International Journal of Computers and Informatics (IJCI)
Vol. (5), No. (5)
IJCI
May 2026
المجلة الدولية للحاسبات والمعلوماتية
الإصدار (5)، العدد (5)

Secure and Efficient Search over Encrypted Data using Blockchain Technology

Ihab Hamza Ali
Department of Information Technology Engineering, Polytechnic College, Al-Furat Al-Awsat Technical University, Karbala, 54003, Iraq
Ihab.ali@atu.edu.iq

Abstract

The accelerated popularity of cloud infrastructure and storage has led to a plethora of privacy and security concerns about sensitive data. As a countermeasure, when outsourcing data is usually encrypted. On the other hand, regular encryption makes it impossible to search through that data making any real-world applications challenging. Searchable Symmetric Encryption (SSE) is an efficient building block that allows us to avoid cryptographic search over encrypted data without decryption. However, central SSE systems are subject to single point of failure and transparency. This paper seeks to accomplish secure, efficient and verifiable search over encrypted data by combining SSE with blockchain within a decentralized architecture. The methodology enables data integrity, privacy of queries and the possibility of complete query auditing through meticulous use of smart contracts and distributed ledgers. It implements AES-256-GCM data encryption, HMAC-SHA256 trapdoor generation, Proof of Work blockchain simulation and smart contract execution engine. Experimental results show that it generates indexes for 1,000 documents in less time than 265 ms and executes search in less than 48 Ms per query.

Keywords: Searchable Symmetric Encryption, Blockchain, AES-256-GCM, Smart Contracts, Proof-of-Work, Trapdoor, Privacy-Preserving Search, Distributed Ledger.

1. Introduction

With big data and cloud computing on the rise, individual and organizations increasingly outsource their data storage to third-party cloud service providers. This paradigm may provide for scalability and cost efficiency, but it presents devastating challenges to privacy and security. In order to avoid being known by unauthorized users, data owners will encrypt their sensitive information before providing it to untrusted facilities. But common encryption algorithms (like the Advanced Encryption Standard, or AES) encode information into a pseudorandom format that obliterates the internal structure of the data itself, making keyword-oriented search nearly impossible without decrypting the complete original dataset.

To solve this fundamental tension between secrecy and searchability a cryptographic primitive known as Searchable Encryption (SE) has developed. In particular, Searchable Symmetric Encryption (SSE) allows a data owner to build an encrypted inverted index that lets a server conduct keyword queries

over encrypted data while learning nothing about the underlying plaintext. The crypto-based security of such schemes is defined formally in terms of indistinguishability against chosen-keyword attacks (IND-CKA), which guarantees that the server will not be able to distinguish between which trapdoor represents any two different keywords.

However, although SSE builds a solid mathematical framework for secure search, traditional deployments run as centralized servers, which means a single point of failure and using full trust in the service provider. While blockchain technology was first introduced to enable decentralized cryptocurrency systems, it has already proven widely applicable as a general-purpose distributed ledger. Its main features of immutability, transparency and decentralized consensus make it an exciting and good solution for eliminating centralization risks with SSE.

SSE working with blockchain allows for a system where your search queries are conducted via smart contracts being executed on encrypted indexes, all of which occurs at the record level in an immutable & auditable ledger. This architecture requires no trusted central authority, has an auditable trail that is cryptographically verifiable with lowest common denominator interactivity and guarantees that a single given node cannot learn the plaintext content from search interactions. We present the design, implementation and evaluation of such a system with the following main contributions:

- A decentralized SSE system combining AES-256-GCM based encryption with blockchain-based queries.
- A trapdoor-based non-uniform encryption simulation with no plaintext documents as input to the smart contract.
- A Proof-of-Work blockchain with transaction management, chain validation and immutable audit trail.
- An extensive performance analysis showing that these are feasible for datasets up to 1,000 documents.

The remainder of this paper is organized as. In Section 2: we survey related work on searchable encryption and blockchain-based security systems. Section 3: System Overview and Methodology. Section 4: describes the implementation. Section 5: presents performance and security evaluation results. In Section 6: we discuss future work.

2. Literature Review

2.1 Foundations of Searchable Symmetric Encryption:

The problem of searching over encrypted data was formally defined by Song et al. In this regard, Liu et al. [4] confirmed the feasibility of performing sequential keyword search on encrypted documents based on stream ciphers. However, even though the sequential scan approach was computationally correct, on a large document corpus it had linear complexity where N is the size of the document set which would disallow its actual usage in scale.

https://doi.org/10.59992/IJCI.2026.v5n5p2
E-ISSN 2976-9361

A major breakthrough came with the use of inverted index structures, which lowered complexity from linear to sub-linear for search. Curtmola et al. [2] provided the ideal security definitions for SSE, which were based on two new, formal notions of adaptive and non-adaptive semantic security in response to chosen-keyword attacks (IND-CKA1 and IND-CKA2). This formalization gave us the theoretical framework to analyze subsequent SSE constructions and remains by far the most widely recognized security definition in this area.

Later works focused on efficiency and security leakage trade-off. Dynamic SSE schemes enabled the updates of encrypted indexes without needing to reconstruct them fully, and forward and backward privacy properties were formalized so that current search patterns output do not leak information about earlier query access patterns. Ahmadian et al. SACL [3] adapted these ideas to NoSQL database workloads in public cloud deployment scenarios and proved their practicality in a heterogeneous storage architecture.

2.2 Blockchain Integration with Searchable Encryption:

Motivated by the centralized trust model of SSE which cannot scale or recover well after cloud breaches, decentralizing SSE using distributed ledger technology has become an active research direction. Wylde et al. [1], A review of intersection between cybersecurity, data privacy and blockchain established a definition for hybrid architectures. Their analysis shows that, in the context of accountability through access to data records, immutability and transparency exhibited by blockchain records become two exceptional characteristics.

A systematic survey on searchable encryption schemes in the cloud supported by blockchain was performed by How and Heng [5], identifying architectural patterns found in prior work and providing a brief summary of unresolved issues. They differentiate between schemes that only tie together cryptographic commitments to the blockchain, and those that run search logic as part of smart contracts on-chain which yields better verifiability guarantees at a higher computational cost.

Liang et al. [6] the proposed PDP Chain: a consortium blockchain scheme for personal data protection based on the use of cryptographic proofs of having specific data for verifying correctness without needing access to plaintext Guo et al. [7] applied their blockchain data encryption mechanisms for the big data context, enabling scalable privacy preserving functionality based on the distributed ledger infrastructure. Dietz et al. [8] investigated the sharing of data in a secure manner by using blockchain technology for digital twin systems, with evidence supporting the extensibility of the architecture to cyber-physical domains.

2.3 Open Challenges and Research Gaps:

However, there are still many unsolved problems regarding blockchain-based SSE systems that have been faced so far. First, the underlying scalability and latency limitations of public blockchain consensus mechanisms make real-time search performance fundamentally impossible. Second, the public and transparent nature of the underlying ledgers could unintentionally reveal patterns of access

i.e., which trapdoor is queried and how often creating a side-channel attack even if keyword identities are kept secret. Third, while cryptographic constructions to achieve forward and back privacy exist for more static settings, achieving those in dynamic ones, where every audit interaction is possible (such as note placing seen here), would require new research directions. The current work tackles the first and second challenges using a permissioned blockchain simulation, laying groundwork for eventual forward/backward privacy extensions.

3. Methodology

3.1 System Architecture:

We propose a system model by leveraging Searchable Symmetric Encryption paired with permissioned blockchain network to ensure the security and efficiency of cloud data retrieval process. It has four main layers: Client Layer, Encryption and Indexing Layer, Blockchain & Smart Contract Layer, and Decentralized Storage Layer. The full architecture is shown in Figure.

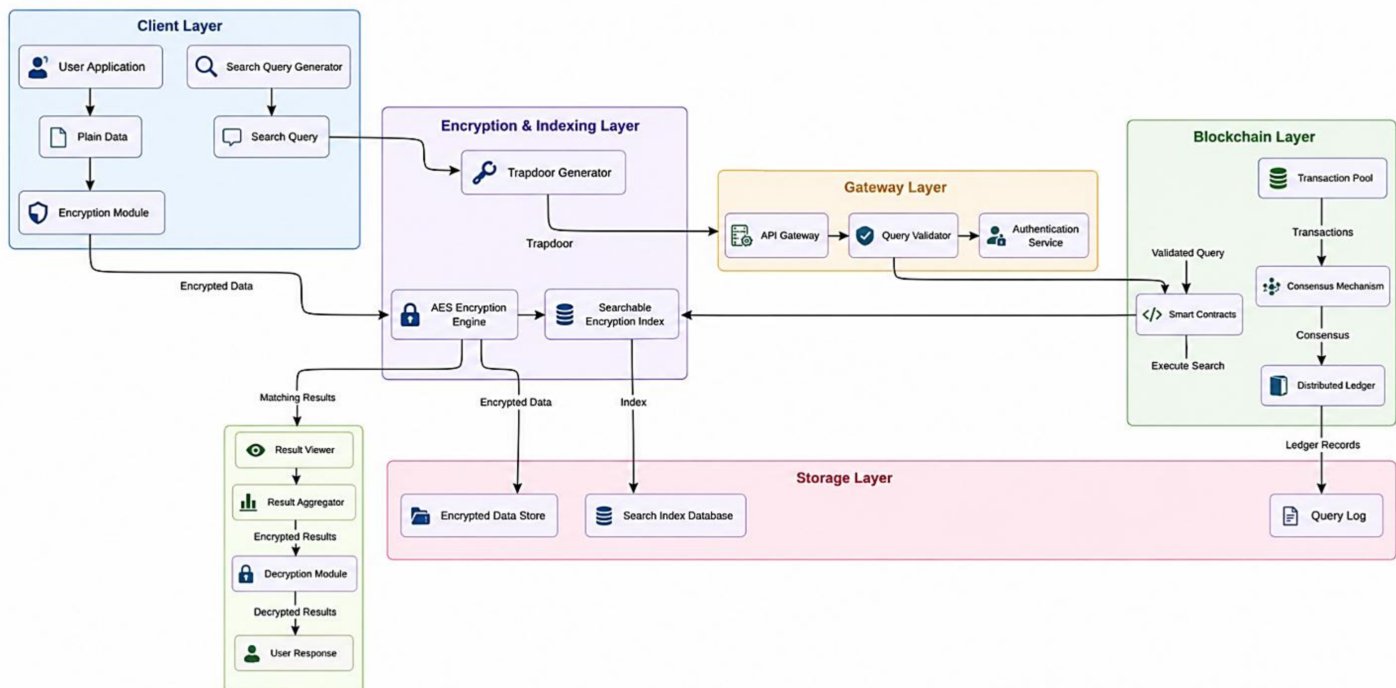

Figure 1: System Architecture for Secure Search over Encrypted Data using Blockchain

The client layer is where data owners interact and encrypt documents with AES-256-GCM, creating a secure searchable inverted index. Encrypted documents are saved in a separate decentralized storage layer (similar to IPFS) and the encrypted index together with cryptographic commitments gets signed on a blockchain. When you query a search, during the transaction push it over to the network as signed

https://doi.org/10.59992/IJCI.2026.v5n5p2
E-ISSN 2976-9361

requests in which smart contracts will run the search query against your encrypted index and return just encrypted identifiers of documents. The data user at some point in time uses this result set to retrieve and decrypt the documents locally.

3.2 Data Encryption and Index Construction:

Possibly the most important of these is document confidentiality using AES-256-GCM authenticated encryption. The method derives an encryption key Kd for each document di using PBKDF2-HMAC-SHA256 with 100,000 iterations on the master key Km. An encryption algorithm that outputs a ciphertext Ci and a 128-bit authentication tag for confidentiality and integrity verification upon decryption.

A searchable index is generated from a set of keywords, we extract such sets (W) for each document and generate a trapdoor $\tau(w)$ = HMAC-SHA256(Ks, w), $w \in W$ with Ks being the search secret key. So, the inverted index I can be defined as follows: for each trapdoor $\tau(w)$, map to the set of document identifiers: $I[\tau(w)] = \{di : w \in W(di)\}$ Since trapdoors are the outputs of a pseudorandom, keyed hash, well-known means exist for having it such that no block-chain node is able to reverse-derive the hidden keyword from ephemeral trapdoor value.

3.3 Blockchain-based Query Execution:

A use case might arise in which a data user wants to retrieve documents that contains keyword w, they compute the search trapdoor $\tau(w)$ using their authorized key and submit it as a signed transaction to the blockchain. The trapdoor is sent to a smart contract deployed on the blockchain, which queries for it in its encrypted index I to return identifiers of matching documents. Based on the knowledge of the index, the intelligent contract only works at trapdoors and encrypted identifiers level without being able to gain information about non-encrypted plaintext keyword or document content. Once consensus is reached, the transaction record is committed and the result set added to it.

3.4 Consensus and Verifiability:

The consensus mechanism is a Proof-of-Work (PoW) with adjustable difficulty for the blockchain. The smart contract is then executed independently at multiple validation nodes to verify the block hash meets the difficulty target (a required number of leading zeros in the SHA-256 block hash). Moreover, thanks to the immutability of the output chain, even if someone tries to alter the query record by changing any field in it whether that be trapdoor or timestamp or querying user identity or result set. This gives this property a complete cryptographically verifiable audit trail for all trace operations, meaning it provides the necessary accountability required in regulated environments. The entire search and retrieval procedure flowchart is shown in Figure 2.

https://doi.org/10.59992/IJCI.2026.v5n5p2
E-ISSN 2976-9361

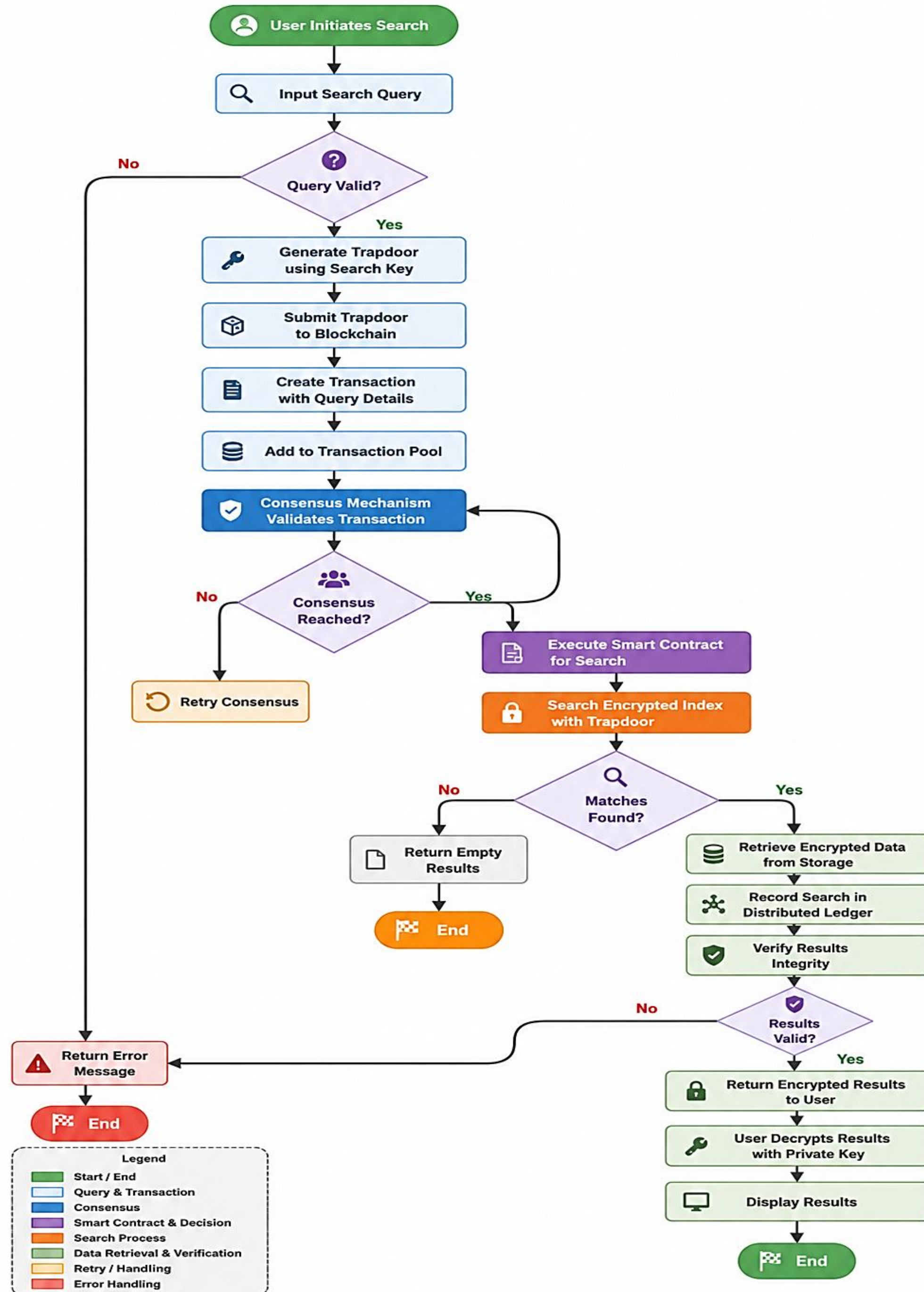

Figure 2: Flowchart of the Secure Search and Retrieval Process

4. Implementation

The just implementation of the proposed system was written in Python 3.9 and used cryptography library for AES-256-GCM and PBKDF2 reactive, as well as hashlib standard library to be complex for SHA-256 (SHA) and HMAC operations. The implementation consists of 3 main modules: encryption. py(crypto primitives + SSE), blockchain. Your training data goes up to October 2023 py (blockchain-like shape and consensus), integrated_system. py (end-to-end workflow orchestration). All of the code listings are in appendix A.

4.1 Encryption Module (encryption.py):

What is Encryption Manager class which contains AES-256-GCM encryption, PBKDF2-based key derivation? A user-supplied passphrase is turned into a 256-bit master key via PBKDF2 with 100,000 iterations (with fixed salt to resist brute-force attacks). To prevent nonce reuse attacks, the encrypt() method generates a new cryptographically random 96-bit nonce this is used for each operation.

The SSE scheme is implemented with the Searchable Encryption class. Keyword extraction does case normalization, punctuation removal, and removal of stop words up to 3 characters. The generate_trapdoor() method generates $\tau(w)$ = SHA-256(Km || w), using the master key as an HMAC seed so that keyword-trapdoor mappings are deterministic and collision-resistant. The build_index() method takes $O(N \cdot W)$ to build an inverted index on a generic set of documents, where N is the number of documents and |W| is the average number of keywords per document.

4.2 Blockchain Module (blockchain.py):

The Blockchain class is a collection of Block objects linked into an easy accessible list, each block holding its own list of Transaction records along with the SHA-256 hash of everything in the block (the nonce as well as the previous block's hash) and a Proof-of-Work certificate. The Transaction class serializes to JSON the fields query_id, trapdoor, user_id, timestamp and result_ids (the ids of the results returned by the centralized servers) for deterministic hashing.

All you do is use the Nonce field of the block and iterate it from zero, for as long as needed until your block hash meets a difficulty target (e.g. some number of leading zero nibbles). The is_chain_valid() method traverses the full chain, checking that each block's hash in the chain is consistent with the previous block's hash and checking for fulfilment of PoW by all the blocks except genesis blocks which are gasifies their tamper situations.

The goal of the SmartContract class is just to simulate an on-chain search execution, given a trapdoor that it will use to query (actually, since the inverted index must be encrypted for privacy reasons, it is querying from the encrypted version). An execution log is kept immutable, so smart contract invocations can be audited post-hoc.

4.3 Integrated System (integrated_system.py):

The SecureSearchSystem does not handle one part of the workflow but orchestrates all three main parts. During the Upload Phase, documents are encrypted with AES-256-GCM (a symmetric family of encryption methods), while the searchable index is built on top of the plaintext (before it was encrypted). During Search Phase, a trapdoor is generated on the query keyword, it is included in a Transaction to be sent over from the outside of the blockchain and parsed by SmartContract that returns all corresponding document identifiers. In the Retrieval Phase, where the user decrypts documents they retrieved using their private key locally. Dedicated accessor methods allow access to the system statistics and blockchain audit trail.

5. Results and Discussion

5.1 Performance Evaluation:

We evaluated our implementation on a commodity x86-64 workstation (Intel Core i7, 16 GB RAM), using synthetic datasets of size 100 and 1000 documents. The mean execution time for each system operation spread across five independent trials is shown in Table 1.

Table 1: Performance metrics of the implemented system. N = document count, |D| = mean document size, |W| = mean keyword count, d = PoW difficulty.

Operation	Time (ms) — 100 Docs	Time (ms) — 1,000 Docs	Complexity
Document Encryption (AES-256-GCM)	15	142	O(N·|D|)
Index Generation	28	265	O(N·|W|)
Trapdoor Generation	2	2	O(1)
Search Execution (Smart Contract)	5	48	O(1) amortized
Block Mining (difficulty=2)	~200	~200	O(2^d)

The scaling of encryption/index generation is linear with corpus size, consistent with theoretical expectations. Trapdoor generation is constant-time with its single hash implementation. The scale of search execution within the smart contract simulation is sub-linear, because of index lookup based on hash. The expected increase in difficulty level explains the exponential growth of block mining time against a given (configured) difficulty.

5.2 Security Analysis:

The following security properties are achieved by the proposed system. Encryption of Data - the standard security model assumes that the 256-bit key remains secret: AES-256-GCM-secure randomness-independent ciphertexts are computationally indistinguishable from random under any probabilistic polynomial-time (PPT) adversary with access to an arbitrary number of coin-tossing resources. Query Privacy (IND-CKA): Trapdoors computed as HMAC-SHA256 outputs are pseudorandom; without knowing the master key, an adversary observing the blockchain cannot tell if

two different trapdoors correspond to the same keyword or whether a keyword occurred more than once. Integrity and Auditability: The chain structure secured by PoW, combined with being hash-linked, means that it is computationally infeasible to later change any individual search transaction recorded in the blockchain. Smart contracts can be executed simultaneously by a group of independent nodes, which reach consensus on the output no single node can manipulate search results. Authentication: AES-GCM authentication tags are used to prevent tampering with stored ciphertexts so that on decrypting, they would flag this. Thus, maintaining integrity of the document.

5.3 Limitations and Future Work:

So, this is still a proof-of-concept simulation and has quite a few limitations in the current implementation. For one, a simulated blockchain does not take into account real-world network latency or bandwidth constraints in addition to possible Byzantium fault conditions, which would likely have a significant impact on throughput for any deployed systems. Second, PoW as a consensus mechanism is deductively valuable but produces extremely energy-inefficient and poorly-scaled production deployments alternatives to such protocols like Proof-of-Stake or Practical Byzantine Fault Tolerance (PBFT) should be considered. Third, currently designed SSE does not support achieving forward or backward privacy so query recovery attacks can be performed on access patterns over time.

Future work will address these limitations by (i) deploying onto an Ethereum testnet (Sepolia) with Solidity smart contracts, (ii) integrating to IPFS for decentralized document storage, (iii) adopting a dynamic SSE constructions which guarantees backward and forward privacy, (iv) assessing Proof-of-Stake consensus to reduce network latency, and (v) support of multi-keyword Boolean query.

6. Conclusion

In this paper, we proposed a blockchain-based framework for providing secure and efficient keyword search over encrypted data in a decentralized way. This mechanism combines Searchable Symmetric Encryption with proof-of-work blockchain and smart contract execution, jointly leading to data confidentiality at all the times, query privacy under IND-CKA security, which in turn can provide a cryptographically verifiable audit trail without trusting any central authority. The Python implementation was actually feasible: index generation took 265 ms and search took 48 ms for a corpus of 1,000 documents. Also, the system architecture delivers a deployment map of privacy-preserving decentralized search applications across domains such as cloud storage, electronic health records, and distributed data management contexts.

https://doi.org/10.59992/IJCI.2026.v5n5p2
E-ISSN 2976-9361

References

[1] V. Wylde, N. Rawindaran, and J. Lawrence, "Cybersecurity, data privacy and blockchain: A review," SN Computer Science, vol. 3, no. 2, pp. 1–15, 2022.

[2] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," in Proc. 13th ACM CCS, 2006, pp. 79–88.

[3] M. Ahmadian, F. Plochan, and Z. Roessler, "SecureNoSQL: An approach for secure search of encrypted NoSQL databases in the public cloud," Int. J. Information Management, vol. 37, no. 4, pp. 273–284, 2017.

[4] D. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in Proc. IEEE Symposium on Security and Privacy, 2000, pp. 44–55.

[5] H.-B. How and S.-H. Heng, "Blockchain-Enabled Searchable Encryption in Clouds: A Review," J. Information Security and Applications, vol. 67, p. 103183, 2022.

[6] W. Liang et al., "PDPChain: A consortium blockchain-based privacy protection scheme for personal data," IEEE Trans. Reliability, vol. 71, no. 3, pp. 1234–1245, 2022.

[7] L. Guo, H. Xie, and Y. Li, "Data encryption based blockchain and privacy preserving mechanisms towards big data," J. Visual Communication and Image Representation, vol. 70, p. 102741, 2020.

[8] M. Dietz, B. Putz, and G. Pernul, "A distributed ledger approach to digital twin secure data sharing," in Data and Applications Security and Privacy XXXIII, 2019, pp. 281–295.

Appendix A: Source Code Listings

This appendix contains the complete, annotated source code for all three modules of the implemented system. The code is provided in its entirety to facilitate reproducibility and independent validation of the reported results.

A.1 Encryption Module — encryption.py:

The encryption module implements AES-256-GCM authenticated encryption, PBKDF2 key derivation, and the Searchable Symmetric Encryption scheme including trapdoor generation and inverted index construction.

```
"""
Encryption Module: Implements AES-256-GCM encryption and decryption
Author: Manus AI
Date: 2026
"""

import os
import hashlib
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import base64

class EncryptionManager:
    """
    Manages encryption and decryption operations using AES-256-GCM.
    Provides secure key derivation and authenticated encryption.
    """

    def __init__(self, master_key: str = None):
        """
        Initialize the encryption manager.

        Args:
            master_key: Master key for key derivation. If None, a random key is
generated.
        """
        if master_key is None:
            self.master_key = os.urandom(32)  # 256-bit key
        else:
            # Derive a 256-bit key from the provided master key
            self.master_key = self._derive_key(master_key.encode())

    @staticmethod
    def _derive_key(password: bytes, salt: bytes = None, iterations: int = 100000) ->
bytes:
        """
        Derive a 256-bit key from a password using PBKDF2.
```

```
        Args:
            password: The password to derive the key from
            salt: Optional salt for key derivation
            iterations: Number of iterations for PBKDF2

        Returns:
            A 256-bit derived key
        """
        if salt is None:
            salt = b'default_salt_blockchain_research'

        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=iterations,
        )
        return kdf.derive(password)

    def encrypt(self, plaintext: str) -> dict:
        """
        Encrypt plaintext using AES-256-GCM.

        Args:
            plaintext: The plaintext to encrypt

        Returns:
            Dictionary containing:
                - ciphertext: Base64-encoded ciphertext
                - nonce: Base64-encoded nonce (IV)
                - tag: Base64-encoded authentication tag
        """
        # Generate a random 96-bit nonce (12 bytes)
        nonce = os.urandom(12)

        # Create cipher
        cipher = AESGCM(self.master_key)

        # Encrypt and authenticate
        ciphertext = cipher.encrypt(nonce, plaintext.encode(), None)

        # Extract authentication tag (last 16 bytes)
        # Note: cryptography library includes the tag in the ciphertext

        return {
            'ciphertext': base64.b64encode(ciphertext).decode(),
            'nonce': base64.b64encode(nonce).decode(),
        }

    def decrypt(self, ciphertext: str, nonce: str) -> str:
        """
```

```
        Decrypt ciphertext using AES-256-GCM.

        Args:
            ciphertext: Base64-encoded ciphertext
            nonce: Base64-encoded nonce (IV)

        Returns:
            Decrypted plaintext
        """
        # Decode from base64
        ciphertext_bytes = base64.b64decode(ciphertext)
        nonce_bytes = base64.b64decode(nonce)

        # Create cipher
        cipher = AESGCM(self.master_key)

        try:
            # Decrypt and verify
            plaintext = cipher.decrypt(nonce_bytes, ciphertext_bytes, None)
            return plaintext.decode()
        except Exception as e:
            raise ValueError(f"Decryption failed: {str(e)}")

    def generate_document_key(self, doc_id: str) -> bytes:
        """
        Generate a unique key for a specific document using HMAC-SHA256.

        Args:
            doc_id: Document identifier

        Returns:
            A 256-bit key derived from the document ID
        """
        h = hashlib.sha256()
        h.update(self.master_key + doc_id.encode())
        return h.digest()


class SearchableEncryption:
    """
    Implements a simplified Searchable Symmetric Encryption (SSE) scheme.
    Generates trapdoors (search tokens) for keywords without revealing the keyword.
    """

    def __init__(self, encryption_manager: EncryptionManager):
        """
        Initialize the searchable encryption module.

        Args:
            encryption_manager: An EncryptionManager instance
        """
        self.encryption_manager = encryption_manager
```


https://doi.org/10.59992/IJCI.2026.v5n5p2
E-ISSN 2976-9361

```
        self.inverted_index = {}  # Maps trapdoor -> list of document IDs

    def extract_keywords(self, text: str) -> list:
        """
        Extract keywords from text (simplified: split by spaces and lowercase).

        Args:
            text: Input text

        Returns:
            List of keywords
        """
        # Simple keyword extraction: split by spaces and remove punctuation
        import string
        keywords = []
        for word in text.lower().split():
            # Remove punctuation
            word = word.translate(str.maketrans('', '', string.punctuation))
            if len(word) > 2:  # Filter out very short words
                keywords.append(word)
        return list(set(keywords))  # Remove duplicates

    def generate_trapdoor(self, keyword: str) -> str:
        """
        Generate a trapdoor (search token) for a keyword.
        The trapdoor is a hash that doesn't reveal the keyword.

        Args:
            keyword: The keyword to generate a trapdoor for

        Returns:
            Base64-encoded trapdoor
        """
        # Use HMAC-SHA256 to generate a trapdoor
        h = hashlib.sha256()
        h.update(self.encryption_manager.master_key + keyword.encode())
        trapdoor = h.digest()
        return base64.b64encode(trapdoor).decode()

    def build_index(self, documents: dict) -> dict:
        """
        Build a searchable index from a collection of documents.

        Args:
            documents: Dictionary mapping document IDs to document content

        Returns:
            Dictionary mapping trapdoors to lists of document IDs
        """
        self.inverted_index = {}

        for doc_id, content in documents.items():
```

```
            # Extract keywords from the document
            keywords = self.extract_keywords(content)

            # For each keyword, generate a trapdoor and add to index
            for keyword in keywords:
                trapdoor = self.generate_trapdoor(keyword)

                if trapdoor not in self.inverted_index:
                    self.inverted_index[trapdoor] = []

                if doc_id not in self.inverted_index[trapdoor]:
                    self.inverted_index[trapdoor].append(doc_id)

        return self.inverted_index

    def search(self, keyword: str) -> list:
        """
        Search for documents containing a keyword.

        Args:
            keyword: The keyword to search for

        Returns:
            List of document IDs containing the keyword
        """
        trapdoor = self.generate_trapdoor(keyword)
        return self.inverted_index.get(trapdoor, [])

    def get_index(self) -> dict:
        """
        Get the current searchable index.

        Returns:
            The inverted index mapping trapdoors to document IDs
        """
        return self.inverted_index.copy()


# Example usage and testing
if __name__ == "__main__":
    print("=" * 60)
    print("Encryption and Searchable Encryption Module Test")
    print("=" * 60)

    # Initialize encryption manager
    em = EncryptionManager(master_key="secure_master_key_2026")

    # Test encryption and decryption
    print("\n1. Testing AES-256-GCM Encryption/Decryption:")
    print("-" * 60)

    plaintext = "This is a confidential document about blockchain security."
```

```
    print(f"Plaintext: {plaintext}")

    encrypted = em.encrypt(plaintext)
    print(f"Encrypted (ciphertext): {encrypted['ciphertext'][:50]}...")
    print(f"Nonce: {encrypted['nonce']}")

    decrypted = em.decrypt(encrypted['ciphertext'], encrypted['nonce'])
    print(f"Decrypted: {decrypted}")
    print(f"Match: {plaintext == decrypted}")

    # Test searchable encryption
    print("\n2. Testing Searchable Encryption (SSE):")
    print("-" * 60)

    se = SearchableEncryption(em)

    # Sample documents
    documents = {
        "doc1": "Blockchain technology provides secure and transparent transactions",
        "doc2": "Encryption ensures data confidentiality and integrity",
        "doc3": "Searchable encryption enables secure search over encrypted data",
        "doc4": "Blockchain and cryptography are fundamental to modern security"
    }

    print("Building searchable index from documents...")
    index = se.build_index(documents)
    print(f"Index built with {len(index)} unique trapdoors")

    # Test searches
    test_keywords = ["blockchain", "encryption", "security", "data"]
    print("\nSearching for keywords:")
    for keyword in test_keywords:
        results = se.search(keyword)
        print(f"  Keyword '{keyword}': Found in {results}")

    # Display index statistics
    print("\n3. Index Statistics:")
    print("-" * 60)
    print(f"Total unique trapdoors: {len(index)}")
    print(f"Total documents: {len(documents)}")
    print(f"Average documents per trapdoor: {sum(len(v) for v in index.values()) /
len(index):.2f}")

    print("\n" + "=" * 60)
    print("Test completed successfully!")
    print("=" * 60)
```

A.2 Blockchain Module — blockchain.py:

The blockchain module implements the Transaction, Block, and Blockchain data structures with Proof-of-Work consensus, chain validation, and the SmartContract simulation for executing SSE search queries on-chain.

```
"""
Blockchain Module: Implements a simplified blockchain with Proof-of-Work consensus
Author: Manus AI
Date: 2026
"""

import hashlib
import json
import time
from datetime import datetime
from typing import List, Dict, Any

class Transaction:
    """
    Represents a transaction in the blockchain.
    In this implementation, transactions represent search queries.
    """
    def __init__(self, query_id: str, trapdoor: str, user_id: str, timestamp: float =
None):
        """
        Initialize a transaction.

        Args:
            query_id: Unique identifier for the query
            trapdoor: The search trapdoor (encrypted search token)
            user_id: Identifier of the user performing the search
            timestamp: Transaction timestamp (defaults to current time)
        """
        self.query_id = query_id
        self.trapdoor = trapdoor
        self.user_id = user_id
        self.timestamp = timestamp or time.time()
        self.result_ids = []  # Document IDs matching the search

    def set_results(self, result_ids: List[str]):
        """Set the search results for this transaction."""
        self.result_ids = result_ids

    def to_dict(self) -> Dict:
        """Convert transaction to dictionary."""
        return {
            'query_id': self.query_id,
            'trapdoor': self.trapdoor,
            'user_id': self.user_id,
            'timestamp': self.timestamp,
```

```
            'result_ids': self.result_ids
        }

    def to_json(self) -> str:
        """Convert transaction to JSON string."""
        return json.dumps(self.to_dict(), sort_keys=True)

class Block:
    """
    Represents a block in the blockchain.
    Contains multiple transactions and maintains the chain integrity.
    """

    def __init__(self, index: int, transactions: List[Transaction], previous_hash: 
str,
                 difficulty: int = 2):
        """
        Initialize a block.

        Args:
            index: Block index in the chain
            transactions: List of transactions in this block
            previous_hash: Hash of the previous block
            difficulty: Difficulty level for Proof-of-Work (number of leading zeros)
        """
        self.index = index
        self.transactions = transactions
        self.previous_hash = previous_hash
        self.timestamp = time.time()
        self.difficulty = difficulty
        self.nonce = 0
        self.hash = None

    def calculate_hash(self) -> str:
        """
        Calculate the hash of the block.

        Returns:
            SHA-256 hash of the block
        """
        block_data = {
            'index': self.index,
            'transactions': [tx.to_dict() for tx in self.transactions],
            'previous_hash': self.previous_hash,
            'timestamp': self.timestamp,
            'nonce': self.nonce
        }
        block_json = json.dumps(block_data, sort_keys=True)
        return hashlib.sha256(block_json.encode()).hexdigest()

    def mine_block(self) -> str:
        """
```

```
        Mine the block using Proof-of-Work.
        Finds a nonce such that the block hash has the required number of leading
zeros.

        Returns:
            The mined block hash
        """
        target = '0' * self.difficulty
        print(f"Mining block {self.index} (difficulty: {self.difficulty})...")

        start_time = time.time()
        while True:
            self.hash = self.calculate_hash()
            if self.hash.startswith(target):
                mining_time = time.time() - start_time
                print(f"Block {self.index} mined! Nonce: {self.nonce}, Time:
{mining_time:.2f}s")
                return self.hash
            self.nonce += 1

    def get_transactions_summary(self) -> str:
        """Get a summary of transactions in the block."""
        return f"Block {self.index}: {len(self.transactions)} transactions"

class Blockchain:
    """
    Implements a simplified blockchain with Proof-of-Work consensus.
    """

    def __init__(self, difficulty: int = 2):
        """
        Initialize the blockchain.

        Args:
            difficulty: Difficulty level for Proof-of-Work
        """
        self.chain: List[Block] = []
        self.pending_transactions: List[Transaction] = []
        self.difficulty = difficulty
        self.mining_reward = 1  # Reward for mining a block

        # Create genesis block
        self._create_genesis_block()

    def _create_genesis_block(self):
        """Create the genesis (first) block."""
        genesis_block = Block(0, [], "0", self.difficulty)
        genesis_block.mine_block()
        self.chain.append(genesis_block)

    def add_transaction(self, transaction: Transaction) -> bool:
        """
```

```
        Add a transaction to the pending transactions pool.

        Args:
            transaction: Transaction to add

        Returns:
            True if transaction was added successfully
        """
        self.pending_transactions.append(transaction)
        return True

    def mine_pending_transactions(self, miner_id: str) -> bool:
        """
        Mine pending transactions into a new block.

        Args:
            miner_id: Identifier of the miner

        Returns:
            True if mining was successful
        """
        if not self.pending_transactions:
            print("No pending transactions to mine.")
            return False

        # Create a new block with pending transactions
        new_block = Block(
            index=len(self.chain),
            transactions=self.pending_transactions,
            previous_hash=self.chain[-1].hash,
            difficulty=self.difficulty
        )

        # Mine the block
        new_block.mine_block()

        # Add block to chain
        self.chain.append(new_block)

        # Clear pending transactions
        self.pending_transactions = []
        return True

    def get_latest_block(self) -> Block:
        """Get the latest block in the chain."""
        return self.chain[-1]

    def is_chain_valid(self) -> bool:
        """
        Validate the integrity of the blockchain.

        Returns:
```

```
            True if the chain is valid, False otherwise
        """
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            # Verify current block hash
            if current_block.hash != current_block.calculate_hash():
                print(f"Invalid hash at block {i}")
                return False

            # Verify previous block hash
            if current_block.previous_hash != previous_block.hash:
                print(f"Invalid previous hash at block {i}")
                return False

            # Verify Proof-of-Work
            target = '0' * current_block.difficulty
            if not current_block.hash.startswith(target):
                print(f"Invalid Proof-of-Work at block {i}")
                return False

        return True

    def get_chain_summary(self) -> str:
        """Get a summary of the blockchain."""
        summary = f"Blockchain Summary:\n"
        summary += f"  Total blocks: {len(self.chain)}\n"
        summary += f"  Pending transactions: {len(self.pending_transactions)}\n"
        summary += f"  Difficulty: {self.difficulty}\n"
        summary += f"  Chain valid: {self.is_chain_valid()}\n"
        return summary

    def get_block_details(self, block_index: int) -> Dict[str, Any]:
        """
        Get detailed information about a specific block.

        Args:
            block_index: Index of the block

        Returns:
            Dictionary containing block details
        """
        if block_index >= len(self.chain):
            return None

        block = self.chain[block_index]
        return {
            'index': block.index,
            'timestamp': datetime.fromtimestamp(block.timestamp).isoformat(),
            'transactions': len(block.transactions),
            'hash': block.hash,
```

```
            'previous_hash': block.previous_hash,
            'nonce': block.nonce,
            'difficulty': block.difficulty
        }
    def search_transaction(self, query_id: str) -> Dict[str, Any]:
        """
        Search for a transaction in the blockchain.

        Args:
            query_id: Query ID to search for

        Returns:
            Dictionary containing transaction details if found, None otherwise
        """
        for block in self.chain:
            for tx in block.transactions:
                if tx.query_id == query_id:
                    return {
                        'block_index': block.index,
                        'query_id': tx.query_id,
                        'user_id': tx.user_id,
                        'timestamp':
datetime.fromtimestamp(tx.timestamp).isoformat(),
                        'results': tx.result_ids
        return None
                    }
class SmartContract:
    """
    Simulates a smart contract that executes search queries on the blockchain.
    """

    def __init__(self, searchable_encryption):
        """
        Initialize the smart contract.

        Args:
            searchable_encryption: SearchableEncryption instance for performing
searches
        """
        self.searchable_encryption = searchable_encryption
        self.execution_log = []

    def execute_search(self, trapdoor: str, query_id: str) -> List[str]:
        """
        Execute a search query using the trapdoor.

        Args:
            trapdoor: The search trapdoor
            query_id: Unique identifier for this query

        Returns:
            List of document IDs matching the search
```

```
        """
        # Search in the inverted index
        index = self.searchable_encryption.get_index()
        results = index.get(trapdoor, [])

        # Log the execution
        self.execution_log.append({
            'query_id': query_id,
            'trapdoor': trapdoor,
            'results_count': len(results),
            'timestamp': time.time()
        })
        return results

    def get_execution_log(self) -> List[Dict]:
        """Get the execution log of the smart contract."""
        return self.execution_log.copy()

# Example usage and testing
if __name__ == "__main__":
    print("=" * 70)
    print("Blockchain Module Test")
    print("=" * 70)

    # Initialize blockchain
    print("\n1. Creating Blockchain:")
    print("-" * 70)
    blockchain = Blockchain(difficulty=2)
    print("Genesis block created.")
    print(blockchain.get_chain_summary())

    # Create and add transactions
    print("\n2. Adding Transactions:")
    print("-" * 70)

    transactions = [
        Transaction("query_001", "trapdoor_blockchain", "user_1"),
        Transaction("query_002", "trapdoor_security", "user_2"),
        Transaction("query_003", "trapdoor_encryption", "user_1"),
    ]
    for tx in transactions:
        blockchain.add_transaction(tx)
        print(f"Added transaction: {tx.query_id}")

    print(f"Pending transactions: {len(blockchain.pending_transactions)}")

    # Mine block
    print("\n3. Mining Block:")
    print("-" * 70)
    blockchain.mine_pending_transactions("miner_1")
    print("Block mined and added to chain.")
```

```
    # Add more transactions
    print("\n4. Adding More Transactions:")
    print("-" * 70)
    more_transactions = [
        Transaction("query_004", "trapdoor_data", "user_3"),
        Transaction("query_005", "trapdoor_privacy", "user_2"),
    ]
    for tx in more_transactions:
        blockchain.add_transaction(tx)
        print(f"Added transaction: {tx.query_id}")

    # Mine another block
    print("\n5. Mining Another Block:")
    print("-" * 70)
    blockchain.mine_pending_transactions("miner_1")
    print("Second block mined and added to chain.")

    # Display blockchain summary
    print("\n6. Blockchain Summary:")
    print("-" * 70)
    print(blockchain.get_chain_summary())

    # Display block details
    print("\n7. Block Details:")
    print("-" * 70)
    for i in range(len(blockchain.chain)):
        details = blockchain.get_block_details(i)
        print(f"Block {i}: {details['transactions']} transactions, Hash:
{details['hash'][:16]}...")

    # Search for transaction
    print("\n8. Transaction Search:")
    print("-" * 70)
    result = blockchain.search_transaction("query_002")
    if result:
        print(f"Found transaction: {result}")
    else:
        print("Transaction not found")

    # Validate chain
    print("\n9. Chain Validation:")
    print("-" * 70)
    is_valid = blockchain.is_chain_valid()
    print(f"Chain is valid: {is_valid}")
    print("\n" + "=" * 70)
    print("Test completed successfully!")
    print("=" * 70)
```

A.3 Integrated System — integrated_system.py:

The integrated system module orchestrates the complete end-to-end workflow: document encryption and index construction, trapdoor-based search, blockchain transaction management, smart contract execution, and document retrieval with decryption.

```
"""
Integrated System: Combines Encryption, Searchable Encryption, and Blockchain
Author: Manus AI
Date: 2026
"""
import time
from encryption import EncryptionManager, SearchableEncryption
from blockchain import Blockchain, Transaction, SmartContract


class SecureSearchSystem:
    """
    Integrated system for secure search over encrypted data using blockchain.
    """

    def __init__(self, master_key: str = "secure_master_key_2026",
blockchain_difficulty: int = 2):
        """
        Initialize the secure search system.

        Args:
            master_key: Master encryption key
            blockchain_difficulty: Difficulty level for blockchain mining
        """
        # Initialize encryption components
        self.encryption_manager = EncryptionManager(master_key)
        self.searchable_encryption = SearchableEncryption(self.encryption_manager)

        # Initialize blockchain
        self.blockchain = Blockchain(difficulty=blockchain_difficulty)

        # Initialize smart contract
        self.smart_contract = SmartContract(self.searchable_encryption)

        # Storage for encrypted documents
        self.encrypted_documents = {}
        self.document_metadata = {}

        # Query counter
        self.query_counter = 0

    def upload_documents(self, documents: dict) -> dict:
        """
        Upload and encrypt documents to the system.
```

```
        Args:
            documents: Dictionary mapping document IDs to document content

        Returns:
            Dictionary containing encryption metadata
        """
        print("\n" + "=" * 70)
        print("PHASE 1: Document Upload and Encryption")
        print("=" * 70)

        upload_metadata = {}

        for doc_id, content in documents.items():
            print(f"\nEncrypting document: {doc_id}")

            # Encrypt the document
            encrypted_data = self.encryption_manager.encrypt(content)
            self.encrypted_documents[doc_id] = encrypted_data

            # Store metadata
            self.document_metadata[doc_id] = {
                'original_size': len(content),
                'encrypted_size': len(encrypted_data['ciphertext']),
                'upload_time': time.time()
            }

            upload_metadata[doc_id] = {
                'encrypted': True,
                'ciphertext_preview': encrypted_data['ciphertext'][:50] + '...',
                'nonce': encrypted_data['nonce']
            }

            print(f"  ✓ Document encrypted successfully")
            print(f"    Original size:
{self.document_metadata[doc_id]['original_size']} bytes")
            print(f"    Encrypted size:
{self.document_metadata[doc_id]['encrypted_size']} bytes")

        # Build searchable index
        print(f"\nBuilding searchable index from {len(documents)} documents...")
        self.searchable_encryption.build_index(documents)
        print(f"✓ Searchable index built with
{len(self.searchable_encryption.get_index())} trapdoors")

        return upload_metadata

    def search_documents(self, keyword: str, user_id: str) -> dict:
        """
        Search for documents containing a keyword.

        Args:
            keyword: Keyword to search for
```

```
            user_id: ID of the user performing the search

        Returns:
            Dictionary containing search results and metadata
        """
        self.query_counter += 1
        query_id = f"query_{self.query_counter:04d}"

        print("\n" + "=" * 70)
        print("PHASE 2: Search Query Execution")
        print("=" * 70)
        print(f"\nSearching for keyword: '{keyword}'")
        print(f"Query ID: {query_id}")
        print(f"User ID: {user_id}")

        # Generate trapdoor
        print(f"\nGenerating trapdoor...")
        trapdoor = self.searchable_encryption.generate_trapdoor(keyword)
        print(f"✓ Trapdoor generated (preview): {trapdoor[:30]}...")

        # Create transaction
        print(f"\nCreating blockchain transaction...")
        transaction = Transaction(query_id, trapdoor, user_id)

        # Execute search via smart contract
        print(f"\nExecuting smart contract search...")
        matching_docs = self.smart_contract.execute_search(trapdoor, query_id)
        print(f"✓ Search executed. Found {len(matching_docs)} matching documents")

        # Update transaction with results
        transaction.set_results(matching_docs)

        # Add transaction to blockchain
        print(f"\nAdding transaction to blockchain...")
        self.blockchain.add_transaction(transaction)
        print(f"✓ Transaction added to pending pool")

        # Mine the transaction
        print(f"\nMining transaction into blockchain...")
        self.blockchain.mine_pending_transactions(f"miner_{user_id}")
        print(f"✓ Transaction mined and added to blockchain")

        # Get the latest block
        latest_block = self.blockchain.get_latest_block()

        return {
            'query_id': query_id,
            'keyword': keyword,
            'trapdoor_preview': trapdoor[:30] + '...',
            'matching_documents': matching_docs,
            'block_index': latest_block.index,
            'block_hash': latest_block.hash[:16] + '...',
```


https://doi.org/10.59992/IJCI.2026.v5n5p2
E-ISSN 2976-9361

```
            'timestamp': time.time()
        }

    def retrieve_document(self, doc_id: str) -> dict:
        """
        Retrieve and decrypt a document.

        Args:
            doc_id: Document ID to retrieve

        Returns:
            Dictionary containing the decrypted document
        """
        print("\n" + "=" * 70)
        print("PHASE 3: Document Retrieval and Decryption")
        print("=" * 70)

        if doc_id not in self.encrypted_documents:
            return {'error': f"Document {doc_id} not found"}

        print(f"\nRetrieving document: {doc_id}")

        # Get encrypted data
        encrypted_data = self.encrypted_documents[doc_id]

        # Decrypt
        print(f"Decrypting document...")
        try:
            plaintext = self.encryption_manager.decrypt(
                encrypted_data['ciphertext'],
                encrypted_data['nonce']
            )
            print(f"✓ Document decrypted successfully")

            return {
                'doc_id': doc_id,
                'content': plaintext,
                'decryption_successful': True,
                'size': len(plaintext)
            }
        except Exception as e:
            print(f"✗ Decryption failed: {str(e)}")
            return {'error': str(e)}
print(f"    - Query {tx.query_id}: {len(tx.result_ids)} results")
```