I. J. Mathematical Sciences and Computing, **2024, 4, 34-50**
Published Online on December 8, 2024 by MECS Press (http://www.mecs-press.org/)
DOI: 10.5815/ijmsc.2024.04.04

Hive-Based Data Encryption for Securing Sensitive Data in HDFS

Shivani Awasthi*
Department of Computer Science, HBTU, Kanpur, India
E-mail: 200304002@hbtu.ac.in
ORCID ID: https://orcid.org/0000-0002-2920-5304
*Corresponding Author

Narendra Kohli
Department of Computer Science, HBTU, Kanpur, India
Email: nkohli@hbtu.ac.in
ORCID ID: https://orcid.org/0000-0002-9407-3598

Received: 22 July, 2024; Revised: 24 August, 2024; Accepted: 08 October, 2024; Published: 08 December, 2024

Abstract: Big Data is a new class of technology that gives businesses more insight into their massive data sets, allowing them to make better business decisions and satisfy customers. Big data systems are also a desirable target for hackers due to the aggregation of their data. Hadoop is used to handle large data sets through reading and writing application programs on a distributed system. Hadoop Distributed File System is used to store massive data. Since HDFS does not safeguard data privacy, encrypting the file is the right way to protect the stored data in HDFS but takes a long time. In this paper, regarding privacy concerns, we use different compression-type data storage file formats with the proposed user-defined function (XOR-Onetime pad with AES) to secure data in HDFS. In this way, we provide a dual level of security by masking the selective data and whole data in the file. Our experiment demonstrates that the whole process time is significantly smaller than that of a conventional method. The proposed UDF with ORC, Zlib file format gives 9-10% better performance results than 2DES and other methods. Finally, we decreased the load time of secure data and significantly improved query processing time with the Hive engine.

Index Terms: AES, Avro, Deflate, ORC, Parquet, Snappy, Gzip, HDFS, cloud environment

1. Introduction

Today, data is one of the most valuable resources for business in all industries. The increasing volume and significance of data have given rise to a new issue that conventional analytic methods are unable to address. The solution to this issue arose from developing a novel paradigm known as big data [1]. The need for large organizations, like Google, Facebook, and Yahoo, to analyze enormous amounts of data led to the term "big data" [2,3]. Numerous methods have characterized big data, including 4V Volume, Velocity, Variety, and Veracity and 3V Volume, Variety, and Velocity [4,5,6,7].

Doug Laney, who works at Gartner, characterized big data using the 3 Vs: volume, velocity, and variety. The terms volume, diversity, and velocity refer to the quantity, variety, and pace at which data comes from and goes out of a given source [5]. IBM and Microsoft included veracity, sometimes known as variability, as the fourth V of their concept of big data. The unreliability and complexity of data are referred to as veracity. As the fourth V in the definition of big data, McKinsey & Co. added value [8]. Value is the worth of unexpected findings within massive data collections. Vast data, as defined generally, is a collection of huge, complicated data sets that are too vast for current data processing technology to handle effectively [5]. However, in addition to the amount and variety of data it contains, big data has also raised new issues with data privacy and security [1]. As a result, establishing security in big data has become one of the main challenges that might prevent technological growth; big data cannot get the necessary degree of confidence without sufficient security assurances [9,10].

Since massive data sets come with great responsibility, we offer storage-level security in this research. When using the Hadoop HDFS data storage service, clients are not strongly motivated to save data locally, so they transmit it continuously. As is customary, the data is stored on the Hadoop HDFS storage server so users may access individual data at any time and from any location. The Hadoop HDFS storage server provides information security assurance and

hardware allocation. The client may obtain their information as long as they have an internet connection. Hadoop is one recent innovation that offers a way to store vast amounts of information. The structure depends on Java and is implemented in an open-source manner. Hadoop is used in big clusters or cloud environments [11]. Although Hadoop is widely utilized, indicating its versatility, it does not have robust security features to safeguard the data stored in HDFS. Researchers use various methods for storing the data in HDFS. An encryption scheme is one of the methods for securing data in HDFS.

HDFS may be deployed on low-cost hardware. File sizes up to terabytes or even petabytes are supported by HDFS. Programs that use it typically do so as a result. Hadoop uses a master/slave design, in which a master computer is one of the cluster's machines and all other machines are its slaves [12]. HDFS stores the file's metadata information on a single standalone server called the Namenode and data in another distributed node called the Datanode. The HDFS architecture is shown in Fig. 1.

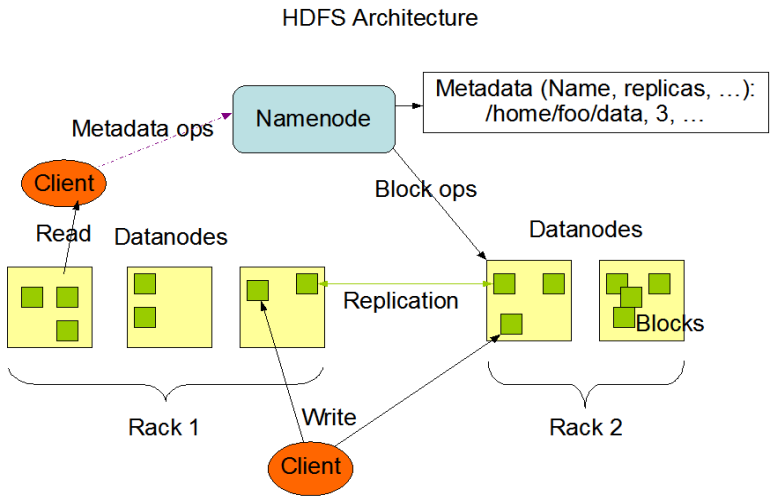

Fig. 1. HDFS architecture [13]

The Namenode and its corresponding job tracker daemon are present on the master node. The Namenode oversees the whole file system's namespace, regulates end-user access to the files, and uses the Heartbeat signal to monitor the Datanode's health. While the actual data is not included, the Namenode is the directory for the Datanode that holds the details about which blocks together make up the file and where those blocks are located. Additionally, keeping a lot of little files is not a good use for HDFS. This is because Namenode memory capacity restricts the total number of files that can be stored in HDFS and stores the file system information. More metadata must be retained when many little files need to be kept, and more metadata takes up memory. The Datanode is made up of a set of all slave nodes that have the Task Tracker daemon attached to them. The real data are spread throughout the cluster and are located at the Datanode. The user data file is distributed by first breaking it into blocks, which are stored in the Datanodes and have a default size of 64 MB. We can manage the size of the data block if it is not fixed. The Namenode determines which file block has to be mapped to which Datanode. The same file has many Namenode (Metadata) components. By default, every block is mapped to three Datanodes to replicate data and offer fault tolerance and dependability. Every block's position in the Datanode is known to the Namenode. In addition, Namenode does several other tasks, including renaming files and directories and opening and shutting files. Additionally, inside a given rack, the Namenode determines which file block has to be written to which Datanode. The rack is a section of storage where several Datanodes are arranged. Based on the proximity of the nodes to each other, the Namenode decides where these blocks belong. The Datanodes communicate more quickly the closer they are to one another.

Namenode is truly single point of failure because Namenode manages all the metadata information such as the location of the data blocks in data nodes. If Namenode fails, the entire HDFS cluster becomes unreachable, as it can not track the data location. To resolve the issue, Secondary Namenode incorporate into HDFS, it is not a redundant backup of the Namenode as is usually misinterpreted. Instead, the secondary namenode create the checkpoint of the namenode metadata at regular basis by integrating its edits log with the file system image. Although this checkpointing helps to prevent the edits log of the primary Namenode from becoming unduly huge, it does not enable failover in the event that the Namenode has a failure. HDFS will stay unavailable until the Namenode is either restored or replaced if it fails to function properly.

In this way, secondary Namenode provides metadata management through checkpoints. It doesn't provide the high availability typically associated with high redundancy.

Apache Hive serves as a database management and query optimization interface for large-scale data processing in HDFS. A key component of many data lake systems, the Hive Meta Store (HMS) provides a central repository for quickly assessed information to enable data-driven, well-informed decision-making [17]. Hive is not a relational database, despite storing the schemas in a database and the processed data in HDFS. Furthermore, Hive features a faster query language similar to SQL called Hive Query Language (HiveQL or HQL) [16]. A database technology called Hive aids in the meaning of databases and schema to analyze structured data. All of the HQL statements are internally translated by a compiler into Map Reduce jobs that are sent to Hadoop for job execution. HDFS is a data warehouse and storage system, while the hive UI is an interface for managing data warehouses. HQL can design and execute Map Reduce queries as statements, and the Hive execution engine is produced by integrating the Map Reduce and HiveQL process engines. [14,15,16].

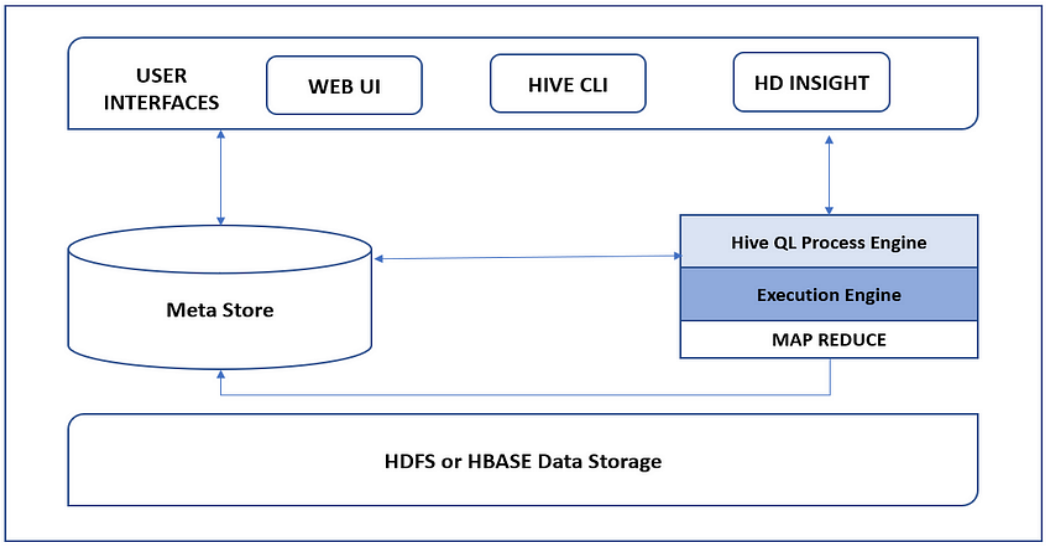

Fig. 2. Hive Architecture [14]

The Hive architecture has been shown in Fig. 2., which works on batch processing. Moreover, the key/pair value is supported by the Hive. It supports all of the database's ACID features. The compiler converts HIVE QL commands into a MapReduce DAG graph [14].

Hive does not directly deal with encryption but plays a main role in accessing secure data. The proposed approach applies to data at rest in HDFS. Hive interacts with this encrypted stored data by HQL without managing the complexity of data decryption or underlying the encryption layers.

Hive serves as a data management and also deals with encrypted data via efficient query processing. Hive query engine-based architecture improves the efficiency of data fetches from HDFS, even data stored in various compressed formats. The various types of storage compressed formats allow the hive to reduce storage space and improve I/O performance with minimum CPU and network overhead. In this way, Hive deals efficiently with encrypted databases even with dual-level overhead by the proposed approach (XOR-Onetimepad with AES) and various file formats in scalable distributed environments.

Privacy, managing, and comprehending vast amounts of data are difficult tasks. Although HDFS can handle a wide variety of file formats, the use case must be taken into account when selecting a format [18]. The most common text formats for storing raw data are CSV, text files, JSON, and XML. The drawback of raw data formats is that they require a significant amount of storage capacity. Another more important problem is storing raw data in HDFS, where each piece of data is duplicated three times. As a result, the quantity of data that HDFS copies would be tripled, making the file size three times larger than it was initially. As a result, compressing and storing the files in proper format into HDFS becomes essential.

HDFS allows for more effective use of storage space based on job definition, and it contains some binary storage data types file formats such as ORC, Avro, and Parquet [19,20]. These formats are designed for systems that use MapReduce types of frameworks. They are a methodical combination of several components, including data storage format, data compression, data encoding, and optimization algorithms for data reading [19]. These file formats provide store data in the form of columns or row-wise in binary form format, so reading and writing from HDFS is easily done from HDFS storage space. Various file formats for the compression of data have been shown in Table 1.

Table 1. Various file formats for the compression of data

File Format	Description	Advantages	Disadvantages
ORC	The file format combines the benefits of both row-wise and column-wise storage for efficient query processing and compression	Faster read and write large datasets. Efficient compression.	Limited support for schema evaluation
Parquet	A columnar storage format that is optimized for processing large data sets in parallel.	Columnar storage allows for efficient querying and processing of large datasets. Efficient compression for reduced storage requirements.	Slower writing speed than a row-based storage format and limited schema evaluation
AVRO	A binary data format that is compact and efficient for storing and exchanging data in big data applications.	Full support for schema evaluation. compact binary format for efficient data exchange.	Slower than another file format due to schema parsing.

On the other hand, the outcomes of various encryption techniques have shown that the original file document size can be increased. Additionally, it is possible to extend both the uploading and downloading times for a particular file. Therefore, to address these issues, we proposed UDF with a different compression type storage file format as a novel encryption-decryption technique. According to simulation data, this method has demonstrated significant gains based on performance parameters such as encryption-decryption time, throughput, and total map-reduce time for encryption-decryption.

Our contributions are:

- Examine Hadoop, weigh its benefits and drawbacks concerning privacy and security, and learn about the various file formats.
- To propose an XOR-Onetime pad with AES-based UDF (User-Defined Function) for encryption and decryption.
- In the Hive shell, the proposed UDF is applied with various compression techniques in different file formats, such as ORC, Parquet, and Avro. The reason behind using the above methodology is that the hive can read and write data from and to the HDFS through HQL and use the different compression-type storage file formats for efficient utilization of space and security of large data over the network.
- Carry out tests using test data to demonstrate the effectiveness of our suggested strategy based on the XOR-Onetime pad with AES and different compression types storage file formats.

2. Related Works

Song et al. [21] propose an HDFS data encryption method compatible with both AES and ARIA algorithms on Hadoop. The Korean government took ARIA as the standard encryption system, even though the available research only supports AES encryption. It includes a block-splitting component and a variable-length data processing component. In this paper, parallel compression and decompression operations on HDFS data SplittableCompressionCodec are used.

Mahmoud et al. [22] suggested a method for enhancing Hadoop's encryption and decryption performance based on inbuilt AES and OTP algorithms. Using this method, the encrypted file's size rose from its original size by 20%.

Kamaruzaman et al. [23] demonstrate that data stored in HDFS can be somewhat secured with the use of the 256-bit AES encryption technique. It provides some degree of security for data saved in HDFS.

Teng et al. [24] suggested modified data encryption algorithms in cloud computing to secure data. Modified AES used random disturbance information, boost key choreography and column mix operation. It ensures attribute privacy and improves decryption efficiency for outsourced data storage.

Jain et al. [25] suggested a blowfish encryption method for access control, authentication, and confidentiality on the Hadoop server instead of the AES algorithm Kamaruzaman et al. [22]. A privacy layer is created between the user and the real owner of data, which addresses vulnerabilities in the Hadoop framework and improves information security.

Khafagy et al. [26] propose a Hybrid-Key Stream Cipher Mechanism (HKSCM) for encryption. HKSCM integrates AES and OTP algorithms to enhance data security. In [25], HKSCM performance is better than AES, RC4, and hybrid algorithms.

Jayapandian et al. [27] discuss the merits and demerits of big data. In this paper, MHT and AES algorithms are used to solve security problems. APEX20KCFPGA is used by AES to improve performance. The drawback of the above scheme is cost and pattern analysis.

Relational database's incapacity to manage unstructured data gave rise to big data. Hadoop and MapReduce are used for processing and handling large amounts of data. HDFS used to store the data. AES encryption is used for the security of large data. It addresses security issues such as the lack of a security model due to the fragmented data in Big Data clusters, and vulnerable environments due to distributed computing and parallel processing [28].

Kaushik et al. [29] evaluate the performance of the popular data encryption methods AES and DESede on the private data kept in HDFS based on data size, encryption time, and decryption time.

Algaradi et al. [30] studied methods and technologies for securing big data in cryptography. Existing encryption algorithms have limitations in performance and execution time. The proposed scheme uses a map-reduce programming model for parallel processing of large data sets and overcomes the limitation.

Mohanraj et al. [31] use a hybrid encryption algorithm using CP-ABE and AES for big data security. This scheme gives 96.5% maximum efficiency, 7.12-minute encryption, and 6.51-minute decryption time.

Khadji et al. [32] investigate how to use lightweight cryptography with the MapReduce architecture to process large amounts of data. It suggests a hybrid key management system to process vast amounts of data securely and effectively. In a lightweight cryptography approach, the author recommended AES and Chacha20 for confidentiality and integrity, although Rabbit stream cipher and NOEKEON block cipher for speed.

Ramya et al. [33] focused on HDFS performance, data node selection, security, and deduplication in a Big data environment. Elliptic curve cryptography with key generation and a PSO-based MapReduce model were employed by the author in this paper. Security, a lack of space, conventional PSO, and FCM issues were the limitations of this paper.

Gupta et al. [34] covered the essential issue of security and privacy in large data processing. To strengthen the security and privacy safeguards that are already in place inside the MapReduce paradigm, the authors suggest a solution that they name the Fortified MapReduce Layer. Their proposed method entails incorporating additional security layers into the MapReduce framework to solve security and privacy challenges. The drawback of this paper was installing and incorporating such a security layer into existing MapReduce systems could be complex and require major modifications.

Muhammad et al. [42] proposed scheme customized genetic algorithm to improve cloud data encryption that offers potential benefits in optimization and performance, it also introduces several drawbacks related to computational complexity, optimization overhead, implementation challenges, security concerns, performance variability, and usability. These factors must be carefully considered when designing and deploying such a system to ensure that the benefits outweigh the costs and risks.

This proposed approach addresses the misunderstanding that HDFS alone can ensure data security, highlighting enhanced security achieved when layered encryption and optimized storage formats are used within the HDFS environment. Our research, taking into account these observations also overcomes the limitation of the paper [34] to introduce the additional map-reduce layer, suggests the dual level of security through hybrid encryption via user-defined function, and uses the different compression types of storage file formats for efficient utilizing space to store large data in a distributed file system context without writing map-reduce jobs. In this way, we reduce load time and improve query processing time with the least computing cost and also secure data within the Hadoop.

3. Proposed Methodology

To protect large amounts of HDFS data and prevent data from specific attacks, we suggested a user-defined function integrating XOR-Onetime pad and AES as hybrid encryption. this approach secures data both selective and file level and also addresses the HDFS's lack of robust, and built-in security. This method overcomes the risk of unauthorized access and interception by encryption of both rest and motion across multiple nodes. In this way, this approach enhances protection against unauthorized access and man-in-middle attacks. This approach further combines the various types of storage formats that improve storage efficiency and also provide secondary layer protection by confusing data patterns and frequency analysis. After that, this approach is applied in Hive to enhance query processing time and minimum load time. This approach minimizes the window time during which data is exposed at transit and rest levels.

When comparing the twofold encryption process to other encryption methods, data security is improved. XOR-Onetime Pad is more secure in comparison to the Onetime Pad. Due to its decentralized access control and secure communication capabilities in dynamic environments, the XOR-Onetime pad with AES has recently drawn increased attention compared to the Onetime pad. The data is cryptographically secured during transmission between the Datanodes as well as storage level by implementing encryption policies with various file formats like ORC, Parquet, and Avro. In this way, we focus on the dual-level encryption of confidential data and overall encrypting the file data via the proposed encryption mechanism with different compression type storage file formats such as ORC, Parquet, and Avro. The overview of hybrid UDF for encryption and decryption to/from HDFS has been shown in Fig. 3.

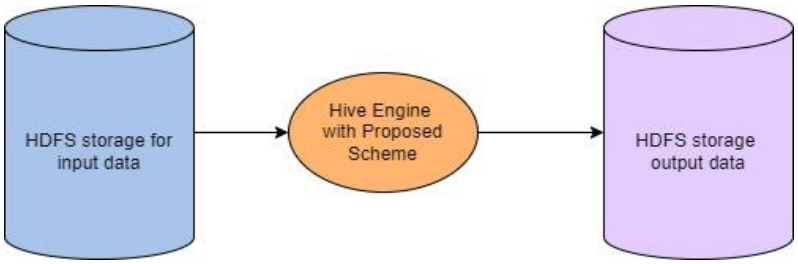

Fig. 3. Overview of the proposed scheme

HDFS stores the file in CSV format then HQL is used for reading and writing the data from HDFS. Through the Hive engine, this HQL easily converts into the map-reduce job and gives the output in the HDFS warehouse based on the above Fig. 3.

3.1. XOR-Onetime pad

A symmetric encryption scheme called the Vernam Cipher was created in 1917 by Gilbert Vernam, an employee of AT & T. One system that demonstrates complete cryptographic stability and perfect secrecy is the Vernam cipher. OTP is a simple Verman cipher cryptosystem. OTP used the "Exclusive OR" (xor) Boolean function [35].

The message is XOR[ed] with the key to obtain the cipher in the one-time pad.

$$c = \text{m XOR K} \tag{1}$$

The cipher is XOR[ed] with the message for decryption.

$$m = \text{x XOR K} \tag{2}$$

When using the XOR-Onetime pad, data is encrypted using a technique that is difficult to decrypt using a brute-force approach, which involves creating random encryption keys and matching them with the right one.

3.2. AES Algorithm

The block length of the AES block cipher is 128 bits. Three distinct key lengths are supported by AES: 128 bits, 192 bits, or 256 bits [36]. 10 processing cycles make up encryption for 128-bit keys, 192-bit key rounds are 12, and 256-bit key rounds are 14-bit keys. One single-byte is included in each processing round-based substitution stage, a permutation step that is row-wise, and a mixing step in columns, as well as the round key. The sequence in which these four actions are carried out varies depending on cryptography and decryption. In contrast to DES, the decryption algorithm and the encryption algorithm are very different. Even though the general procedures for encryption are typically the procedures are performed in the following order: distinct, as was previously indicated. The general architecture of the AES (Advanced Encryption Standard) encryption mechanism has been shown in Fig. 4. [36,37].

In this proposed method, a dual-level encryption combination of XOR-Onetimepad with AES enhances data security in HDFS. Traditional Onetimepad is theoretically unbreakable and achieves perfect secrecy when implements a true random key as long as the message. XOR-Onetimepad does not guarantee perfect secrecy if it uses the truly random and single key. In practical, generating this key with perfect randomness is often infeasible for large datasets, and the reuse or predictability of keys can compromise the security of XOR operation.

The XOR-Onetimepad in a dual-level encryption system serves as an additional layer of security rather than a stand-alone encryption method. When combined with AES, this provides the layered cover mechanism that increases the complexity for attackers but should not be perfectly secure in isolation. The use of AES encryption further enhances data confidentiality compensated by XOR-Onetimepad.

Different compression-type storage file formats are used with the proposed user-defined function based on (XOR-Onetime pad with AES) in the hive shell that provides double-layer security. The first layer uses encryption and decryption functions based on a hybrid approach (XOR-Onetime pad with AES) to protect the sensitive data in the table after that uses different file formats such as ORC, Parquet, and Avro with their compression method to provide dual-level security of important data and overall data that are arranged in the table. Using the above approach, we provide security for important data that is stored in HDFS and easily move data to the cloud environment. The compression method is used by the different storage file formats, by which we compress more data in less space as well as reduce load time over the distributed environment. The different file formats and their types of compression have been shown in Table 2. [38,39,40].

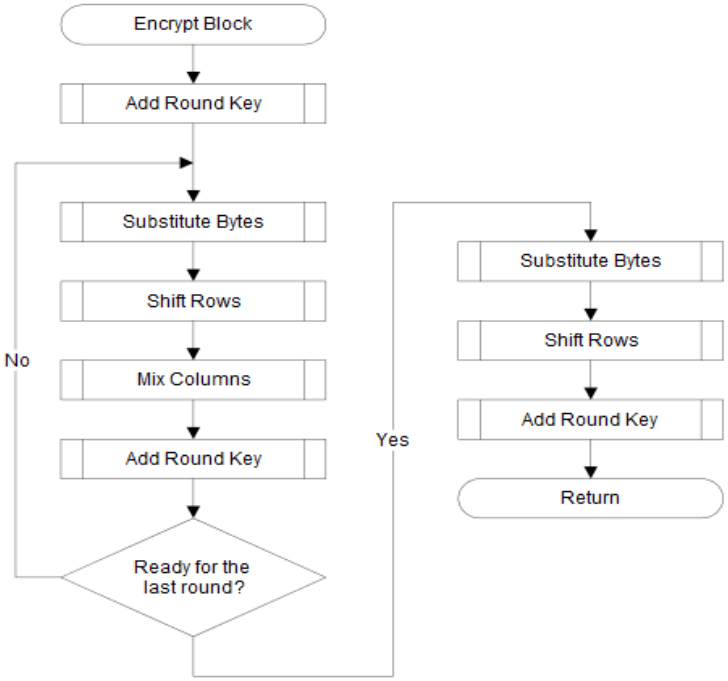

Fig. 4. AES Encryption Process [34]

Table 2. File format and its Compression type

ORC and its Compression type	Parquet and its Compression type	Avro and its Compression type
Snappy	Snappy	Snappy
Zlib	Gzip	Deflate
None	Uncompressed	Uncompressed

A comparative analysis of the impact of ORC, Parquet, and Avro file format in encryption overhead, memory usage, and query performance with optimal use cases is shown in Table 3.

Table 3. Comparative analysis of ORC, Parquet, and Avro file format with encryption overhead, memory usage, and query performance with optimal use cases

File Format	Compression type	Encryption overhead	Memory consumption	Query performance	Optimal use cases
ORC	Zlib, Snappy	Zlib has Low encryption overhead, and Snappy has the best Moderate encryption overhead due to compression,	High memory efficiency due to columnar storage and advanced compression techniques	High query performance on analytical queries.	Best in large-scale data processing distributed environment.
Parquet	Gzip , Snappy	High encryption overhead as compared to Avro due to complex encoding technique	Low memory usage due to its columnar storage and efficient data encoding techniques	High query performance on analytical queries.	High performance in a distributed environment.
Avro	Deflate, Snappy	Low encryption overhead due to row-based storage. Serialization and deserialization fast in Avro.	Moderate memory consumption	Moderate query performance in the analytical platform due to row-based storage.	Suitable for write-intensive workflow.

The above Table 3., highlights that ORC and Parquet are best for query performance due to their columnar data storage, while Avro is best suited for write-intensive flow. ORC offers good efficiency and high query speed for large data environments. Parquet has high encryption overhead and efficient read-intensive data flow.

3.3. Pseudocode of Proposed UDF (XOR-Onetime pad with AES)

1 Generate XOR-Onetime pad pseudocode
function generate_one_time_pad12(len):
pad = random_bytes(len) // Generate random bytes of specified length return pad

2 XOR encryption using Onetime pad Pseudocode
function xor_encrypt1(plaintext1, one_time_pad12):
ciphertext1 = ""
for i from 0 to length(plaintext1) - 1:
ciphertext1 += char_to_byte(plaintext1[i]) XOR one_time_pad12[i]
return ciphertext1
3 XOR decryption using Onetime pad Pseudocode
function xor_decrypt1(ciphertext1, one_time_pad12):
plaintext1 = ""
for i from 0 to length(ciphertext1) - 1:
plaintext1 += byte_to_char(ciphertext1[i] XOR one_time_pad12[i])
return plaintext1
4 AES Encryption Pseudocode
function aes_encrypt1(plaintext1, key):
cipher = AES.new (key, AES.MODE_ECB) // Initialize AES cipher in CBC mode
ciphertext1 = cipher.encrypt(pad(plaintext1)) // Pad plaintext if necessary and then encrypt
return ciphertext1
5 AES Decryption Pseudocode
function aes_decrypt1(ciphertext1, key):
cipher = AES.new(key, AES.MODE_ECB) // Initialize AES cipher in CBC mode
plaintext1 = unpad(cipher. decrypt(ciphertext1)) // Decrypt ciphertext and then unpad plaintext
return plaintext1
6 Combine XOR-Onetime pad with AES Pseudocode
function xor_aes_encrypt1(plaintext1, aes_key_length):
one_time_pad12 = generate_one_time_pad12(aes_key_length)
ciphertext1 = xor_encrypt1(plaintext1, one_time_pad12)
aes_key = generate_one_time_pad12(aes_key_length) // Generate a separate AES key
encrypted_aes_key = aes_encrypt1(aes_key, one_time_pad12) // Encrypt AES key using XOR-one-time pad
return (ciphertext1, encrypted_aes_key)
function xor_aes_decrypt1(ciphertext1, encrypted_aes_key):
(one_time_pad12, _) = xor_aes_decrypt1(encrypted_aes_key, encrypted_aes_key) // Decrypt AES key using XOR-one-time pad
plaintext1 = xor_decrypt1(ciphertext1, one_time_pad12)
return plaintext1

3.4. Proposed UDF based on (XOR-Onetime pad with AES)

The proposed UDF is shown in Fig. 5. This UDF converts into a jar file and loads this jar file to HDFS work as a permanent function.

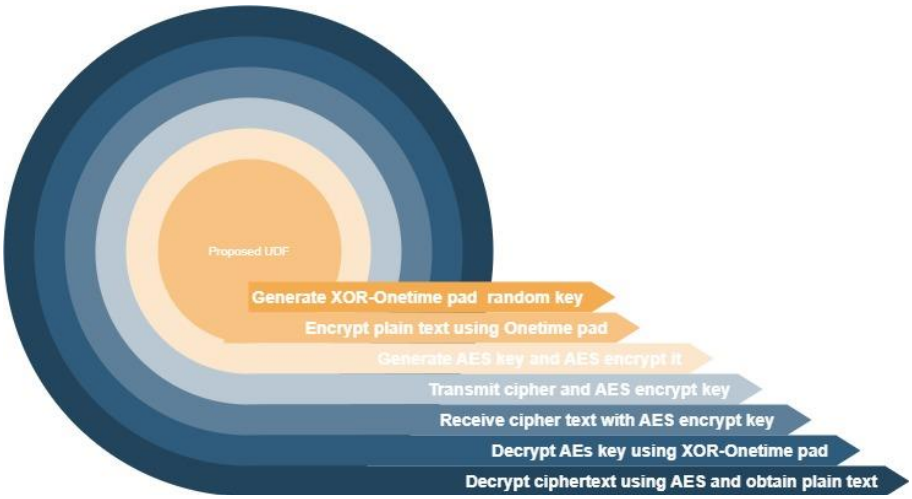

Fig. 5. Flow chart of Proposed UDF

The steps of the Proposed methodology in the Hive shell are shown in Fig. 6.

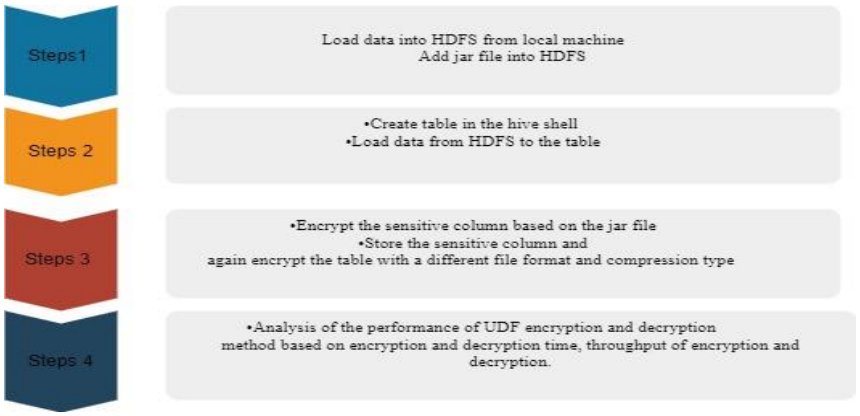

Fig. 6. Steps for integrating the proposed methodology in Hive

A way to increase security that combines proposed data encryption UDF with different compression types of storage data file formats is shown in Fig. 6. After processing, HDFS-encrypted files stay encrypted in the HDFS storage warehouse in the Apache Hadoop environment. The results, including the intermediate findings, are always kept in an encrypted format on a non-HDFS file system within the cluster. Using a parallel computing technique, data has been divided into smaller chunks at the client level, and encrypted storage has been made at HDFS. Additionally, encrypted data can be directly used for Map Reduce jobs, which can be decrypted before processing by importing the relevant decryption library. A MapReduce job divides its input into fixed-size chunks. These are the input split portions of the input used by one map alone. The input data is processed in decrypted form at the Map function, while the output data is saved in encrypted form in the HDFS. After the intermediate results of the Map function are decrypted, the Reduce function is applied, and the final encrypted data is placed in HDFS which is only accessible by authorized clients. The encryption and decryption processes are identical, and they are both easy to use, affordable, and scalable without sacrificing functionality, performance, or scalability. As a result, they are simple to suggest and successfully solve huge data cluster security flaws. In this proposed methodology, we used the hive query language with UDF and different compression-type storage file formats to provide dual-level security at REST and Motion throughout the network. Serialization-deserialization and compression-decompression are easily done in various file formats, so the researcher shouldn't be worried about serialization-deserialization and compression-decompression.

In this complex approach, the researcher uses encryption to protect the data and compression to provide an extra layer of data hiding, thereby addressing various data security issues in a distributed environment. The first layer of XOR-Onetime pad encryption is applied. XOR-Onetime pad resistant to brute-force attacks and helps mask data before it is stored in HDFS. This first lightweight layer provides security against attacks within distributed storage. The second layer AES encryption provides bulk-level data security. AES offers high security against advanced cryptographic attacks such as pattern and frequency analysis within distributed storage. These encryption layers ensure if the lower XOR encryption layer is bypassed, the AES layer protects the data by its complex algorithms from unauthorized users.

Further, enhancing the security of the above XOR-Onetimepad with AES combined with various file formats and compression methods. These formats maximize both storage and security by structuring data in ways that decrease vulnerability to distinctive attack vectors, such as pattern recognition. Compression not only reduces storage size but also makes data less predictable and tougher to study by attackers, as patterns and redundancy are cut. By combining these security measures—XOR-One-time Pad and AES encryption with selective file structuring and compression our solution provides layered protection, making HDFS data safe, space-efficient, and protected to both computational and pattern-based attacks.

3.5. Justifying the Use of XOR-Onetimepad with AES as UDF

XOR-Onetimepad is a lightweight and efficient method used in HDFS. It used bitwise XOR operation for encoding sensitive data segments. XOR-Onetimepad has no strict cryptography requirement unlike AES and DES alone. Its simplicity enables resilience against brute-force assaults if correctly implemented with random keys, which is especially ideal for safeguarding smaller, high-frequency data transfers inside HDFS. XOR-Onetime pad requires minimal computer resources, which enables fast data transmission and reduces latency. This characteristic is beneficial in the HDFS environment. By reducing processing time and resources XOR-onetime pad supports rapid efficient encryption, helping maintain high performance even in data-intensive operations. This encryption efficiency does not degrade the overall system throughput where speed is the top priority.

The choice of XOR-One-time Pad over more prevalent cryptographic standards like DES or RSA comes from its lowest computation demands, aligning with HDFS's design for high-speed, scalable processing. Furthermore, XOR-One-time Pad's simplicity makes it less subject to the overhead and limits on resources that other algorithms may offer, thus boosting speed without sacrificing fundamental security.

XOR-Onetimepad operates through bitwise XOR so it is computationally inexpensive. XOR-Onetimepad quickly encrypts and decrypt the data segments, even with a large sequential data flow in a distributed environment. It provides a fundamental layer of security for unauthorized access. Its simplicity allows it to secure high-frequency data exchange without compromising its performance. XOR-Onetimepad is ideal for an HDFS environment based on speed and scalability.

AES resilience against frequency analysis and cryptographic analysis attacks. Its existence in hybrid paradigms provides a solid layer of encryption for the bulk of data in HDFS.AES safeguards massive datasets from sophisticated cryptographic attacks that XOR alone cannot withstand. Integrate as a hybrid, satisfies as a twin need for fast selective and complete protection of HDFS data.

AES protects against brute force and frequency analysis attacks. Its ability to handle large bulk of data where large amounts of sensitive data are stored. AES's responsiveness lets it maintain performance while protecting complex information, which is essential in big data instances that balance the huge flow of data and strict safety requirements.

4. Key Management Strategies

Key management strategies address the dual-level encryption by focus three main areas such as

4.1. Key Generation

A unique key is generated for encryption. These keys must be random, robust, and securely managed to prevent data.

4.2. Key Rotation

This phase involves updating encryption keys at regular basis to reduce the risk of potential key compromise. Regular basis rotation reduces the lifespan of any single key. In this way, key rotation-based concepts, key management strategies reduce the risk of unauthorized access to the data in a distributed environment. Key rotation makes system to resilient against brute-force attacks and key reuse vulnerability in cloud environment.

4.3. Key Distribution

In this phase share the keys across the nodes and authorized users within the HDFS. The key distribution across the nodes is a very challenging process due to the network weakness. In the proposed approach, HDFS data is prevented through network weakness and unauthorized access. This approach provides robust key management, enhancing performance, and security by use of compression techniques and encryption methods.

5. Implementation Complexity and Resource Implication

Table 4. below is used to describe the implementation complexity and resource Implication by considering the various factors.

Table 4. Describes the implementation complexity and resource Implication by considering the various factors.

<table>
<tr><th>Aspect</th><th>Description</th><th>Impact on System</th><th colspan="2">Complexity of Each encryption and double level encryption</th></tr>
<tr><td rowspan="3">Encryption Techniques</td><td rowspan="3">Combine XOR-Onetimepad with AES</td><td rowspan="3">XOR-onetime pad offers quick transfer for high-frequency encryption and AES provides robust encryption bulk of data</td><td>XOR-Onetimepad</td><td>O(n)</td></tr>
<tr><td>AES</td><td>O(n*r)</td></tr>
<tr><td>XOR-Onetimepad with AES</td><td>O(n)+O(n*r)</td></tr>
<tr><td>Implementation Complexity</td><td>Combine XOR-Onetimepad with AES as a double layer XOR-Onetimepad simple to implement but AES increases the overhead burden due to complex structure and the key rounds</td><td colspan="3">Enhance the security by the dual-level encryption</td></tr>
<tr><td>Computational Burden</td><td>Increase the CPU burden due to the double layer of encryption</td><td colspan="3">Increase the CPU overhead cost and system latency</td></tr>
<tr><td>Resource Usage</td><td>Dual-level encryption increases the processing time and memory usage</td><td colspan="3">Increase the processing time and resource usage so the resources are carefully located</td></tr>
<tr><td>Compression Techniques</td><td>Different types of storage file formats reduce the storage size of data and efficient access of data through network is possible.</td><td colspan="3">Reduce the storage cost and easily accessing large data is possible</td></tr>
</table>

Performance Impact	Experiment results show that ORC+Zlib with Dual-level encryption gives better results compared to 2DES	Increase the efficiency for large data handling and improve the query processing time.
Query handling by Hive	Integrate the Hive to easily access large data without concentrating the map-reduce job	High query processing time with the hive in large-scale data processing environment.
Scalable in Distributed system	Support in large-scale processing environments between multiple nodes	Provide scalability with the use of optimal resources for consistent performance

6. Result and Discussion

Experiments on a Hadoop cluster running on an i7 CPU running at 2.80 GHz with 16 GB of RAM, 64-bit OS, and installed via Cloudera VM have validated the proposed encryption approach for large data security in the HDFS environment. Other nodes are chosen as Datanodes, and one node is designated as the master node. Different-size files [41] such as 8 KB, 11 KB, and 16 KB are utilized to assess the performance of the encryption and decryption processes. Encryption and decryption times, the throughput of encryption, and decryption, and total map-reduce time for encryption and decryption are among the parameters that are compared with the proposed methodology.

6.1. *Encryption Time of Proposed Methodology*

The encryption time is obtained by calculating the time taken to generate the cipher text from the plain text. It is observed in Fig. 7., Fig. 8, and Fig. 9. that the encryption time gradually increases when the file size is too long. Fig. 7. shows the result of encryption with ORC file format and its different compression techniques such as NONE, SNAPPY, and ZLIB, with proposed UDF (XOR-Onetime pad with AES). The result of encryption is based on the Parquet file format and its different compression techniques such as UNCOMPRESSED, SNAPPY, and GZIP with UDF shown in Fig. 8. The encryption time result based on the Avro file format and its various compression methods, including UNCOMPRESSED, SNAPPY, and DEFLATE with UDF, is displayed in Fig. 9.

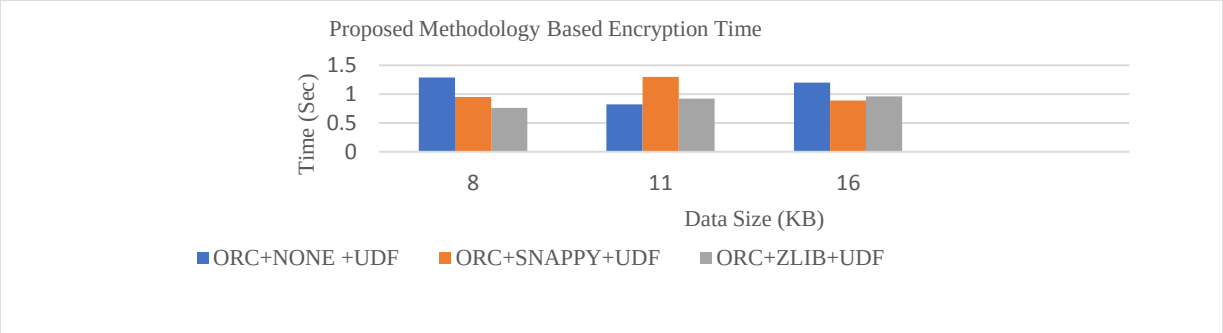

Fig. 7. Encryption Time-Based Comparison Between ORC File Format and its Different Compression Techniques such as NONE, SNAPPY, and ZLIB, with Proposed UDF (XOR-Onetime pad with AES)

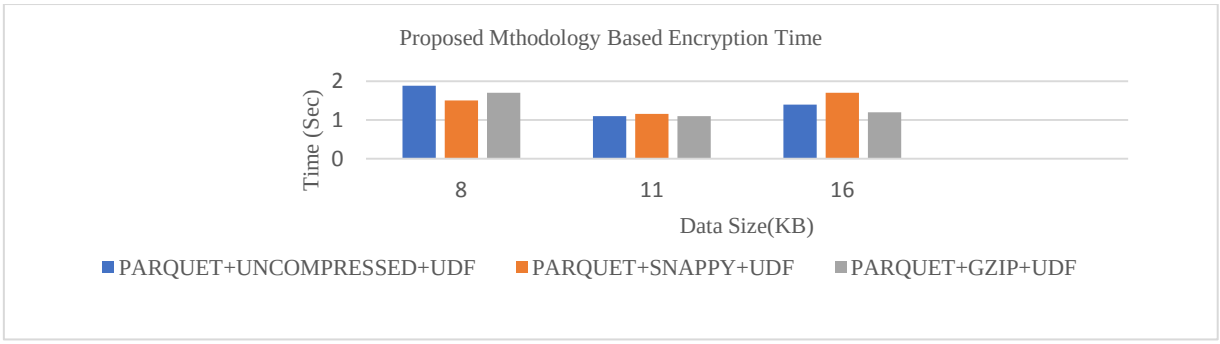

Fig. 8. Encryption Time-Based Comparison between PARQUET File Format and its Different Compression Techniques such as UNCOMPRESSED, SNAPPY, and GZIP with Proposed UDF (XOR-Onetime pad with AES)

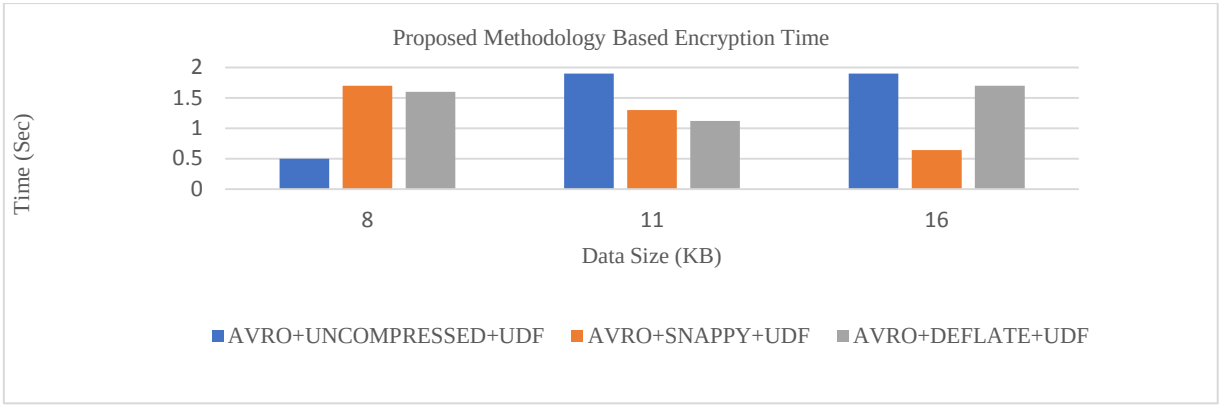

Fig. 9. Encryption Time-Based Comparison between AVRO File Format and its Different Compression Techniques such as UNCOMPRESSED, SNAPPY, and DEFLATE with Proposed UDF (XOR-Onetime pad with AES)

6.2. Decryption Time of Proposed Methodology

The decryption time is obtained by calculating the time taken to generate the plain text from the cipher text. It is observed in Fig. 10., Fig. 11., and Fig. 12. The outcome of decrypting an ORC file format using several compression methods, including NONE, SNAPPY, and ZLIB, is displayed in Fig. 10. with the proposed UDF (XOR- Onetime pad with AES). Fig. 11. shows the result of decryption based on the parquet file format and its different compression techniques such as UNCOMPRESSED, SNAPPY, and GZIP with UDF. Fig. 12. displays the result of decryption time depending on Avro file format and its various compression strategies, such as UNCOMPRESSED, SNAPPY, and DEFLATE with UDF.

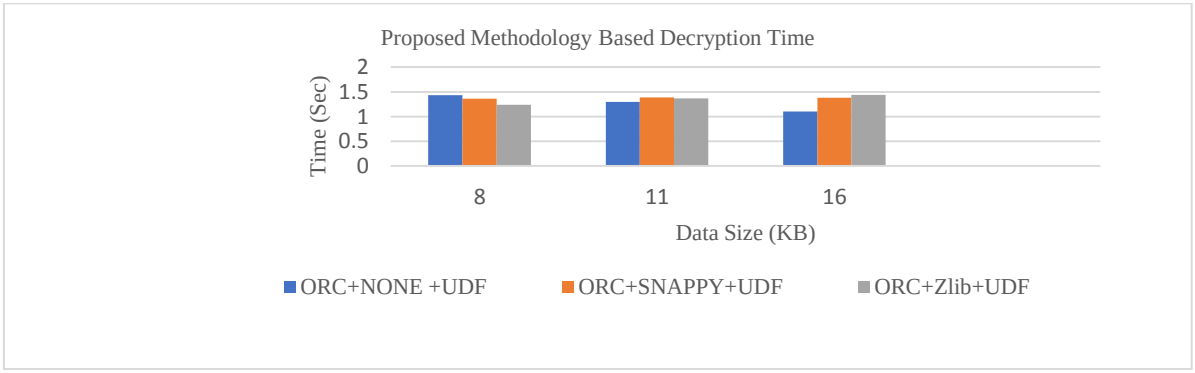

Fig. 10. Decryption Time-Based Comparison between ORC File Format and its Different Compression Techniques such as NONE, SNAPPY, and ZLIB, with Proposed UDF (XOR-Onetime pad with AES)

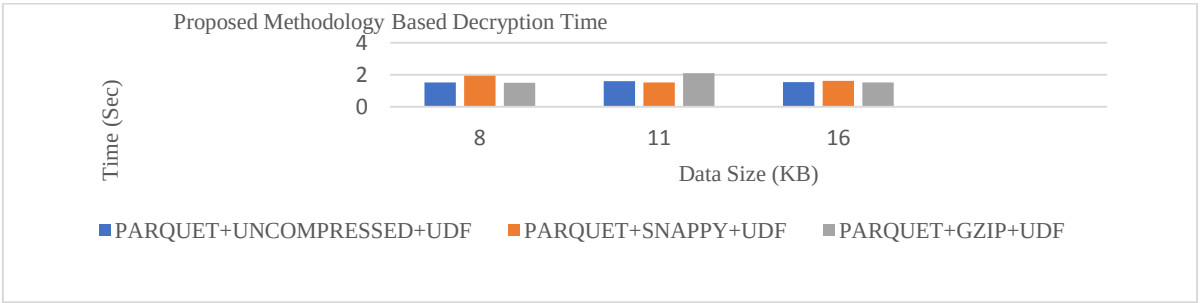

Fig. 11. Decryption Time-Based Comparison between PARQUET File Format and its Different Compression Techniques such as UNCOMPRESSED, SNAPPY, and GZIP with Proposed UDF (XOR-Onetime pad with AES)

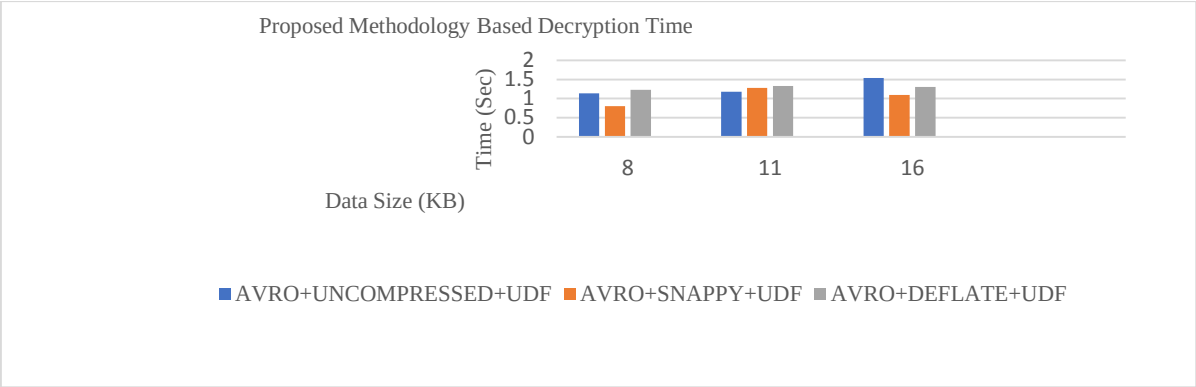

Fig. 12. Decryption Time-Based Comparison between AVRO File Format and its Different Compression Techniques such as UNCOMPRESSED, SNAPPY, and DEFLATE with Proposed UDF (XOR-Onetime pad with AES)

6.3 Throughput

Throughput is obtained by calculating the size of the file divided by the time for encryption and the size of the file divided by the decryption time. Fig. 13. shows the throughput based on encryption time and the throughput based on decryption time is shown in Fig. 14.

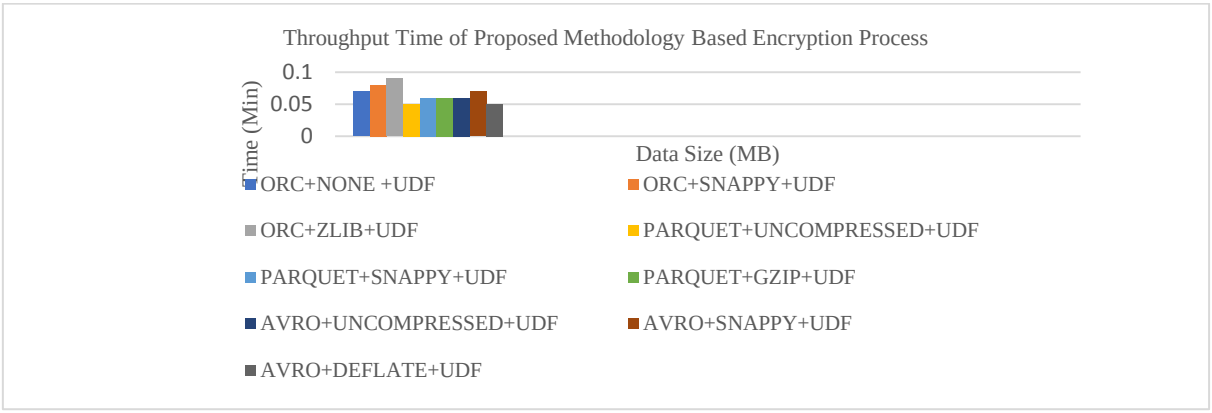

Fig. 13. Throughput Time of Encryption Process Comparison between Different Compression types of Storage File Format with Proposed UDF (XOR-Onetime pad with AES)

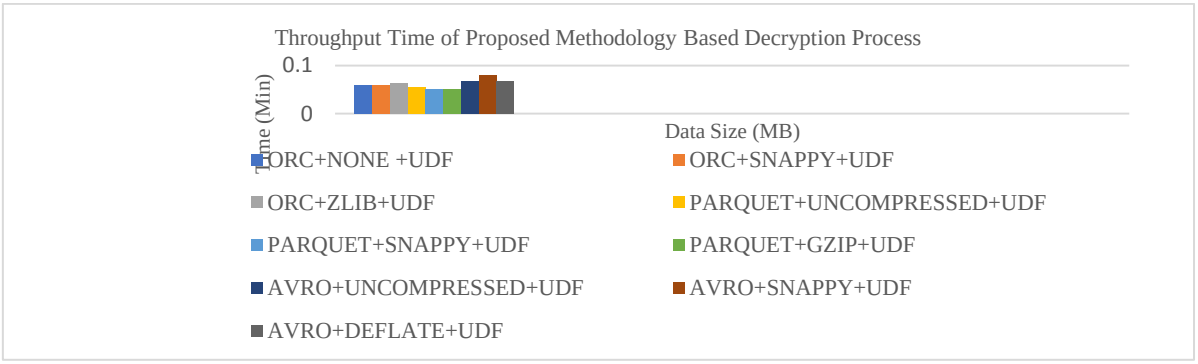

Fig. 14. Throughput Time of Decryption Process Comparison between Different Compression types of Storage File Format with Proposed UDF (XOR-Onetime pad with AES)

Table 5., and Table 6. compare the results of the proposed methodology based on encryption and decryption time. It proves that the XOR-Onetime pad with AES combined with ORC Zlib file format is a better approach than the other methods. Based on the suggested methodology Tables 7., and Table 8. present the overall map-reduce time of the encryption and decryption process.

Table 5. Proposed Methodology based File Encryption Performance Comparison

File Size	ORC-based **Proposed Methodology**			Parquet-based **Proposed Methodology**			Avro-based **Proposed Methodology**			2DES
KB	None (Sec)	Snappy (Sec)	Zlib (Sec)	Uncompressed (Sec)	Snappy (Sec)	Gzip (Sec)	Uncompressed (Sec)	Snappy (Sec)	Deflate (Sec)	(Sec)
8	1.29	0.95	0.76	1.88	1.50	1.70	0.50	1.70	1.60	6.4
11	0.82	1.30	0.92	1.1	1.16	1.10	1.90	1.30	1.12	7.2
16	1.20	0.89	0.96	1.40	1.70	1.20	1.90	0.64	1.70	10.2
Throughput (MB/Min)	0.07	0.08	0.09	0.05	0.06	0.06	0.06	0.07	0.05	0.01

Table 6. File Decryption Performance Comparison Using the Suggested Methodology

File Size	ORC-based **Proposed Methodology**			Parquet-based **Proposed Methodology**			Avro-based **Proposed Methodology**			2DES
KB	None (Sec)	Snappy (Sec)	Zlib (Sec)	Uncompressed (Sec)	Snappy (Sec)	Gzip (Sec)	Uncompressed (Sec)	Snappy (Sec)	Deflate (Sec)	(Sec)
8	1.43	1.36	1.24	1.51	1.93	1.49	1.14	0.80	1.23	6.3
11	1.30	1.39	1.37	1.59	1.52	2.10	1.18	1.28	1.33	7.5
16	1.10	1.38	1.44	1.54	1.61	1.52	1.54	1.1	1.31	10.4
Throughput (MB/Min)	0.06	0.06	0.064	0.056	0.051	0.051	0.067	0.08	0.067	0.01

Table 7. Total Map-Reduce Time for Encryption based on the Proposed Methodology

File Size	ORC-based **Proposed Methodology**			Parquet-based **Proposed Methodology**			Avro-based **Proposed Methodology**		
KB	None (msec)	Snappy (msec)	Zlib (msec)	Uncompressed (msec)	Snappy (msec)	Gzip (msec)	Uncompressed (msec)	Snappy (msec)	Deflate (msec)
8	1290	950	760	1880	1500	1700	500	1700	1600
11	820	1300	920	1100	1160	1100	1900	1300	1120
16	1200	890	960	1400	1700	1200	1900	640	1700

Table 8. Proposed Methodology based on Total Map-Reduce Time for Decryption

File Size	ORC-based **Proposed Methodology**			Parquet-based **Proposed Methodology**			Avro-based **Proposed Methodology**		
KB	None (msec)	Snappy (msec)	Zlib (msec)	Uncompressed (msec)	Snappy (msec)	Gzip (msec)	Uncompressed (msec)	Snappy (msec)	Deflate (msec)
8	1430	1360	1240	1510	1930	1490	1140	800	1230
11	1300	1390	1370	1590	1520	2100	1180	1280	1330
16	1100	1380	1440	1540	1610	1520	1540	1100	1310

Table 9. shows the suggested methodology-based performance comparison

Metric	Proposed Methodology	2DES	Other methods	% Improvement with the proposed Methodology
Encryption time	Mean 0.77 sec +-0.03	1.22 sec+-0.03	1.04 sec +- 0.05	9 -10 % reduction in encryption time
Decryption time	Mean 1.09 sec +-0.05	1.62 sec +-0.04	1.32 sec +- 0.07	9-10 % reduction in decryption time
throughput	Mean 0.07 MB/min +-0.02	0.07 MB/sec +- 0.01	0.08 MB/sec +-0.01	10-15 % increase the throughput
Map-reduce time	940 msec +- 50	1200 msec +- 70	1200 +- 50	9-10% improve processing time

Table 9. shows that XOR-Onetimepad with AES improved encryption-decryption times by 9-10 % over 2DES and higher throughput. ($p<0.05$ for all comparisons).

7. Trade-offs and Strategies for Balancing between Performance and Security

The proposed method for safeguarding data in HDFS involves combining XOR-One-time pad and AES encryption techniques with compression-type storage file formats (ORC, Parquet, and Avro). This approach aims to strike a balance between security and performance. However, it requires a balance between security and performance. The use of XOR-One-time pad and AES encryption creates a strong dual layer of encryption, making it harder for malicious actors to access data. However, this method also increases computing overhead, potentially affecting system performance during data input and retrieval. Key management is crucial for ensuring the security of both encryption methods. Compression techniques like ORC, Parquet, and Avro optimize storage and query efficiency but should be implemented before encryption to prevent pattern recognition attacks. However, compression can cause latency and impair query performance. High data volumes require strong encryption to prevent large-scale breaches, and scalability concerns arise. Algorithm selection is crucial for balancing security and performance. Strategies to optimize algorithms include optimizing versions of the XOR-One-time pad and AES algorithms, developing efficient key management, applying selective encryption, leveraging parallel processing, exploring hybrid encryption approaches, and using adaptive compression techniques. In conclusion, the proposed dual encryption method offers a solid solution for safeguarding data in HDFS, but trade-offs must be considered.

8. Case Study

A healthcare company manages sensitive patient data sets, including medical records, treatment history, and personal data, to ensure data security and privacy. To meet regulatory requirements like HIPAA, the company adopted a proposed encryption solution that combines compression-type storage file formats (ORC, Parquet, and Avro) with AES and XOR-One-time pad encryption algorithms in their Hadoop Distributed File System (HDFS). The company classified data based on sensitivity, using dual encryption for sensitive data like medical records and patient identities. The encryption method used XOR-One-time Pad and AES encryption for maximum secrecy. Data was compressed using ORC, Parquet, or Avro formats, reducing storage footprint and improving query performance. An advanced key management system was implemented to manage key production, distribution, and rotation. Hardware acceleration was used to optimize encryption and decryption procedures. The dual encryption method enhanced data security, demonstrated compliance with HIPAA, and improved performance. The combination of encryption and compression ensured efficient storage usage, reducing costs and expediting data recovery.

9. Limitations and Future Work

The proposed approach of integrating an XOR-One-time pad with AES encryption algorithms and compression-type storage file formats for data protection in HDFS has shown promising results. However, it faces several constraints, including performance complexity, key management complexity, scalability issues, compatibility concerns, security flaws, and maintenance and upkeep. The dual encryption process adds computational complexity, which can be problematic in resource-limited contexts or dealing with large datasets. The proposed encryption technologies may require major adjustments to current systems, and any flaws in implementation or integration can pose risks. Regular updates and maintenance are necessary to ensure the security and efficiency of the method. Future work should focus on optimizing encryption algorithms, enhancing key management solutions, conducting scalability testing, integrating encryption algorithms with developing technologies, conducting security audits and penetration testing, developing user-friendly tools, and exploring hybrid approaches that combine encryption and compression techniques. By addressing these constraints, the proposed approach can be further enhanced to provide a more robust, efficient, and secure solution for preserving sensitive data in HDFS by AES-CBC modes. AES-CBC mode is more secure but increases the computational overhead, so in this research, AES-ECB mode is used to reduce the computational cost in a distributed environment. AES-CBC has randomization features. The plaintext block XORed with ciphertext block, so reduces the risk of pattern recognition but increases the computational cost.

10. Conclusion

This paper provides a proposed UDF based on (an XOR-Onetime pad with AES) encryption and decryption techniques with different compression-type storage file formats to provide two levels of security of sensitive data in a Hadoop environment. We have focused on encrypting sensitive data and whole file data within the Hadoop. Hive has properties to read and write the data into HDFS via a user-defined function so the researcher does not face the problem of building the map-reduce jobs. Additionally, hive uses different storage file formats and compression techniques to store and move data easily so that the decreases load time and improve query processing time of large data sets. The above approach used the ETL (Extract-Transform-Load) pipeline method based on HDFS to Hive. The above novel method results compared to the 2DES and other methods prove that the proposed methodology based upon UDF (XOR-Onetime pad with AES) with ORC Zlib file format is more beneficial and increases the performance by 9-10%. It also

provides dual-level security for sensitive data. In a real-world scenario, our main aim is to provide the security of sensitive data storage level and transit level in the distributed environment. The suggested model's effectiveness was confirmed in terms of encryption time, decryption time, throughput time for the encryption and decryption process, and total map-reduce time for encryption and decryption.

Statements and Declarations

- ☑ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.
- ☑ The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.
- ☑ The authors of this paper have directly participated in the planning, execution, or analysis of this study.
- ☑ The authors of this paper have read and approved the final manuscript.

References

[1] Moreno, J., Serrano, M., & Fernández-Medina, E. (2016). Main Issues in Big Data Security. Future Internet, 8(3), 44. https://doi.org/10.3390/fi8030044

[2] Yaqoob, I., Hashem, I. a. T., Gani, A., Mokhtar, S., Ahmed, E., Anuar, N. B., & Vasilakos, A. V. (2016). Big data: From beginning to future. International Journal of Information Management, 36(6), 1231–1247. https://doi.org/10.1016/j.ijinfomgt.2016.07.009

[3] Garlasu, D., Sandulescu, V., Halcu, I., Neculoiu, G., Grigoriu, O., Marinescu, M., & Marinescu, V. (2013). A big data implementation based on Grid computing. https://doi.org/10.1109/roedunet.2013.6511732

[4] Gandomi, A., & Haider, M. (2015). Beyond the hype: Big data concepts, methods, and analytics. International Journal of Information Management, 35(2), 137–144. https://doi.org/10.1016/j.ijinfomgt.2014.10.007

[5] Chen, C. P., & Zhang, C. Y. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. Information Sciences, 275, 314–347. https://doi.org/10.1016/j.ins.2014.01.015

[6] Rodríguez-Mazahua, L., Rodríguez-Enríquez, C. A., Sánchez-Cervantes, J. L., Cervantes, J., García-Alcaraz, J. L., & Alor-Hernández, G. (2015). A general perspective of Big Data: applications, tools, challenges and trends. the Journal of Supercomputing/Journal of Supercomputing, 72(8), 3073–3113. https://doi.org/10.1007/s11227-015-1501-1

[7] Hashem, I. a. T., Yaqoob, I., Anuar, N. B., Mokhtar, S., Gani, A., & Khan, S. U. (2015). The rise of "big data" on cloud computing: Review and open research issues. Information Systems, 47, 98–115. https://doi.org/10.1016/j.is.2014.07.006

[8] Chen, M., Mao, S., & Liu, Y. (2014). Big Data: A Survey. Journal on Special Topics in Mobile Networks and Applications/Mobile Networks and Applications, 19(2), 171–209. https://doi.org/10.1007/s11036-013-0489-0

[9] Thuraisingham, B. (2015). Big Data Security and Privacy. https://doi.org/10.1145/2699026.2699136

[10] Van Rijmenam, M. Think Bigger.(2014) AMACOM.

[11] Polato, I., Ré, R., Goldman, A., & Kon, F. (2014). A comprehensive view of Hadoop research—A systematic literature review. Journal of Network and Computer Applications, 46, 1–25. https://doi.org/10.1016/j.jnca.2014.07.022

[12] Balusamy, B., R, N. a., Kadry, S., & Gandomi, a. H. (2021)Big data. John Wiley & Sons. Wiley.

[13] Apache Hadoop 3.3.6: HDFS Architecture [Internet]. Available from: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[14] Bansal, K., Chawla, P., & Kurle, P. (2018). Analyzing Performance of Apache Pig and Apache Hive with Hadoop. In Lecture notes in electrical engineering (pp. 41–51). https://doi.org/10.1007/978-981-13-1642-5_4

[15] Priya S. (2019) Big data: analytics, technologies, and applications.

[16] Wadhera, S., Kamra, D., Kumar, A., Jain, A., & Jain, V. (2021). A systematic Review of Big data tools and application for developments. 2021 2nd International Conference on Intelligent Engineering and Management (ICIEM). https://doi.org/10.1109/iciem51511.2021.9445326

[17] Apache Hive [Internet]. Available from: https://hive.apache.org/

[18] Naidu V. Performance enhancement using appropriate file formats in big data Hadoop ecosystem. International Research Journal of Engineering and Technology (IRJET).

[19] Morán, J., De La Riva, C., & Tuya, J. (2015). Testing data transformations in MapReduce programs. https://doi.org/10.1145/2804322.2804326

[20] Cloudera Document [Internet]. Available from: https://docs.cloudera.com

[21] Song, N. Y., Shin, N. Y. S., Jang, N. M., & Chang, N. J. W. (2017). Design and implementation of HDFS data encryption scheme using ARIA algorithm on Hadoop. https://doi.org/10.1109/bigcomp.2017.7881720

[22] Mahmoud, H., Hegazy, A., & Khafagy, M. H. (2018). An approach for big data security based on Hadoop distributed file system. https://doi.org/10.1109/itce.2018.8316608

[23] Kamaruzaman, S. H., Nik, W. N. S. W., Mohamed, M. A., & Mohamad, Z. (2018). Design and Implementation of Data-at-Rest Encryption for Hadoop. International Journal of Engineering & Technology, 7(2.15), 54. https://doi.org/10.14419/ijet.v7i2.15.11212

[24] Teng, L., Li, H., Yin, S., & Sun, Y. A modified advanced encryption standard for data security. International Journal of Network Security.

[25] Sonal Jain, Mohit Jain. (2019) Privacy Preserving mining using data encryption scheme for Hadoop Ecosystem. International Journal for Rapid Research in Engineering Technology & Applied Science.

[26] Khafagy, O. H., Ibrahim, M. H., & Omara, F. A. (2020). Hybrid-Key Stream Cipher Mechanism for Hadoop Distributed File System Security. https://doi.org/10.1109/itce48509.2020.9047775

[27] Jayapandian, N. (2020). SECURING CLOUD DATA AGAINST CYBER-ATTACKS USING HYBRID AES WITH MHT ALGORITHM. Computing, 561–568. https://doi.org/10.47839/ijc.19.4.1989
[28] Gattoju, S., & Nagalakshmi. (2021). AN EFFICIENT APPROACH FOR BIGDATA SECURITY BASED ON HADOOP SYSTEM USING CRYPTOGRAPHIC TECHNIQUES. Indian Journal of Computer Science and Engineering, 12(4), 1027–1037. https://doi.org/10.21817/indjcse/2021/v12i4/211204132
[29] Kaushik, A., & Srivastava, V. K. (2020). Performance Analysis Of AES And DESede On The Sensitive Data Stored In HDFS. 2020 2nd International Conference on Advances in Computing, Communication Control and Networking (ICACCCN). https://doi.org/10.1109/icacccn51052.2020.9362755
[30] Algaradi, T., & Rama. (2021) A new encryption scheme for performance improvement in big data environment using MapReduce. Journal of Engineering Science and Technology.
[31] Mohanraj, T., & R. Santhosh. (2022) Hybrid encryption algorithm for big data security in the Hadoop distributed file system. Computer Assisted Methods in Engineering and Science.
[32] Khadji, Khoulji, & Kerkeb, M. (2023) Efficient Big Data security: Evaluating the performance of a proposed hybrid key management algorithm using lightweight cryptography. Journal of Theoretical and Applied Information Technology.
[33] Ramya, P., & Sundar, C. (2020). SecDedoop: Secure Deduplication with Access Control of Big Data in the HDFS/Hadoop Environment. Big Data, 8(2), 147–163. https://doi.org/10.1089/big.2019.0120
[34] Gupta, M., & Dwivedi, R. K. (2023). Fortified MapReduce Layer: Elevating Security and Privacy in Big Data. ICST Transactions on Scalable Information Systems. https://doi.org/10.4108/eetsis.3859
[35] Iavich, M., & Kevanishvili, Z. Modified one time pad. ResearchGate. Scientific and Practical Cyber Security Journal (SPCSJ). 2018;
[36] Fathy, A., Tarrad, I. F., Hamed, H. F. A., & Awad, A. I. (2012). Advanced Encryption Standard Algorithm: Issues and Implementation Aspects. In Communications in computer and information science (pp. 516–523). https://doi.org/10.1007/978-3-642-35326-0_51
[37] Rihan, S., Khalid, & F. Oshman, S. A Performance Comparison of Encryption Algorithms AES and DES. International Journal of Engineering Research & Technology, vol 4(12). International Journal of Engineering Research & Technology. 2015;
[38] LanguageManual - Apache Hive - Apache Software Foundation [Internet]. Available from: https://cwiki.apache.org/confluence/display/hive/languagemanual
[39] Ngo, S. Compressing Parquet Files: A Basic Guide [Internet]. Assisty - Shopify Data Analytics Solution. Available from: https://assisty.ai/compressing-parquet-files-a-basic-guide/
[40] Apache Avro Data Source Guide - Spark 2.4.0 Documentation [Internet]. Available from: https://spark.apache.org/docs/2.4.0/sql-data-sources-avro.html
[41] Kaggle: your machine learning and data science community [Internet]. Available from: https://kaggle.com/.
[42] Arshad, M. J., Department of Computer Science, Virtual University of Pakistan, Lahore, Umair, M., Munawar, S., Naveed, N., & Naeem, H. (2020). Improving cloud data encryption using customized genetic algorithm. International Journal of Intelligent Systems and Applications, 12(6), 46–63. https://doi.org/10.5815/ijisa.2020.06.04

Authors' Profiles

Shivani Awasthi received her B. Tech. and M. Tech. degrees in Computer Science and Engineering from Dr. A.P.J. Abdul Kalam Technical University, Lucknow, and SHAUTS, Allahabad. She is pursuing a Ph.D. in Computer Science and Engineering at Harcourt Butler Technical University, Kanpur, India. Her research areas are Big data and Cloud Computing.

Narendra Kohli received his Ph.D. from IIT Kanpur, India. He is currently working as a professor at HBTU, Kanpur. His research interests include Machine Learning, Digital Image Processing, and Database Management systems.

How to cite this paper: Shivani Awasthi, Narendra Kohli, "Hive-Based Data Encryption for Securing Sensitive Data in HDFS", International Journal of Mathematical Sciences and Computing(IJMSC), Vol.10, No.4, pp. 34-50, 2024. DOI: 10.5815/ijmsc.2024.04.04