp. 1-6, doi: 10.1109/I2C2.2017.8321774.

[6] R. S. Patil and P. Biradar, "Secure Parallel Processing on Encrypted Cloud Data Using Fully Homomorphic Encryption," 2018 4th International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), 2018, pp. 242-247, doi: 10.1109/iCATccT44854.2018.9001284.

[7] Z. H. Mahmood and M. K. Ibrahem, "New Fully Homomorphic Encryption Scheme Based on Multistage Partial Homomorphic Encryption Applied in Cloud Computing," 2018 1st Annual International Conference on Information and Sciences (AiCIS), 2018, pp. 182-186, doi: 10.1109/AiCIS.2018.00043.

[8] Timothy Oladunni and Sharad Sharma, “Homomorphic Encryption and Data Security in the Cloud.” Proceedings of 28th International Conference on Software Engineering and Data Engineering (SEDE - 2019), San Diego, CA, USA, September 2019.

[9] Mohammad Hossain, Maninder Singh, and Sameer Abufardeh. “Vertical Data Processing for Mining Big Data: A Predicate Tree Approach,” 28th International Conference on Software Engineering and Data Engineering (SEDE - 2019), San Diego, CA, USA, September 2019, pp. 68-77.

[10] Mohammad Hossain, and Sameer Abufardeh. “A New Method of Calculating Squared Euclidean Distance (SED) Using pTree Technology and Its Performance Analysis.” Proceedings of 34th International Conference of Computers and Their Applications (CATA-2019)