A Multilayered Approach to Enhance Cloud Security using Homomorphic, AES, and Hashgraph

Ayush Verma, Tanuj Chandela, Geetanjali Rathee
Netaji Subhash University of Technology, Dwarka
New Delhi-110078, India
{integralayush, tanujchandela, geetanjali.rathee123}@gmail.com

***Abstract*—The rise of cloud technology is a big deal for how we share and access data together. It makes working together easier and opens up a ton of new possibilities. But with all this sharing, we need to make sure our information stays safe and that everyone follows the rules we've agreed upon for how services should work. Blockchain technology seems like a good way to keep track of these rules by recording everything in a secure and unchangeable way. However, the usual blockchain systems have some weaknesses. They can still be attacked in ways that could disrupt services, like with DDoS attacks. Plus, the way blockchain reaches agreements can slow things down. However, managing SLAs itself does not ensure the security of the data and user's privacy. Various solutions have been proposed, but none comprehensively address all the issues associated cloud environment. This paper introduces a framework constructed using Hashgraph-based distributed ledger technology to enhance scalability, security, and performance in the tamper-proof logging of all events through smart contracts. This structure aids in detecting points of failure and is applicable for automatic Service Level Agreement (SLA) verification. To safeguard user privacy, protect data from intruders, and ensure semantic security, we have implemented double-layer encryption. A homomorphic encryption technique is employed to preserve user privacy, allowing computations to be performed on the encrypted data. Additionally, AES (Advanced Encryption Standard) is used for secure transportation over an open network to prevent potential attacks such as known-plain-text attacks. The performance of our framework was assessed in terms of latency, CPU usage, and memory usage, while the security aspect was conventionally analyzed.**

I. INTRODUCTION

Cloud computing is essentially about renting resources, hosting applications, and outsourcing services. The system simplifies resource sharing and computation costs, which is a significant improvement. The cloud computing model allows users to access and use a variety of computing resources, such as storage, processing power, and databases, via the internet. This on-demand availability of resources provides scalability, flexibility, and cost-efficiency, making it an appealing solution for both individuals and businesses alike. [1] Cloud services are typically divided into three categories: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), which give users varying levels of control and management over their computing environments.The cloud computing paradigm has transformed how technology is used, enabling innovations in fields as diverse as data analytics, artificial intelligence, and the Internet of Things (IoT).

In recent years, the widespread adoption of cloud services, spearheaded by industry giants such as Google, Amazon, and Microsoft, has encountered notable security challenges. This surge in usage has inevitably resulted in security lapses, and a key challenge revolves around establishing accountability in the cloud. This predicament poses a dilemma for both users and service providers. While the issue of accountability isn't unique to cloud services, it remains a pervasive challenge across various service providers. Instances of data loss or incorrect computation results raise critical questions about responsibility. Notably, major companies like Microsoft and Amazon have devised their policies to address these challenges. For example, [1] Microsoft's Service Level Agreements (SLAs) incorporate a provision designating the service provider (Microsoft) as responsible for adjudicating disputes initiated by clients. In contrast, Amazon Web Services adopts a more detached stance. Their general customer agreement explicitly disclaims any form of compensation for a broad spectrum of failures, effectively sidestepping the need to pinpoint responsibility.

A. Motivation

Cloud computing brings multiple issues, including worries about data privacy and the occurrence of cyber threats such as Distributed Denial of Service (DDoS), Sybil attacks, known plaintext assaults, eavesdropping attacks, etc. These problems underscore the complex terrain that both customers and service providers must negotiate in the rapidly changing world of cloud technology. Encryption methods and blockchain technology have been introduced autonomously to enhance security and trust, yet neither satisfies all requirements.

B. Contribution

In this paper, we aim to provide a framework that combines Hedera Hashgraph with a double-layered encryption mechanism to fortify our structure against various cloud computing assaults. This strategic integration attempts to efficiently maintain confidence between users and service providers while also improving security. The framework employs an Elliptic Curve Diffie-Hellman key exchange method to generate a shared AES key between a user and an associated company. By using this shared AES key, the company can retrieve data from the server and perform AES decryption, as well as perform

computations on the homomorphically encrypted data of its clients without violating their privacy.

The remaining structure of the paper is organized as follows. Section two deliberates the literature survey while securely transmitting the information. Further, section three elaborates on the proposed methodology in detail. Furthermore, section four analyses the overall performance of the framework over average memory usage, average execution time, and CPU consumption. Next, in section five we provide a conventional security analysis of our framework against various threats. Finally, section six concludes the paper.

C. Hashgraph

The Hashgraph protocol introduces a revolutionary approach to distributed consensus, offering a novel platform for establishing trust among users within a decentralized network. Unlike conventional blockchain structures that require continual pruning of branches to maintain a single coherent chain, Hashgraph seamlessly integrates new transaction data into its ledger. In both blockchain and Hashgraph frameworks, users have the autonomy to generate transactions, which are subsequently organized into containers or blocks and disseminated across the network. However, while blockchain aims to construct a singular, linear chain of blocks, Hashgraph diverges by incorporating every container of transactions into the ledger, thus preventing the need for pruning and ensuring efficiency. This design eliminates the risk of forks and allows for the perpetual existence of all branches, woven together into a cohesive whole. Fig. 1 shows Hashgraph Architecture.

Critically, Hashgraph's resilience to rapid growth distinguishes it from blockchain. While blockchain implementations often struggle to cope with the influx of new containers, leading to potential forks and requiring mechanisms like proof-of-work to artificially slow down growth, Hashgraph seamlessly accommodates rapid expansion without compromising integrity or efficiency. Moreover, Hashgraph's unique properties enable it to provide robust mathematical guarantees, including Byzantine fault tolerance and fairness in transaction ordering. Unlike other distributed database systems like Paxos, which may exhibit Byzantine behaviour without ensuring transaction fairness, and blockchain, which lacks Byzantine tolerance and fairness, Hashgraph offers a comprehensive set of features. By being fair, fast, Byzantine-tolerant, ACID-compliant, efficient, cost-effective, timestamped, and resistant to denial-of-service attacks, the Hashgraph algorithm represents the pinnacle of innovation in distributed consensus mechanisms.

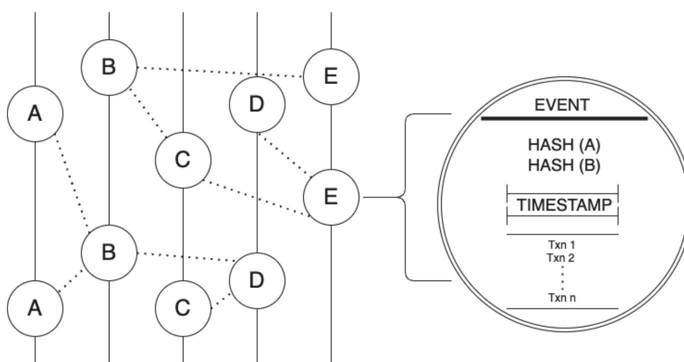

Fig. 1. Hashgraph Architecture Overview

D. Homomorphic Encryption

Homomorphic Encryption (HE) is a unique cryptographic approach that allows a third party to perform computations on encrypted data while maintaining data integrity and meeting the third party's needs without jeopardizing user privacy. This method ensures the confidentiality of sensitive information while allowing an external entity to perform necessary computations. [2]The conventional public key encryption method consists of three algorithms KeyGen (A stochastic algorithm based on a designated security parameter that generates a Private_Key Kprivate and Public_Key Kpublic), Encryptor (takes plaintext P and Kpublic as input and based on this it generates the ciphertext (CT)), and Decryptor (CT and Kprivate is passed as an input and returns the original plaintext). In addition to these algorithms HE additionally have an Evaluator (It takes the function f, CT, and Kpublic and performs the computation on the CT) [3].

Algorithm 1 Working of Homomorphic Algorithm

0: Step 1: (Kprivate, Kpublic) $\leftarrow$ KeyGen(#), {where # is the security parameter}
0: Step 2: Ciphertext(CT) $\leftarrow$ Encryptor(Kpublic, P)
0: Step 3: M $\leftarrow$ func(P), {normally executing the function on the plaintext}
0: Step 4: CT* $\leftarrow$ Evaluator(func, Kpublic, CT)
0: Step 5: R $\leftarrow$ Decryptor(Kprivate, CT*), {As per HE, M = R.}

E. Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) [4] is a widely adopted algorithm globally to safeguard data from unauthorized access. It was originally proposed by Belgian cryptographers Joan Daemen and Vincent Rijmen and later adopted as the Rijndael Algorithm. AES operates as a symmetric-key block cipher, where both the sender and receiver use the same key for encryption and decryption. This approach, approved by the US National Institute of Standards and Technology (NIST), replaced the Data Encryption Standard (DES). [5] AES works with fixed-size data blocks and supports key lengths of 128, 192, and 256 bits. The algorithm uses a substitution-permutation network (SPN) structure, with encryption taking place in multiple rounds depending on the key length. A 128-bit key undergoes 10 rounds of encryption, whereas 192 and 256-bit keys go through 12 and 14 rounds, respectively. The data is subjected to a variety of encryption functions during each round, including SubBytes, ShiftRows, MixColumns, and Round Key Addition.AES provides strong security by repeatedly applying cryptographic functions to

input data, using the initial key and round-specific sub-keys Fig. 2.

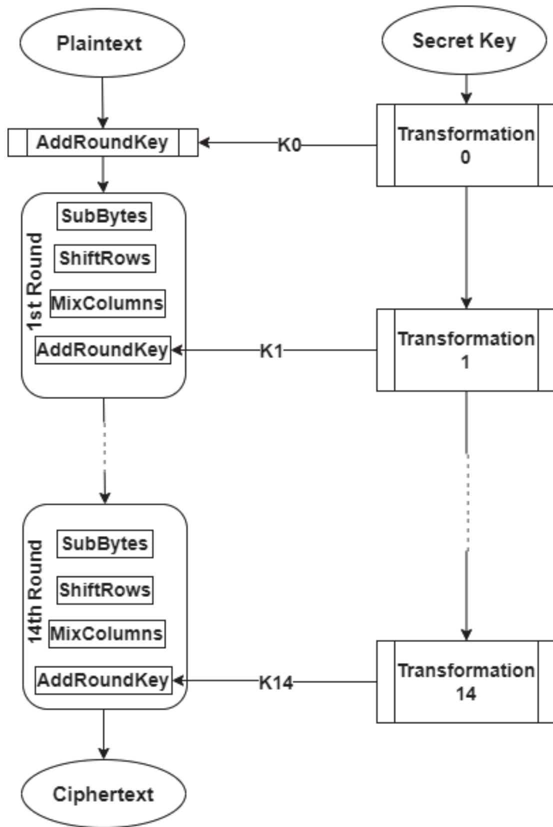

Fig. 2. AES Encryption process

SubBytes: During this phase, each byte in the state is substituted using a predefined substituted box called the S-box. The S-box introduces a nonlinear transformation, which adds complexity to the encryption process.

ShiftRows: During this step, bytes in each row of the state undergo a cyclic shift. This permutation ensures that the resulting output is more random and resistant to pattern analysis.

MixColumns: This operation uses matrix multiplication in a finite field to process each state column independently. It adds more diffusion, ensuring that each byte is influenced by several input bytes.

Round Key Addition: The State performs a bitwise Exclusive-OR operation with a round key. Each round key consists of Nb words generated as part of the Key Expansion process.

Key Expansion: The Key Expansion process generates a sequence of words from the original encryption key. [6] The total number of words produced is calculated as $N_b(N_r+1)$, where Nb represents the number of columns in the state (block size, the count of 32-bit words derived from the initial key), and N_r is the determined number of rounds based on the key size. A set of N_k words (corresponding to 4 bytes words in the original key) is derived directly from the original key. Following this, the remaining words are generated through an iterative process. As a result of the Key Expansion process, an array of four-byte words is produced, denoted as w_i, where the index i ranges from 0 to $N_b(N_r+1)-1$. Each round key employed in the encryption process is constructed by concatenating four words from the Key Expansion output. Thus, for every round, the round key (i) is generated by concatenating four sequential words w_i, w_{i+1}, w_{i+2}, and w_{i+3}.

F. Elliptic Curve Diffie Hellman (ECDH)

Elliptic Curve Diffie Hellman (ECDH) is a shared secret agreement protocol designed to establish a shared secret key between two parties over an insecure network. The security of ECDH relies on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP), which forms the foundation of elliptic curve cryptography. In the elliptic curve cryptosystem, all parties involved must agree on a [7] finite field F_q consisting of integers modulo p, where p is a prime number and q is the order of the finite field. An elliptic curve as shown in Fig. 3 is defined over F_q by the constants a and b used in its defining equation

$$y^2 = x^3 + ax + b$$

(Weierstrass Form). Consider a point P on the curve with order n, which is the smallest positive integer such that nP results in the point at infinity on the curve, serving as the identity element. If Q is another point on the curve, there exists an integer i in the range [0, n-1] such that Q = iP. The challenge lies in computing this integer i.

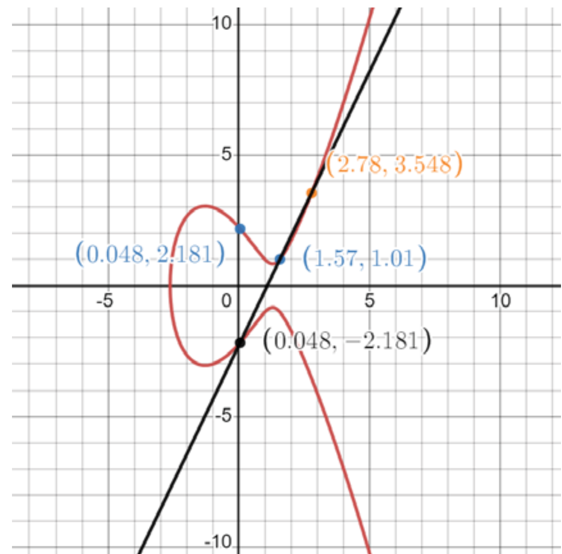

Fig. 3. Example of an Elliptic Curve with a = -5 and b = 5

ECDH operates on a similar principle (Fig. 4). [8] [9] Suppose two parties, A and B, each select a base point G on the curve and randomly choose private keys $Pvt_A = a$ and $Pvt_B = b$ respectively. Using their private keys and the base point G, A and B generate their public keys $Pub_A = aG$ and $Pub_B = bG$, denoted as points A1 and B1 on

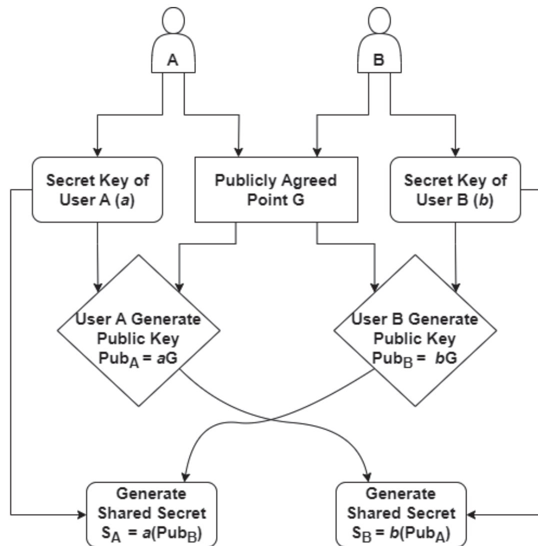

Fig. 4. ECDH Key Agreement Protocol

the curve respectively. These public keys can be exchanged over the insecure network. Upon receiving each other's public keys, A and B independently compute the secret key $S_A = a(Pub_B) = a(bG)$ and $S_B = b(Pub_A) = b(aG)$ respectively, resulting in $S_A = S_B$. ECDH ensures that even if an eavesdropper intercepts the public keys exchanged between A and B, computing the secret key without knowledge of the private keys remains computationally infeasible due to the ECDLP.

II. Related Work

Cloud computing is known for its innovation, resource flexibility, and economies of scale. It enables users to access infrastructure and applications over the Internet, eliminating the need for on-premises installation and maintenance. This has allowed businesses to outsource their operations to the cloud, leading to significant cost savings on storage and computation units for processing extensive databases. However, challenges persist, including issues related to data privacy, efficient data management, interoperability, and accountability. To tackle these challenges, various approaches have been proposed, including:

Mirko Zichichi et al. [1] introduced a prototype demonstrating the application of blockchain technology for dispute resolution in cases of SLA violations through the secure logging of untampered data on the blockchain. The authors implemented their methodology across various blockchain platforms, including GoQuorum, Hyperledger Besu, and Polygon, each leveraging distinct consensus protocols. Their analysis indicates that Polygon and GoQuorum Raft exhibit enhanced performance with reduced response times and minimal error rates. The study employs a standard cloud storage service and outlines a protocol for smart contract logging for each operation.

Akanksha Saini et al. [10] developed an IoT-enabled framework using blockchain technology to address single points of failure in the healthcare system, with a specific focus on the security of Electronic Medical Records (EMRs) controlled by centralized cloud providers. They introduced four smart contracts: user verification, access authorization, misbehaviour detection, and access revocation. Furthermore, the authors described a method for encrypting sensitive EMRs before storing them in the cloud, utilizing cryptographic functions such as Elliptic Curve Cryptography (ECC) and the Edwards-Curve Digital Signature Algorithm (EdDSA). The generated hashes are then stored in the blockchain. The performance evaluation and efficiency analysis of the proposed solution demonstrated faster response times in access.

B. Sowmiya et al. [11] proposed a method to safeguard user data through the use of cryptography and the establishment of a private network using Hyperledger blockchain. The data is divided into sensitive and non-sensitive categories with the help of a linear regression model. The author employs the LECC method and RSA to encrypt sensitive and non-sensitive data, respectively. Furthermore, a modified Spider Optimization Search Algorithm (MSOA) is utilized in conjunction with Linear Elliptical Curve Digital Signature (LECDS) to ensure integrity verification and prevent data loss during communication with the cloud. The author conducted tests on various performance metrics, including security, throughput, classification accuracy, and error rate.

Hamza Javed et al. [12] introduced a model designed to log all events associated with user data in the healthcare sector through cloud services. These logs are securely stored in a private blockchain. To prevent attacks on public healthcare data, the author proposed the implementation of a Cloud Access Security Broker (CASB). This system functions as an additional layer of security by offering an immutable ledger to track user data, effectively preventing unauthorized tampering with sensitive information. The integration of this model aims to mitigate the risk of data loss. The examination of auditing logs using blockchain was conducted by varying the number of patients and actions. Overall, this solution provides secure auditing for health data, leveraging IoT-based monitoring.

Poonam Verma et al. [13] suggest a model based on hashgraph to enhance the access control mechanism in the healthcare system for patient data. By utilizing the gossip protocol in hashgraph, the author aims to reduce energy consumption. Additionally, the paper discusses the improvement in data security, as frameworks built with hashgraph prevent users from Sybil and double spending attacks. The proposed framework is evaluated in terms of time complexity.

Junfeng Tian et al. [14] proposed a security audit scheme to tackle potential security risks in cloud-based shared data storage services used by groups and agents. They incorporated hashgraph technology and devised a Third-Party Medium (TPM) management strategy to ensure security management and lightweight computation for group members. Additionally, to enhance agent security, they set up a virtual TPM pool using TCP sliding window technology, which operates independently. The scheme achieves efficient calculation of

TABLE I
LITERATURE SURVEY

Ref & Author	Consensus	Cryptography	Performance Metrics	Limitation
Mirko Zichichi et al. [1]	IBFT,Clique,PoS,QBFT,Raft	SHA256	Response Time, Throughput	Lack of User's Data privacy
Akanksha Saini et al. [10]	PoS,PoW	EdDSA,ECC	Average Latency	Inefficient Memory Consumption
B. Sowmiya et al. [11]	PBFT,PoET	LECC,RSA,LECDS	Throughput & Error Rate	DDoS and Sybil attacks are possible
Hamza Javed et al. [12]	PoA	CP-ABE	Execution time	Sybil attacks are possible
Poonam Verma et al. [13]	DAG	Hask key Generation	Time Complexity	Not Resistance to Replay Attack
Junfeng Tian et al. [14]	DAG	SHA-1,SHA-256	Time Overhead	User Privacy issues
Hong et al. [15]	×	ECC,Homomorphic	Cipher Text Size	Lack of Accountability

TPM by reducing overhead in both the audit and data upload phases.

Hong et al. [15] suggested a dependable homomorphic encryption scheme based on Elliptic Curve Cryptography (ECC) to ensure the privacy of users. Homomorphic encryption is commonly employed to store and process encrypted outsourced data in cloud computing environments, effectively protecting user privacy. The proposed solution involves managing secure multiparty computation using ECC, a departure from the traditional reliance on a trusted third party. This shift is motivated by the challenges posed by the unavailability and computational complexity associated with the initial approach. However, it's worth noting that the author's work falls short in addressing the issue of accountability. A comparison of Existing solutions and their limitations are shown in Table I.

III. PROPOSED FRAMEWORK

In this section, we explore our proposed Cloud Service architecture Fig. 5, built upon the Hashgraph distributed ledger, incorporating user authentication protocol, intending to manage Service Level Agreement (SLA) violations. Additionally, we leverage Homomorphic Encryption, AES, and the Elliptic Curve Diffie-Hellman key exchange protocol to enhance the security of user data and protect it from unauthorized access. In our proposed framework, we introduce three smart contracts: Subscription, User, and Chronicler, all deployed on the Hedera Hashgraph. Additionally, our architecture incorporates an Encryption Module and a Decryption Module. The detailed operations of both the contracts and modules are elaborated below:

A. Subscription Contract (SC)

This contract facilitates a subscription mechanism for the User, granting access to the cloud service network. Initially, the participant establishes the necessary credentials for service access. This involves configuring a Secret Message, which undergoes SHA256 hashing and the resulting hash is stored globally, serving as a vital component in user authentication for all actions. When a user subscribes to the service, the sequential counters at the receiver and user side are set to the current UNIX time. The user's provided secret message is subjected to a SHA256 hash, followed by XOR with the user's side counter current number. After this operation, the number is incremented. At the receiver's end, which also uses the same current number, the received XORed hash undergoes another XOR operation with the number to retrieve the original hash, which is then verified. Following this verification, the receiver's number is also incremented. Additionally, the participant is prompted to provide the MAC addresses of up to five devices (adjustable as needed), thereby restricting service access to these designated devices. After configuring the credentials, the system deploys a user contract that provides functionalities such as file upload, read, update, and deletion, each tied to a specified duration. This enforces a predefined time limit set by the service provider, requiring contract renewal for continued service access. Finally, a Chronicler contract is deployed to meticulously record all events occurring within the service, as implied by its name.

B. User Contract (UC)

This agreement is formulated for interacting with the cloud by invoking any of the four methods: uploading, deleting, updating, and reading files. To trigger a method, users must do so from any of their registered devices. Furthermore, they must submit their secret message to authenticate their identity and verify their user status, to ensure that only authorized users can avail themselves of the service. All the events made by this contract are managed by the following methods:

1) DataQuery() and fileReg(): The DataQuery function accepts a fileID and the desired action to be executed on the file as inputs. Using the UNIX timestamp, it logs the initiation events of methods (Read, Upload, Delete, Update) with the assistance of the map (map_transactions) and the event (event_transaction) data structure. Similarly, in the fileReg function, logs are associated with the completion of an action, and the serverFileID (the file address generated by the cloud) is concurrently mapped using the map (fileLedger) and event (efileLedger) mechanisms. Both the fileReg() and DataQuery() functions are invoked privately, with one at initialization and the other upon successful completion of an operation. Marking each event will assist in pinpointing the point of failure in the event of any fault occurrence, enabling us to make informed decisions for subsequent actions. Events that trigger the DataQuery() and fileReg() are as follows:
    a) UploadInit: This method records the occurrence of an upload action being initiated.
    b) UploadComplete: This method documents the completion of the upload action.

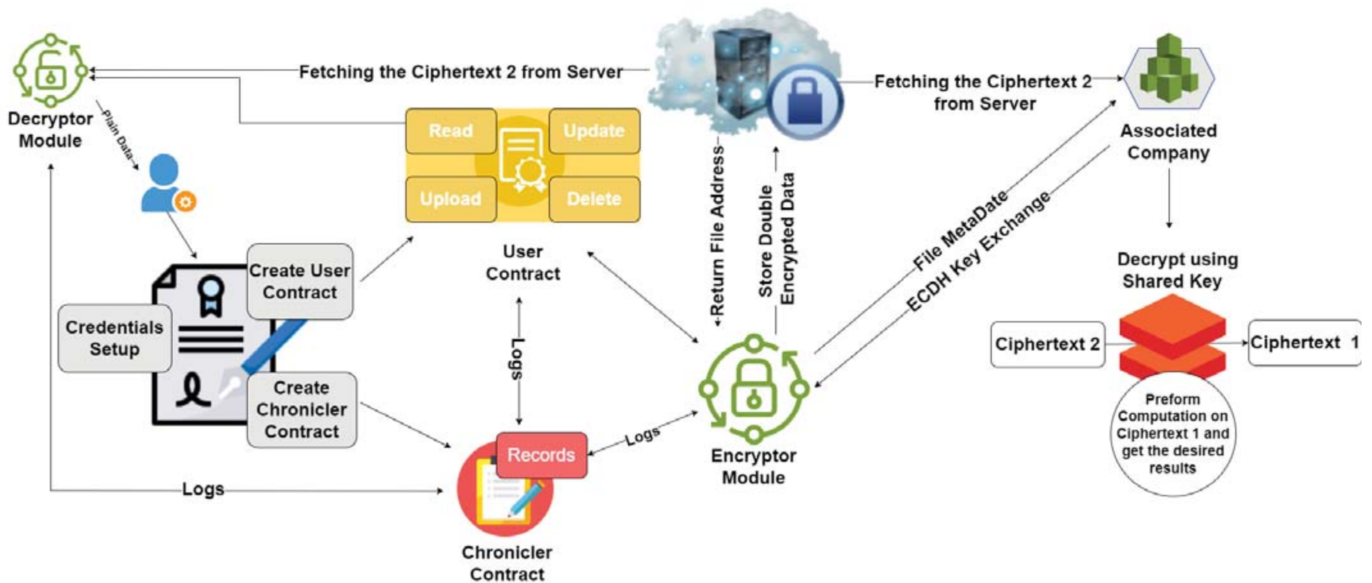

Fig. 5. Proposed Architecture

c) UpdateInit: This method records the occurrence of an update action being initiated.
d) UpdateComplete: This method documents the completion of the update action.
e) ReadInit: This method records the occurrence of a read action being initiated.
f) ReadComplete: This method documents the completion of the read action.
g) DeleteInit: This method records the occurrence of a delete action being initiated.
h) DeleteComplete: This method documents the completion of the delete action.

2) Chronicler Contract: Analogous to the User Contract, the Chronicler Contract stores all records of events occurring outside the User Contract (UC). It incorporates a function called CloudQuery, which requires parameters such as user address, fileID, action, and serverFileID. Using this metadata, the method generates a log associated with the UNIX timestamp, providing detailed information about the event. Events that invoke CloudQuery are as follows:

a) DecryptModuleInit(): This method is triggered simultaneously when the Decryption Module is invoked, marking the corresponding event.
b) DecryptModuleComplete(): This method is invoked upon the completion of the Decryption Module to signify the conclusion of the event.
c) DecryptHomoFileInit(): This method is triggered simultaneously when the Homomorphic decryption of the ciphertext 1 to plain data is initiated, marking the corresponding event.
d) DecryptHomoFileComplete(): This method is invoked upon the completion of the Homomorphic decryption to signify the conclusion of the event.
e) AESDecryptInit(): This method is triggered simultaneously when the AES decryption of ciphertext 2 to ciphertext 1 is initiated, marking the corresponding event.
f) AESDecryptComplete(): This method is invoked upon the completion of the AES decryption to signify the conclusion of the event.
g) EncryptModuleInit(): This method is triggered simultaneously when the Encryption Module is invoked, marking the corresponding event.
h) EncryptModuleComplete(): This method is invoked upon the completion of the Encryption Module to signify the conclusion of the event.
i) EncryptHomoFileInit(): This method is triggered simultaneously when the Homomorphic encryption of the plain data to ciphertext 1 is initiated, marking the corresponding event.
j) EncryptHomoFileComplete(): This method is invoked upon the completion of the Homomorphic encryption to signify the conclusion of the event.
k) AESInit(): This method is triggered simultaneously when the AES encryption of ciphertext 1 to ciphertext 2 is initiated, marking the corresponding event.
l) AESComplete(): This method is invoked upon the completion of the AES encryption to signify the conclusion of the event.
m) CloudUploadInit(): The pre-upload method is invoked prior to transferring the file to the cloud storage, signalling the initiation of the upload process.
n) CloudUploadComplete(): This method associates the fileId provided by the user with the Server-

FileId upon successful file upload to the cloud. Additionally, it triggers the CloudQuery() function to record the transaction log.

o) CloudUpdateInit(): This method invokes CloudQuery() and logs the initiation of this function when an update on the file is performed.

p) CloudUpdateComplete(): This method logs the event upon successful completion of a file update.

q) CloudReadInit(): This method invokes CloudQuery() and logs the initiation of this function when a read operation is performed.

r) CloudReadComplete(): This method is invoked when the file is fetched from the cloud server and registers the corresponding event.

s) CloudDeleteInit(): This method invokes DataQuery() to indicate that the delete operation is encountered, and it logs the initialization of the delete operation for cloud data.

t) CloudDeleteComplete(): This method removes the serverFileId from the mapping and logs the successful deletion from the cloud.

u) FileNotFound(): This method will be invoked when a user attempts to retrieve a file from the cloud server, but the file is not present. This logs the event of the file not being found.

v) The function 'checkFileOnCloud()' can be invoked by the user to verify the presence of a file in the cloud. In cases where a user has uploaded a file but it is not found, the user can initiate SLA verification to determine if the cloud acknowledges the file upload. Penalties for violations will be assessed accordingly. Events recorded during these actions can be traced back to identify instances of SLA violations.

3) Encryption Modules: This module is invoked whenever the user triggers the upload or update method. Initially, the user uploads the file to the module, and it undergoes homomorphic encryption, producing ciphertext 1. Subsequently, ciphertext 1 is processed through AES encryption, resulting in ciphertext 2. The secret key used for AES encryption is generated through the Elliptic Curve Diffie-Hellman key exchange protocol between the associated company and the client. This shared key is then employed by AES to encrypt ciphertext 1 into ciphertext 2. The ciphertext 2 file is then transmitted to the cloud storage from the user's side over an open network. Now for implementation purposes, we have used-

   a) Paillier cryptosystem for implementing Homomorphic Encryption.
   b) BrainpoolP512r1 curve is used for ECDH to generate the secret key.
   c) CTR (Counter) mode is used for AES encryption.

4) Decryption Module: This module is activated whenever the user initiates the read method. Initially, the module retrieves the ciphertext 2 file from the cloud server. Subsequently, ciphertext 2 is decrypted to ciphertext 1 using the shared AES key. Following this, ciphertext 1 undergoes decryption to plain data through homomorphic decryption, thereby providing the user with the original data.

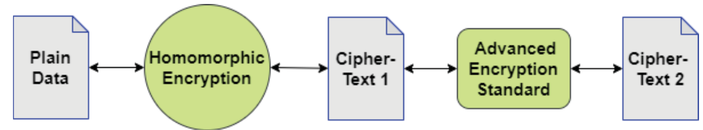

Fig. 6. Encryptor/Decryptor Module

5) Associated Company: This refers to the entity associated with users, requiring access to analyze user data. The company has direct access to the server's data, as file metadata is shared with them. Upon obtaining the doubly encrypted data, referred to as ciphertext 2, the company can utilize a shared key to decrypt it into ciphertext 1, which is homomorphically encrypted data. Subsequently, the company can conduct the necessary computations on ciphertext 1, obtaining results without compromising user privacy.

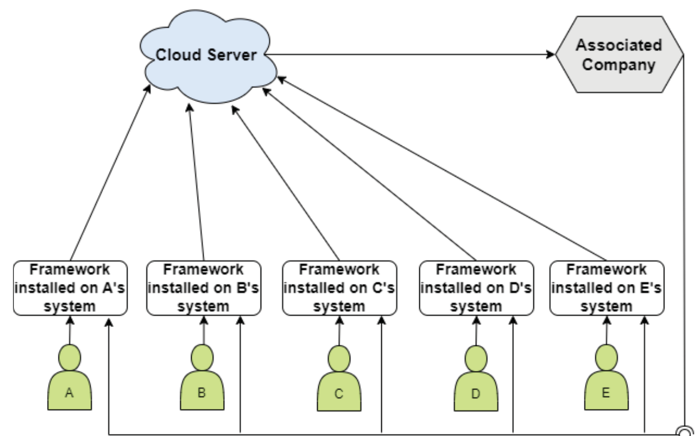

Fig. 7. Interactions of Multiple Users

The process is outlined in Fig. 8. When a user starts the upload function, the first step is to submit initialization logs, as explained in the smart contract. After that, the file undergoes encryption using the encryption module. Once the encryption is complete, the file is then uploaded to cloud storage. For retrieving a file, when a user triggers the read operation, the file is fetched from cloud storage and decrypted using the decryption module. In the upload method, the new file replaces the old file with the same serverFileId. In the case of the Delete operation, the file is deleted from cloud storage. The chronicler contract carefully monitors all the events related to these operations, and logs are generated through the functions detailed in the chronicler contract.

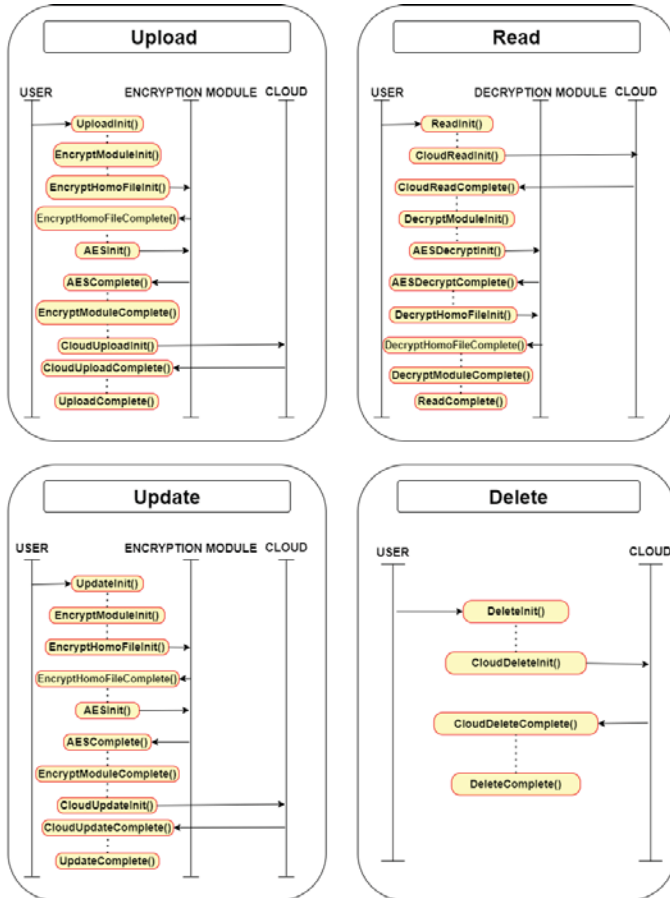

Fig. 8. Sequence diagram depicting the interactions involved in our framework.

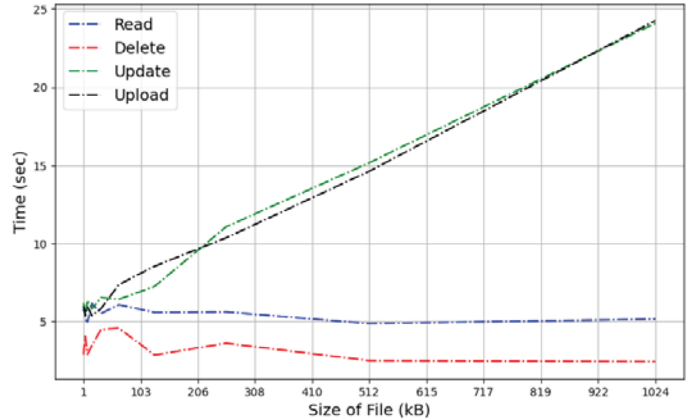

Fig. 9. Average Execution Time for Contract and Cloud Service

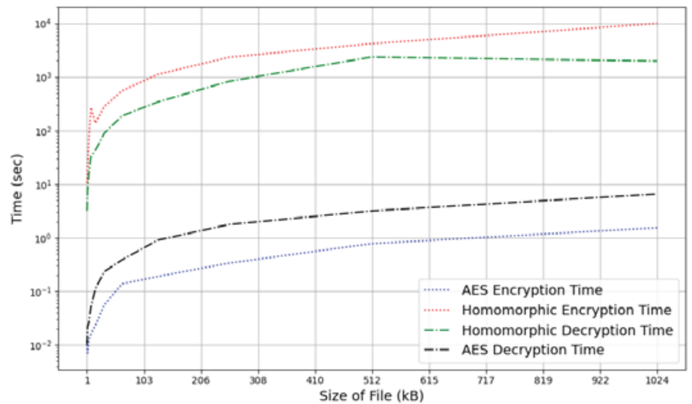

Fig. 10. Average Time for Encryption and Decryption

IV. Performance Analysis

In this section, we analyze our proposed solution in terms of latency, CPU usage, and memory used. We performed all operations within a scenario where a user communicates with cloud services to perform actions on a file. All communication is facilitated through a smart contract deployed on the Hedera network. We simulated our model on a laptop with the device name HP Pavilion, featuring 16 GB DDR4 RAM 3200Mhz, a 12th Gen Intel(R) Core(TM) i7-1260P 2.10 GHz Processor, and running Windows 11 Home operating system. The performance of our model relies on cryptographic schemes, smart contracts, and the time taken to access cloud storage. For our framework, to establish a cloud storage service environment, we utilized the Google Drive API v3 through Google's OAuth 2.0 Playground for developers. Our simulation is executed against varying file sizes, capped at 1 MB for comprehensive analysis. We've encompassed all scenarios where a user can upload, update, read, and delete files from and to the cloud, in addition to encryption and decryption of the files. For each operation, we computed the average data by executing these operations 100 times.

Latency measure of Smart Contract, Cloud Service, Encryptor, and Decryptor: As depicted in Fig. 9 and Fig. 10, we observed that as the file size increased, the latency also gradually increased, with uploads taking longer to execute compared to other operations. The delete operation exhibited the shortest execution time among all operations. The time taken to encrypt and decrypt the file using homomorphic encryption followed by an advanced encryption standard algorithm is also computed by passing the file to the encryption/decryption module. As shown in Fig. 10, the execution time for encryption and decryption using the homomorphic algorithm is higher, whereas the AES algorithm requires less time.

Average Memory Usage of Smart Contract, Cloud Service, Encryption, and Decryption: As shown in Fig. 11 and Fig. 12, We can analyze a direct relationship between memory consumption and file size. Initially, the memory consumption of both the cloud service and Contracts exhibited some randomness, but as the file size increased, the memory consumption became steady and increased linearly. For both the encryption and decryption processes, their memory usage demonstrated a logarithmic increase with the file size. Additionally, the average memory consumption of AES encryption and decryption is comparatively higher than that of Homomorphic encryption and decryption.

Average CPU Usage of Smart Contract, Cloud Service,

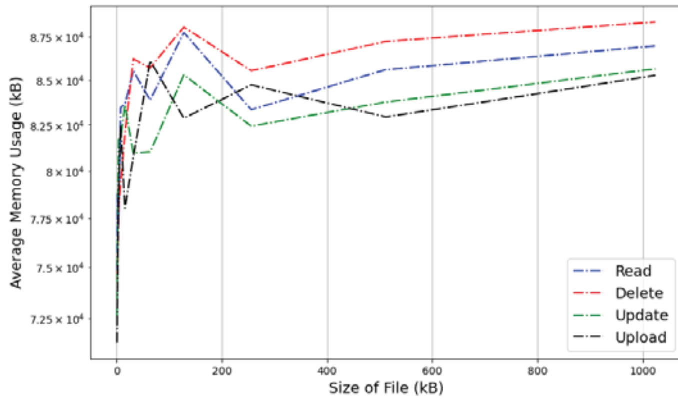

Fig. 11. Average Memory Usage of Contract and Cloud Service

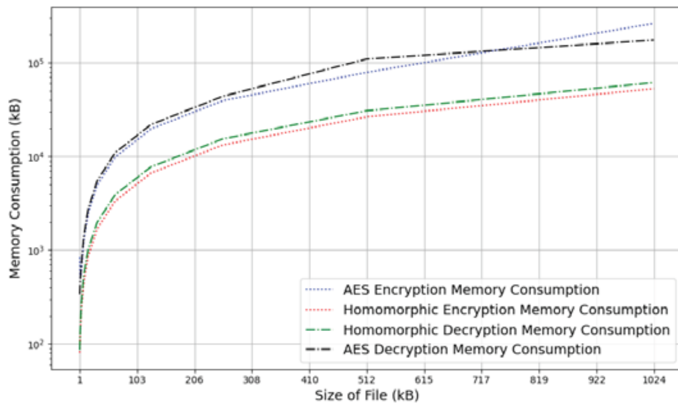

Fig. 12. Average Memory Consumption of Encryption and Decryption

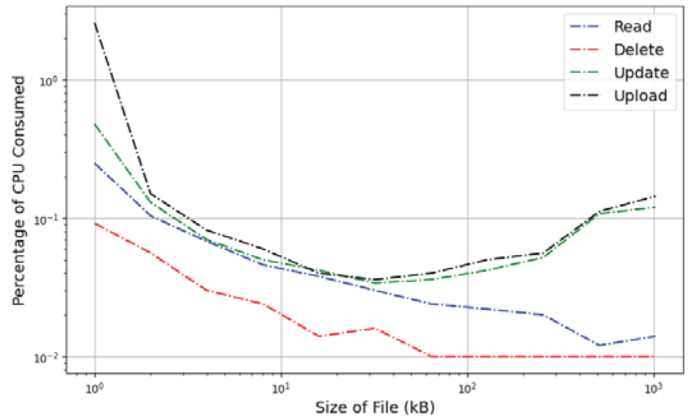

Fig. 13. Average CPU Usage of Contract and Cloud Service

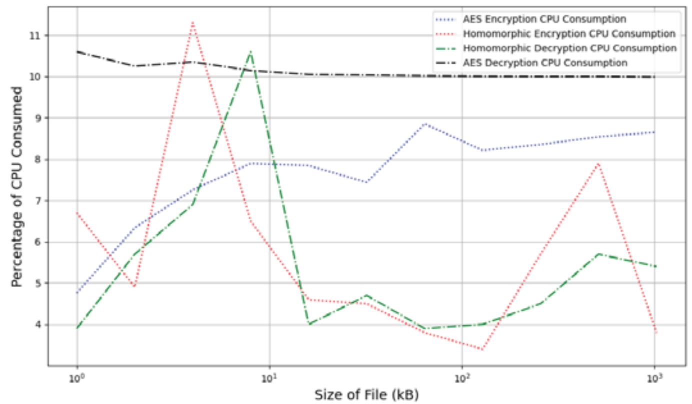

Fig. 14. Average CPU Usage of Encryption and Decryption

Encryption, and Decryption: As shown in Fig. 13 and Fig. 14, We can observe that there is no direct correlation between CPU Usage and file size. Increased CPU usage is noted during uploads, taking longer to execute compared to other operations. Update and Upload exhibit higher usage compared to Read and Delete, as the workload is more intensive for the former due to file uploads on the cloud. In the case of Homomorphic encryption and decryption, their CPU usage is not fixed, indicating independence from the file size. For AES encryption and decryption, the CPU usage of decryption is significantly higher than that of encryption. The file size does not have a significant impact on the CPU usage in this context.

V. Security Analysis

In this section, we traditionally outline our framework's security features, emphasizing its resistance to attacks, privacy protection, and data security. The comparison of the proposed mechanism is presented in Table II.

- Privacy and Data Protection: Addressing user concerns about privacy in cloud services, we have implemented a dual-layer encryption approach. Initially, user data undergoes homomorphic encryption, followed by AES encryption. The shared secret key for AES is established through Elliptic Curve Diffie Hellman (ECDH) between the user and the associated company. Notably, all these cryptographic processes take place at the user's end, eliminating the involvement of intermediaries. Consequently, the data sent to the server from the user side is already doubly encrypted, providing robust protection against intruders. In scenarios where the company needs to perform computations on user data, it can securely retrieve the data from the server, employ the shared key to decrypt the doubly encrypted data into the homomorphic form, and subsequently perform the required computations. This methodology ensures the preservation of user privacy.
- Accountability: Accountability pertains to the reliability and detection of faults within a cloud service. Our resilient hashgraph based contract framework ensures meticulous logging of every event across each service, rendering the proposed solution accountable for any impropriety.
- Resistance to Replay Attack: In a replay attack, the adversary attempts to intercept and replay the access request sent by the user to gain unauthorized access to the data. Our framework addresses this threat by implementing a sequential number generator at both ends. The initial number is set to the UNIX time when the user

TABLE II
COMPARISON OF THE PROPOSED SOLUTION WITH EXISTING ONES

Safety Feature	[1]	[10]	[11]	[12]	[13]	[14]	[15]	Our
Privacy and Data Protection	Y	Y	Y	Y	Y	Y	Y	Y
Accountability	Y	Y	Y	Y	Y	Y	N	Y
Resistance to Replay Attack	N	Y	N	N	N	N	N	Y
Resistance to Known-Plaintext Attack	N	Y	Y	N	N	N	Y	Y
Resistance to Eavesdropping Attack	N	Y	Y	N	N	N	Y	Y
Resistance to Password Cracking Attack	N	N	N	N	N	N	N	Y
Resistance to DDoS	N	N	N	Y	Y	Y	N	Y
Resistance to Sybil Attack	N	N	N	N	Y	Y	N	Y
Resistance to Masquerade Attack	N	N	N	N	N	N	N	Y

subscribes to the service. The secret message provided by the user undergoes a SHA256 hash, followed by XOR with the current number. After this operation, the number is incremented. At the receiver's end, which also utilizes the same current number, the received XORed hash undergoes XOR again with the number to obtain the original hash, which is then verified. Following this, the receiver's number is also incremented. This multi-step process ensures that it remains inconsequential even if an attacker gains access to information after a single use. Let's denote:

- m as the secret message provided by the user,
- $H(m)$ as the SHA256 hash of the message m,
- n_i as the sequential number at the sender's end at time i,
- n_j as the sequential number at the receiver's end at time j.

At the sender's end:

1) Calculate the XORed hash:
$$\text{XOR}_s = H(m) \oplus n_i$$
2) Increment the sender's number:
$$n_{i+1} = n_i + 1$$

At the receiver's end:

1) Obtain the original hash:
$$H'(m) = \text{XOR}_s \oplus n_j$$
2) Verify the hash:
$$H(m) = H'(m)$$
3) Increment the receiver's number:
$$n_{j+1} = n_j + 1$$

This process ensures that even if an attacker intercepts the XORed hash (XOR_s), they cannot obtain the original hash without knowing the current sequential number (n_i). Additionally, each time the sender sends a message, the number is incremented, making it inconsequential for an attacker even if they gain access to information after a single use.

- Resistance to Known-Plaintext Attack: Known-Plaintext Attack is a cryptographic weakness in which the attacker has access to both the encrypted ciphertext and the original plaintext. In our suggested system, we protect against this type of attack by using a multi-layer encryption technique. Specifically, the attacker may access ciphertext 2, which is the result of AES encryption of ciphertext 1. However, the encryption procedure used to obtain ciphertext 1 is hidden from the attacker, making it impossible for them to use this knowledge to compromise the encryption keys.
- Resistance to Eavesdropping Attack: This cybersecurity threat involves an unauthorised third party covertly intercepting communication channels between two parties, and gaining access to the transmitted data. To counter this, we've implemented double encryption on data, utilizing the ECDH method for generating shared AES keys. Even if an attacker intercepts the data over the network, it remains useless as we apply AES encryption with a robust 256-bit key. In the unlikely event that an attacker intercepts the connection between the user and the company, obtaining the shared key involves solving the Elliptic Curve Discrete Logarithm Problem. After this, the attacker would still be left with homomorphically encrypted data for which they lack the necessary keys. In essence, the eavesdropping attack would have a negligible impact
- Resistance to Password Cracking Attack: In the event of an attacker attempting to crack the password through various techniques, our security measures render such efforts futile. Even if the attacker discovers the secret message set by the user, it holds no significance. Access to the service is stringent, requiring authentication from registered devices with specific MAC addresses.
- Resistance to DDoS and Sybil Attack: A Sybil attack is a security threat in which a single adversary has control over numerous nodes or entities on the network. while A Distributed Denial of Service (DDoS) attack is a malicious attempt to disrupt a network, service, or website by flooding it with internet traffic. It uses multiple sources to generate a massive amount of traffic, making mitigation more challenging. The objective is to overwhelm the target's resources such as bandwidth, server capacity, or network components rendering it inaccessible to users. Attackers often deploy botnets (networks of compromised

computers) for coordinated DDoS attacks.

To counter these attacks, we used Hedera Hashgraph, which uses an asynchronous Byzantine Fault Tolerant (aBFT) consensus algorithm to provide an unprecedented level of security against a variety of attacks. In the face of DDoS threats, Hedera employs a fair access mechanism in which users must acquire or stake tokens, reducing the risk of overwhelming the network with malicious transactions. Furthermore, the unique Hashgraph consensus algorithm ensures that transactions are finalized quickly and efficiently, making it resistant to DDoS attacks. To counter Sybil's attacks, Hedera's permissioned network requires nodes to verify their identities, preventing malicious actors from creating a large number of pseudonymous nodes. The economic model, which includes native cryptocurrency (HBAR) staking, further discourages the creation of Sybil nodes by making it economically unfeasible. The governance model, with a council of reputable organizations, adds an extra layer of protection.

- Resistance to Masquerade Attack: A masquerade attack occurs when an attacker pretends to be someone else to gain unauthorized access to systems or data. In our approach, during the subscription process, we request the MAC address of the device that will access the service. A secret message is also set, known only to the user. These credentials are verified at every action, such as upload or reading. For an attacker to cause harm, they would need access to one of the registered devices, knowledge of the secret message, and access to the account linked to the contracts. The convergence of these requirements makes any malicious activity highly improbable.

VI. Conclusion

In our paper, we introduce a comprehensive framework designed to safeguard user data during interactions with cloud services and address SLA violations. Our solution relies on Hashgraph for creating immutable logs and incorporates AES and Homomorphic Encryption to ensure the security of user data. By combining AES with the homomorphic algorithm, we enhance the system's security significantly. Additionally, Hashgraph utilizes a gossip protocol, which reduces computational overhead. An advantage of this approach is that the company associated with the system can perform computations on data post-AES decryption without requiring the user's private key. Our proposed solution is robust against threats, providing a high level of security and efficiency. Furthermore, we assess our framework's performance in terms of latency, CPU usage, and memory consumption.

References

[1] M. Zichichi, G. D'angelo, S. Ferretti, and M. Marzolla, "Accountable clouds through blockchain," *IEEE Access*, 2023.

[2] M. Ogburn, C. Turner, and P. Dahal, "Homomorphic encryption," *Procedia Computer Science*, vol. 20, pp. 502–509, 2013.
clouds through blockchain," *IEEE Access*, 2023.

[3] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2011, pp. 129–148.

[4] A. M. Abdullah *et al.*, "Advanced encryption standard (aes) algorithm to encrypt and decrypt data," *Cryptography and Network Security*, vol. 16, no. 1, p. 11, 2017.

[5] N. Su, Y. Zhang, and M. Li, "Research on data encryption standard based on aes algorithm in internet of things environment," in *2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*. IEEE, 2019, pp. 2071–2075.

[6] X. Zhang and K. K. Parhi, "Implementation approaches for the advanced encryption standard algorithm," *IEEE Circuits and systems Magazine*, vol. 2, no. 4, pp. 24–46, 2002.

[7] D. Brown, "Standards for efficient cryptography, sec 1: elliptic curve cryptography," *Released Standard Version*, vol. 1, 2009.

[8] R. Haakegaard and J. Lang, "The elliptic curve diffie-hellman (ecdh)," *Online at https://koclab. cs. ucsb. edu/teaching/ecc/project/2015Projects/Haakegaard+ Lang. pdf*, 2015.

[9] S. Aikins-Bekoe and J. B. Hayfron-Acquah, "Elliptic curve diffie-hellman (ecdh) analogy for secured wireless sensor networks," *International Journal of Computer Applications*, vol. 176, no. 10, pp. 1–8, 2020.

[10] A. Saini, Q. Zhu, N. Singh, Y. Xiang, L. Gao, and Y. Zhang, "A smart-contract-based access control framework for cloud smart healthcare system," *IEEE Internet of Things Journal*, vol. 8, no. 7, pp. 5914–5925, 2020.

[11] B. Sowmiya, E. Poovammal, K. Ramana, S. Singh, and B. Yoon, "Linear elliptical curve digital signature (lecds) with blockchain approach for enhanced security on cloud server," *IEEE Access*, vol. 9, pp. 138 245–138 253, 2021.

[12] H. Javed, Z. Abaid, S. Akbar, K. Ullah, A. Ahmad, A. Saeed, H. Ali, Y. Y. Ghadi, T. J. Alahmadi, H. K. Alkahtani *et al.*, "Blockchain-based logging to defeat malicious insiders: The case of remote health monitoring systems," *IEEE Access*, 2023.

[13] P. Verma, V. Tripathi, and B. Pant, "Secure hashgraph for healthcare: Strengthening privacy and data security in patient records," *IEEE Transactions on Consumer Electronics*, 2024.

[14] J. Tian and X. Jing, "A lightweight secure auditing scheme for shared data in cloud storage," *IEEE Access*, vol. 7, pp. 68 071–68 082, 2019.

[15] M.-q. Hong, P.-Y. Wang, and W.-B. Zhao, "Homomorphic encryption scheme based on elliptic curve cryptography for privacy protection of cloud computing," in *2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)*. IEEE, 2016, pp. 152–157.