

# A Patient-Centric Blockchain Framework for Secure Electronic Health Record Management: Decoupling Data Storage from Access Control

Tanzim Hossain Romel<sup>1</sup>, Kawshik Kumar Paul<sup>1</sup>, Tanberul Islam Ruhan<sup>1</sup>, Maisha Rahman Mim<sup>1</sup>, and Abu Sayed Md. Latiful Hoque<sup>1</sup>

<sup>1</sup>Department of Computer Science & Engineering, Bangladesh University of Engineering & Technology, Dhaka, Bangladesh

## Abstract

We present a patient-centric architecture for electronic health record (EHR) sharing that separates content storage from authorization and audit. Encrypted FHIR resources are stored off-chain; a public blockchain records only cryptographic commitments and patient-signed, time-bounded permissions using EIP-712. Keys are distributed via public-key wrapping, enabling storage providers to remain honest-but-curious without risking confidentiality. We formalize security goals (confidentiality, integrity, cryptographically attributable authorization, and auditability of authorization events) and provide a Solidity reference implementation deployed as single-patient contracts. On-chain costs for permission grants average **78,000 gas** (L1), and end-to-end access latency for 1 MB records is **0.7–1.4 s** (mean values for S3 and IPFS respectively), dominated by storage retrieval. Layer-2 deployment reduces gas usage by 10–13×, though data availability charges dominate actual costs. We discuss metadata privacy, key registry requirements, and regulatory considerations (HIPAA/GDPR), demonstrating a practical route to restoring patient control while preserving security properties required for sensitive clinical data.

**Keywords:** Blockchain, Electronic Health Records, Access Control, Healthcare Privacy, Smart Contracts, FHIR, Cryptographic Protocols

## 1 Introduction

The digitization of healthcare has transformed medical practice, enabling evidence-based decision-making and population health management at unprecedented scales. However, health records remain trapped in organizational silos with incompatible systems. When patients seek care from multiple providers or relocate, critical medical history becomes inaccessible, leading to duplicated tests, adverse drug interactions, and suboptimal treatment decisions.

### 1.1 The Centralization Problem

Contemporary health information exchange architectures exhibit three fundamental weaknesses. First, they create single points of failure where system compromise can affect millions of patient records. Major healthcare data breaches have exposed the medical information of hundreds of millions of individuals [1]. Second, centralized systems require patients to trust intermediary organizations with unfettered access. While policies constrain behavior, insider threats persist, and audit logs maintained by audited entities offer limited assurance. Third, patients exercise minimal control over sharing, contradicting principles of autonomy and informed consent.

### 1.2 Blockchain as an Architectural Primitive

Blockchain technology addresses specific weaknesses through replicated, append-only ledgers where transactions are cryptographically verified rather than institutionally authorized. However, naive blockchain application introduces problems: storing protected health information on public blockchains violates privacy through transparency and immutability, and transaction costs make large document storage impractical.

The key insight is architectural separation: encrypted records reside in off-chain storage optimized for large objects, while blockchain serves exclusively as authorization layer and integrity mechanism. This exploits complementary strengths while avoiding respective weaknesses.

### 1.3 Deployment Model

Our architecture deploys **one contract per patient** rather than a multi-tenant registry. This design choice provides strong isolation between patients' data, simplifies permission management, and aligns with patient sovereignty principles. While this increases deployment costs (one-time contract creation), it eliminates cross-patient vulnerabilities and simplifies auditing. Health-

care institutions can deploy patient contracts on their behalf with appropriate delegation mechanisms. The contract does not use the ERC-721 token standard, instead implementing a simpler patient-specific authorization model.

#### 1.4 Contributions

1. **Formal Architecture:** We specify how off-chain encrypted storage combined with on-chain access control achieves confidentiality against curious storage providers, integrity verification, and patient-controlled authorization with cryptographic attribution.
2. **Reference Implementation:** Complete Ethereum smart contract handling record registration, permission granting through signed messages with explicit nonce management, time-bounded access with revocation, update/rotation capabilities, and comprehensive auditability through event logs.
3. **Healthcare Integration:** Integration with HL7 FHIR standards, showing how FHIR resources serve as plaintext while supporting de-identified data release.
4. **Performance Characterization:** Empirical evaluation showing gas costs, latency profiles, and scalability across Layer-1 and Layer-2 deployments.

## 2 Background

### 2.1 Threat Model

We consider adversaries with varying capabilities:

1. **Storage Provider ( $\mathcal{S}$ ):** Honest-but-curious cloud provider (IPFS, AWS S3) who stores encrypted records.  $\mathcal{S}$  follows protocol but attempts to learn patient information from stored data.
2. **Network Adversary ( $\mathcal{N}$ ):** Observes blockchain transactions and network traffic. Cannot break cryptographic primitives but can analyze patterns, timing, and metadata.
3. **Revoked Recipient ( $\mathcal{R}$ ):** Previously authorized healthcare provider whose access was revoked. Possesses historical wrapped keys and may have cached plaintext from authorized period.
4. **Malicious Provider ( $\mathcal{M}$ ):** Healthcare provider attempting unauthorized access or privilege escalation beyond granted permissions.

We assume cryptographic primitives are secure: adversaries cannot break AES-256-GCM, ECIES on secp256k1 (using standard KDF/MAC/encoding from audited libraries), or forge ECDSA signatures.

### 2.2 Cryptographic Building Blocks

**Symmetric Encryption:** AES-256-GCM provides authenticated encryption with associated data (AEAD). Given key  $K$ , nonce  $N$ , plaintext  $M$ , and associated data  $AD$ :  $\text{Enc}(K, N, M, AD) \rightarrow (C, T)$  where  $C$  is ciphertext,  $T$  is authentication tag.

**Nonce Requirements:** AES-GCM security critically depends on nonce uniqueness. Implementations MUST use cryptographically secure random number generators (CSPRNG) or counter-mode deterministic random bit generators (CTR-DRBG) to ensure uniqueness. Consider XChaCha20-Poly1305 if nonce management is operationally risky, as it provides a larger nonce space and better misuse-resistance.

**Public Key Encryption:** ECIES (Elliptic Curve Integrated Encryption Scheme) on secp256k1 curve provides IND-CCA2 secure public key encryption. We use the following configuration for standards compliance:

- KDF: ANSI X9.63 with SHA-256
- DEM: AES-128-CTR
- MAC: HMAC-SHA-256
- Point encoding: Uncompressed (0x04 prefix)
- Ephemeral key included in ciphertext

**Digital Signatures:** ECDSA signatures on secp256k1 provide authentication and non-repudiation. EIP-712 structured data signing prevents signature malleability across contexts.

**Cryptographic Hash Functions:** We use SHA-256 for content digests throughout the system. SHA-256 provides 256-bit collision resistance and is the standard for IPFS content addressing and general cryptographic applications. The digest  $d$  is consistently defined as  $d = \text{SHA-256}(C||T||N||AD)$  where  $C$  is ciphertext,  $T$  is authentication tag,  $N$  is nonce, and  $AD$  is associated data.

### 2.3 Blockchain Infrastructure

**Smart Contracts:** Ethereum smart contracts execute deterministically based on transaction inputs. Gas fees incentivize efficient code and prevent denial-of-service.

**Layer-2 Solutions:** Rollups (Optimistic/ZK) reduce costs by batching transactions. However, data availability charges often dominate L2 costs, particularly for calldata-heavy operations like storing wrapped keys.

**Events:** Smart contract events provide efficient, queryable logs. Events cost 375 gas base + 375 gas per topic + 8 gas/byte for data.

## 2.4 Healthcare Standards

**HL7 FHIR:** Fast Healthcare Interoperability Resources define standard formats for clinical data. Resources include Patient, Observation, Medication, Procedure, etc. We use FHIR R4.

**HIPAA:** Requires access controls, audit logs, and data encryption. Our architecture maps to HIPAA's administrative (identity-based access control with explicit authorization), physical (N/A for digital systems), and technical safeguards (encryption at rest and in transit).

**GDPR:** European privacy regulation granting data subject rights. Blockchain immutability creates tension with “right to be forgotten”—we address through minimal on-chain data, treating blockchain entries as legally-required audit logs potentially exempt from erasure under Article 17(3)(b), and implementing data minimization strategies.

## 3 System Architecture

### 3.1 Overview

The system separates concerns across three layers:

- Storage Layer:** Distributed storage (IPFS) or cloud storage (S3) holds encrypted health records. Storage providers see only ciphertext.
- Blockchain Layer:** Ethereum smart contracts manage metadata, permissions, and audit trails. Each patient has a dedicated contract instance.
- Application Layer:** Client applications handle encryption/decryption, signature generation, and user interfaces.

### 3.2 Key Management Architecture

**Key Registry Contract:** A separate registry contract maintains current encryption public keys for all participants. The invariant is that `getKey(user)` returns the latest non-revoked key and its version. Clients MUST fetch the key immediately prior to wrapped key computation to avoid time-of-check-time-of-use (TOCTOU) issues:

```

1 interface IKeyRegistry {
2     function registerKey(bytes memory publicKey)
3         external;
4     function rotateKey(bytes memory newPublicKey)
5         external;
6     function getKey(address user)
7         external view returns (bytes memory, uint256
8         version);
9     function revokeKey() external;
10 }
```

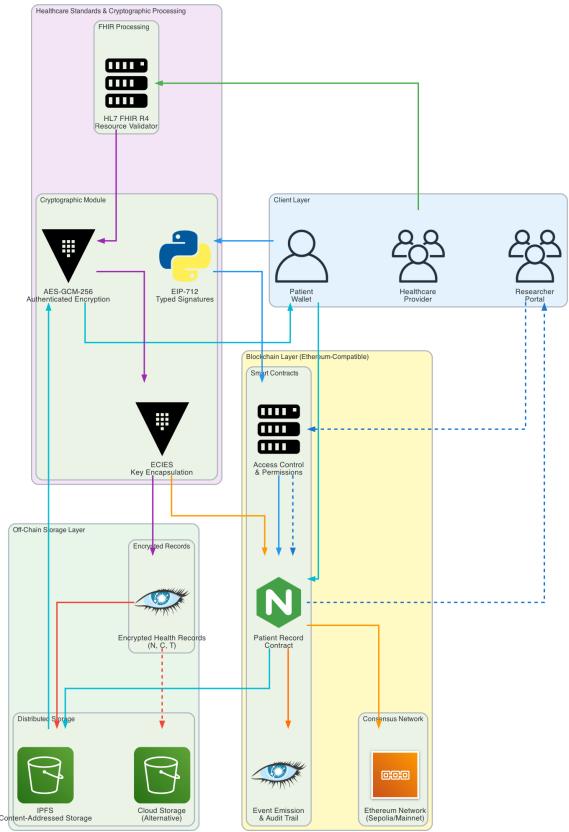


Figure 1: System architecture showing separation between on-chain authorization and off-chain encrypted storage. Digest  $d = \text{SHA-256}(C||T||N||AD)$ .

```

7     event KeyRegistered(address indexed user, bytes
8         publicKey);
9     event KeyRotated(address indexed user,
10        bytes newKey, uint256 version);
11    event KeyRevoked(address indexed user);
12 }
```

Listing 1: Key Registry Interface

### 3.3 Data Model

Each health record consists of:

- Record ID ( $rid$ ):** Unique identifier within patient's contract
- Plaintext ( $M$ ):** FHIR resource bundle in JSON format
- Symmetric Key ( $SymmK$ ):** AES-256 key for record encryption
- Ciphertext ( $C$ ):** Encrypted record stored off-chain

- **Storage Pointer ( $ptr$ ):** IPFS CID or S3 URL
- **Content Digest ( $d$ ):** SHA-256 hash of complete ciphertext blob:  $d = \text{SHA-256}(C||T||N||AD)$
- **Wrapped Keys ( $W$ ):** ECIES-encrypted  $SymmK$  for authorized parties
- **Permissions:** Time-bounded access grants with wrapped keys

### 3.4 AEAD Metadata Considerations

When using AEAD, the associated data (AD) parameter authenticates but does NOT encrypt additional context. To prevent metadata leakage:

1. **Minimal AD:** Use only non-sensitive, constant values (e.g., version number, fixed resource type identifier)
2. **Encrypted Metadata:** Include sensitive metadata (timestamps, detailed resource types) within the encrypted payload  $M$  itself
3. **Constant Format:** Ensure AD format doesn't vary in ways that leak information through length or structure

## 4 Protocol Workflows

### 4.1 Workflow 1: Record Creation

Patient creates new health record:

1. **Prepare Plaintext:** Construct FHIR bundle  $M$  containing clinical data.
2. **Generate Symmetric Key:** Generate random AES-256 key:  $SymmK \leftarrow \{0,1\}^{256}$  using CSPRNG.
3. **Encrypt Record:** Using AES-256-GCM with CSPRNG-generated nonce:  
 $(C, T) \leftarrow \text{AES-GCM-Enc}(SymmK, N, M, AD_{minimal})$   
 where  $AD_{minimal}$  contains only non-sensitive version identifier.
4. **Upload to Storage:** Store  $(C, T, N, AD_{minimal})$  to IPFS/S3. Receive storage pointer  $ptr$ .
5. **Compute Digest:**  $d \leftarrow \text{SHA-256}(C||T||N||AD_{minimal})$ .
6. **Wrap Key for Owner:** Using patient's public key from registry:

$$W_{owner} \leftarrow \text{ECIES-Enc}(PK_P, SymmK)$$

7. **Register On-Chain:** Call  $\text{addRecord}(ptr, d, W_{owner})$ . Contract stores metadata, assigns  $rid$ , emits  $\text{RecordAdded}(rid, d, ptr)$ .

### 4.2 Workflow 2: Permission Grant

Patient grants time-bounded access using EIP-712 signatures. Note that patients MUST generate unique nonces for each grant (e.g., 256-bit random values) to enable parallel permission grants. Figure 2 illustrates this workflow:

1. **Retrieve Recipient Key:** Query key registry for recipient's current public key  $PK_R$  immediately before wrapping.
  2. **Wrap Symmetric Key:**  $W_R \leftarrow \text{ECIES-Enc}(PK_R, SymmK)$ .
  3. **Generate Unique Nonce:** Patient generates unique nonce (256-bit random recommended).
  4. **Prepare Typed Data:** Construct EIP-712 message with explicit nonce:
- ```

{
  recordId: rid,
  grantee: addr_R,
  expiration: timestamp,
  wrappedKey: W_R,
  nonce: uniqueRandomNonce
}

```
5. **Sign Message:**  $\sigma \leftarrow \text{ECDSA-Sign}(SK_P, \text{TypedDataHash}(message))$ .
  6. **Recipient Submits:** Recipient calls  $\text{grantPermissionBySig}(rid, expiration, W_R, nonce, \sigma)$ .
  7. **Contract Verification:** Contract verifies signature, checks nonce hasn't been used, stores permission, marks nonce as consumed, emits  $\text{PermissionGranted}$ .

### 4.3 Workflow 3: Record Access

Authorized recipient retrieves and decrypts record as shown in Figure 3. Note that the contract optionally gates metadata for UX consistency; confidentiality relies solely on encryption:

1. **Request Metadata:** Call  $\text{getRecordMetadata}(rid)$  which returns  $(ptr, d)$  if authorized. The gating is for user experience; the same data is available in public events. Patient additionally receives  $W_{owner}$  through  $\text{getOwnerWrappedKey}(rid)$ .
2. **Retrieve Ciphertext:** Fetch  $(C, T, N, AD)$  from storage using  $ptr$ .
3. **Verify Integrity:** Compute  $d' = \text{SHA-256}(C||T||N||AD)$ . Verify  $d' = d$ .

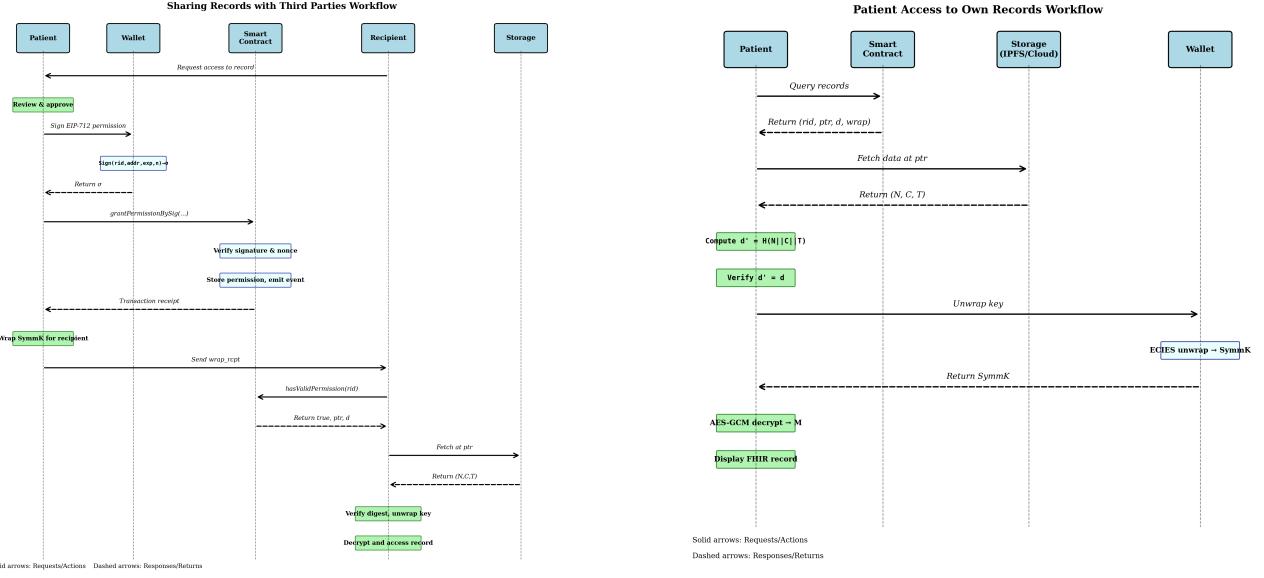


Figure 2: Permission grant workflow using EIP-712 signed messages with explicit nonce management. Each nonce can be used only once.

4. **Unwrap Key:** Decrypt wrapped key using recipient's private key:

$$SymmK \leftarrow \text{ECIES-Dec}(SK_R, W_R)$$

5. **Decrypt Record:**  $M \leftarrow \text{AES-GCM-Dec}(SymmK, N, C, T, AD)$ .
6. **Optional: Log Access:** Call  $\logAccess(rid, \text{SHA-256}(accessDetails))$  to create on-chain access receipt (not just authorization).

#### 4.4 Workflow 4: Permission Revocation

Patient revokes access:

1. **Submit Revocation:** Patient calls  $\text{revokePermission}(rid, addr_R)$ . Contract sets  $\text{permissions}[rid][addr_R].revoked = true$  and emits  $\text{PermissionRevoked}(rid, addr_R)$ .
2. **Future Access Blocked:** Subsequent  $\text{getRecordMetadata}(rid)$  calls by  $addr_R$  fail permission check.

**Limitation:** Revocation prevents future access but cannot retract already-decrypted plaintext. If recipient downloaded  $M$  before revocation, they retain that data. This is fundamental to cryptographic access control and must be clearly communicated to patients.

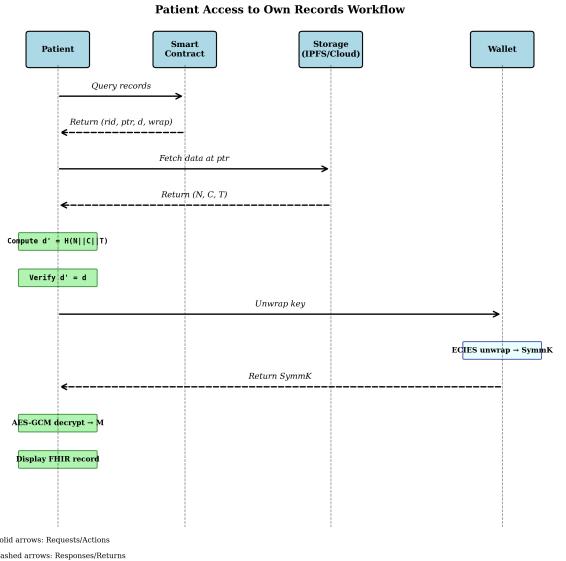


Figure 3: Record access workflow. Contract gates metadata for UX consistency (data also in events). Digest verification:  $d = \text{SHA-256}(C||T||N||AD)$ .

#### 4.5 Workflow 5: Record Update/Key Rotation

For stronger guarantees when trust relationships end or to update record content:

1. **Generate New Key:**  $SymmK' \leftarrow \{0,1\}^{256}$ .
2. **Re-encrypt Record:** Encrypt updated content  $M'$  with  $SymmK'$ , upload to storage, get new  $ptr'$ .
3. **Compute New Digest:**  $d' = \text{SHA-256}(C'||T'||N'||AD')$ .
4. **Update On-Chain:** Call  $\text{updateRecord}(rid, ptr', d', W_{owner})$ .
5. **Invalidate Old Version:** Previous  $(ptr, d, SymmK)$  become obsolete. Revoked recipients cannot access new version.
6. **Emit Event:** Contract emits  $\text{RecordUpdated}(rid, d', ptr')$  for audit trail.

### 5 Smart Contract Implementation

#### 5.1 Design Rationale

The smart contract serves three roles: (1) **Metadata Registry**—storing storage pointers and content digests; (2) **Authorization Engine**—verifying permissions before metadata release; (3) **Audit Log**—emitting events for all operations.

We deploy **one contract per patient** using a simple authorization model without token standards. This provides:

- Strong isolation between patients
- Simplified permission model (no cross-patient checks)
- Clear ownership semantics
- Independent upgrade paths per patient

While this increases deployment costs, it eliminates shared-state vulnerabilities and aligns with patient sovereignty.

## 5.2 Core Data Structures

```

1 contract PatientHealthRecords is EIP712 {
2     address public immutable patient;
3     uint256 private _recordCounter;
4
5     struct RecordMetadata {
6         string storagePointer;
7         bytes32 contentDigest;
8         bytes wrappedKeyOwner;
9         uint64 createdAt;
10        uint64 updatedAt;
11    }
12
13    struct Permission {
14        uint64 expiration;
15        bool revoked;
16        bytes wrappedKey;
17    }
18
19    mapping(uint256 => RecordMetadata) private
20        _records;
21    mapping(uint256 => mapping(address => Permission)) public permissions;
22    mapping(bytes32 => bool) public usedNonces;
23
24    modifier onlyPatient() {
25        require(msg.sender == patient, "Only patient")
26        ;
27    }
28
29    modifier validRecordId(uint256 rid) {
30        require(rid > 0 && rid <= _recordCounter,
31            "Invalid record ID");
32        -
33    }
34
35    // Event Declarations
36    event RecordAdded(uint256 indexed rid, bytes32
37        digest, string ptr);
38    event RecordUpdated(uint256 indexed rid, bytes32
39        digest, string ptr);
40    event PermissionGranted(uint256 indexed rid,
41        address indexed grantee, uint64 expiration);
42    event PermissionRevoked(uint256 indexed rid,
43        address indexed grantee);
44    event EmergencyAccessGranted(bytes32 indexed
45        grantId,
46        uint256 indexed rid, address physician1,
47        address physician2,
48        uint8 justificationCode, uint64 expiration,
49        uint64 requestTime);
50    event EmergencyAccessConfirmed(bytes32 indexed
51        grantId,
52        uint256 indexed rid, address physician,
53        bytes32 justificationHash);
54    event AccessLogged(uint256 indexed rid,
55        address indexed accessor, bytes32 detailsHash)
56        ;
57 }
```

Listing 2: Smart Contract Data Structures

## 5.3 Record Registration

```

1 function addRecord(
2     string memory ptr,
3     bytes32 digest,
4     bytes memory wrappedKey
5 ) external onlyPatient returns (uint256) {
6     _recordCounter++;
7     uint256 rid = _recordCounter;
8
9     _records[rid] = RecordMetadata({
10        storagePointer: ptr,
11        contentDigest: digest,
12        wrappedKeyOwner: wrappedKey,
13        createdAt: uint64(block.timestamp),
14        updatedAt: uint64(block.timestamp)
15    });
16
17    emit RecordAdded(rid, digest, ptr);
18    return rid;
19 }
```

Listing 3: Record Registration Function

Gas cost: ~180,000 gas first record (cold storage), ~165,000 gas subsequent records.

## 5.4 Signature-Based Permission Grant with Explicit Nonce

Gas cost: ~78,000 gas. The ecrecover precompile itself costs ~3,000 gas; storage operations and calldata processing account for the remainder.

## 5.5 Permission Verification and Metadata Access

## 5.6 Record Update and Key Rotation

## 5.7 Permission Revocation

Gas cost: ~31,000 gas.

## 5.8 Emergency Access Pattern

When patients are incapacitated and cannot grant permissions, emergency access is critical. The two-physician multisignature pattern ensures medical necessity while maintaining accountability. The wrapped keys for emergency physicians are generated by an institutional guardian service (HSM-backed) that unwraps the patient's owner key (or uses a pre-established envelope key) and re-wraps for each authorized physician—see §9.2 for institutional key management details. The contract merely anchors authorization and audit.

```

1 mapping(address => bool) public emergencyPhysicians;
2 mapping(bytes32 => EmergencyGrant) public
3     emergencyGrants;
4
5 struct EmergencyRequest {
6     uint256 rid;
7     uint8 justificationCode;
8     uint64 requestTime;
9     uint64 maxSkewSeconds;
10 }
```

```

1  function grantPermissionBySig(
2    uint256 rid,
3    uint64 expiration,
4    bytes memory wrappedKey,
5    uint256 nonce,
6    bytes memory signature
7 ) external validRecordId(rid) {
8   // Construct nonce hash to prevent reuse
9   bytes32 nonceHash = keccak256(
10     abi.encodePacked(patient, nonce)
11   );
12   require(!usedNonces[nonceHash], "Nonce already
13     used");
14
15   // Recover signer with provided nonce
16   address signer = _recoverSigner(
17     rid, msg.sender, expiration,
18     wrappedKey, nonce, signature
19   );
20   require(signer == patient, "Invalid signature");
21
22   // Check expiration is future
23   require(expiration > block.timestamp,
24     "Expiration must be future");
25
26   // Mark nonce as used
27   usedNonces[nonceHash] = true;
28
29   // Store permission
30   permissions[rid][msg.sender] = Permission({
31     expiration: expiration,
32     revoked: false,
33     wrappedKey: wrappedKey
34   });
35
36   emit PermissionGranted(rid, msg.sender, expiration
37 );
38
39   function _recoverSigner(
40     uint256 rid, address grantee, uint64 expiration,
41     bytes memory wk, uint256 nonce, bytes memory sig
42   ) internal view returns (address) {
43     bytes32 structHash = keccak256(abi.encode(
44       PERMISSION_TYPEHASH,
45       rid, grantee, expiration,
46       keccak256(wk), nonce
47     ));
48     bytes32 hash = _hashTypedDataV4(structHash);
49     return ECDSA.recover(hash, sig);
50   }

```

Listing 4: Permission Grant with Explicit Nonce Management

```

1  function hasValidPermission(uint256 rid)
2   public view validRecordId(rid)
3   returns (bool) {
4   // Patient always has access
5   if (msg.sender == patient) return true;
6
7   Permission memory p = permissions[rid][msg.sender
8     ];
9   return !p.revoked &&
10      p.expiration > 0 &&
11      p.expiration > block.timestamp; // Strict
12      inequality
13  }
14
15  function getRecordMetadata(uint256 rid)
16   external view validRecordId(rid)
17   returns (string memory ptr, bytes32 digest) {
18   require(hasValidPermission(rid),
19     "Not authorized");
20
21   RecordMetadata memory rec = _records[rid];
22   ptr = rec.storagePointer;
23   digest = rec.contentDigest;
24   // Note: This gating is for UX; data is public in
25   events
26  }
27
28  // New function for owner to retrieve their wrapped
29  // key
30  function getOwnerWrappedKey(uint256 rid)
31   external view onlyPatient validRecordId(rid)
32   returns (bytes memory) {
33   return _records[rid].wrappedKeyOwner;
34  }

```

Listing 5: Permission Check with Owner Key Retrieval

```

34  require(msg.sender != physician2,
35    "Different physicians required");
36
37  // Check time skew tolerance
38  uint256 timeDiff = (block.timestamp > requestTime)
39    ?
40      block.timestamp - uint256(requestTime) :
41      uint256(requestTime) - block.timestamp;
42  require(timeDiff <= uint256(maxSkewSeconds),
43    "Request time outside tolerance");
44
45  // Verify both signatures over EIP-712 struct
46  bytes32 requestHash = _hashTypedDataV4(
47    keccak256(abi.encode(
48      EMERGENCY_REQUEST_TYPEHASH,
49      rid,
50      justificationCode,
51      requestTime,
52      maxSkewSeconds
53    )));
54
55  address signer1 = ECDSA.recover(requestHash,
56    signature1);
57  address signer2 = ECDSA.recover(requestHash,
58    signature2);
59  require(signer1 == msg.sender && signer2 ==
60    physician2,
61    "Invalid signatures");
62
63  // Create 2-hour emergency grant
64  uint64 emergencyExpiration = uint64(block.
65    timestamp + 2 hours);
66
67  // Store wrapped keys for both physicians
68  permissions[rid][msg.sender] = Permission({
69    expiration: emergencyExpiration,
70    revoked: false,
71    wrappedKey: wrappedKey1
72  });
73
74  permissions[rid][physician2] = Permission({
75    expiration: emergencyExpiration,
76    revoked: false,
77    wrappedKey: wrappedKey2
78  });

```

```

10
11  struct EmergencyGrant {
12    uint256 recordId;
13    address physician1;
14    address physician2;
15    uint64 expiration;
16    bool confirmed;
17  }
18
19  function emergencyGrantAccess(
20    uint256 rid,
21    address physician2,
22    uint8 justificationCode,
23    uint64 requestTime,
24    uint64 maxSkewSeconds,
25    bytes memory wrappedKey1,
26    bytes memory wrappedKey2,
27    bytes memory signature1,
28    bytes memory signature2
29  ) external validRecordId(rid) {
30   require(emergencyPhysicians[msg.sender],
31     "Not emergency physician");
32   require(emergencyPhysicians[physician2],
33     "Not emergency physician");

```

```

1 function updateRecord(
2     uint256 rid,
3     string memory newPtr,
4     bytes32 newDigest,
5     bytes memory newOwnerWrappedKey
6 ) external onlyPatient validRecordId(rid) {
7     RecordMetadata storage rec = _records[rid];
8
9     rec.storagePointer = newPtr;
10    rec.contentDigest = newDigest;
11    rec.wrappedKeyOwner = newOwnerWrappedKey;
12    rec.updatedAt = uint64(block.timestamp);
13
14    emit RecordUpdated(rid, newDigest, newPtr);
15 }

```

Listing 6: Record Update Function

```

1 function revokePermission(
2     uint256 rid, address grantee
3 ) external onlyPatient validRecordId(rid) {
4     require(permissions[rid][grantee].expiration > 0,
5            "No permission to revoke");
6
7     permissions[rid][grantee].revoked = true;
8     emit PermissionRevoked(rid, grantee);
9 }

```

Listing 7: Permission Revocation

```

15     grantId, // Include grantId for tracking
16     grant.recordId,
17     msg.sender,
18     justificationHash
19 );
20 }

```

Listing 8: Corrected Emergency Access Implementation

## 5.9 Optional Access Logging

```

1 event AccessLogged(uint256 indexed rid,
2                     address indexed accessor, bytes32 detailsHash);
3
4 function logAccess(uint256 rid, bytes32 detailsHash)
5     external {
6     require(hasValidPermission(rid), "Not authorized")
7     ;
8     emit AccessLogged(rid, msg.sender, detailsHash);
9 }

```

Listing 9: Optional Read Receipt Logging

## 5.10 Security Properties

**Read Gating:** `getRecordMetadata` enforces authorization via `hasValidPermission` for UX consistency. The same metadata is available in public events; confidentiality relies entirely on encryption, not on gating. The `permissions` mapping is public for transparency and indexing; confidentiality relies solely on encryption—exposing `wrappedKey` ciphertexts does not endanger plaintext.

**Complete Owner Access:** Patient can always retrieve their wrapped key via `getOwnerWrappedKey`, ensuring they never lose access to their own records.

**Transparency Trade-off:** Storage pointers `ptr` and digests `d` are public in events and contract storage. This is acceptable because pointers reference encrypted content. Without wrapped keys, adversaries obtain only ciphertext.

**Replay Protection:** Explicit nonce management prevents signature replay. Each nonce can be used exactly once. Patients must generate unique nonces (e.g., 256-bit random values) for each grant to enable parallel permission grants without ordering hazards. Clients **MUST** generate a fresh 256-bit random nonce per grant and persist it until on-chain confirmation to avoid accidental reuse.

**Time-Bounded Access:** All permissions have expiration timestamps. Expired permissions fail `hasValidPermission` checks automatically, using strict inequality ( $>$  rather than  $\geq$ ) for clear expiration semantics.

## 6 Security Analysis

### 6.1 Confidentiality Against Storage Providers

**Theorem 1** (Storage Provider Confidentiality). *Assuming AES-GCM provides IND-CCA2 security and ECIES*

(with specified parameters) provides IND-CCA2 security, no honest-but-curious storage provider  $\mathcal{S}$  can distinguish encrypted health records from random strings with non-negligible advantage.

*Proof Sketch.* By contradiction. Suppose adversary  $\mathcal{S}$  has non-negligible advantage  $\epsilon$  in distinguishing encrypted records. Storage contains  $(C, T, N, AD_{minimal})$  where  $(C, T)$  are outputs of AES-GCM-Enc( $SymmK, N, M, AD_{minimal}$ ) and  $AD_{minimal}$  contains only non-sensitive version identifiers. By IND-CCA2 security of AES-GCM,  $(C, T)$  is computationally indistinguishable from random strings without  $SymmK$ . Since  $\mathcal{S}$  cannot obtain  $SymmK$  (wrapped keys  $W$  are ECIES ciphertexts using specified KDF/MAC, also indistinguishable from random without private keys),  $\mathcal{S}$  cannot distinguish  $(C, T)$  from random. This contradicts the assumption of advantage  $\epsilon > 0$ .  $\square$

## 6.2 Integrity Verification

**Theorem 2** (Tamper Detection). *Assuming SHA-256 is collision-resistant, any modification to stored records is detected with probability  $\geq 1 - 2^{-128}$ .*

*Proof Sketch.* On-chain digest  $d = \text{SHA-256}(C||T||N||AD)$  commits to encrypted blob. To pass verification, adversary must produce  $(C', T', N', AD')$  where  $\text{SHA-256}(C'||T'||N'||AD') = d$  with  $(C', T') \neq (C, T)$ . This requires finding collision in SHA-256. With 256-bit output and collision resistance, success probability is  $\leq 2^{-128}$  (birthday bound).  $\square$

**Authentication Tag:** AES-GCM's tag  $T$  provides additional integrity protection. Even if adversary finds hash collision, forging valid tag without  $SymmK$  has negligible probability (AES-GCM provides 128-bit authentication security).

## 6.3 Authorization Authenticity

**Theorem 3** (Cryptographically Attributable Authorization). *Assuming ECDSA over secp256k1 provides existential unforgeability under chosen message attacks (EUF-CMA), no adversary without patient's private key  $SK_P$  can forge valid permission signatures with non-negligible probability.*

*Proof Sketch.* Suppose adversary  $\mathcal{A}$  forges signature  $\sigma'$  for message  $(rid, addr_R, expiration, W_R, nonce)$  that passes verification. By EUF-CMA security of ECDSA, this occurs with negligible probability without  $SK_P$ . EIP-712 domain separator binds signature to specific contract and chain, preventing cross-contract/cross-chain replay. Unique nonces prevent same-message replay.  $\square$

Table 1: Threat Coverage

| Threat                 | Defense                 | Residual Risk |
|------------------------|-------------------------|---------------|
| Storage snooping       | Encryption              | None          |
| Data tampering         | Digest verification     | None          |
| Unauthorized access    | On-chain authz          | None          |
| Permission forgery     | EIP-712 signatures      | None          |
| Replay attacks         | Unique nonces           | None          |
| Audit log tampering    | Blockchain immutability | None          |
| Patient key theft      | —                       | High          |
| Malicious patient      | —                       | Inherent      |
| Storage unavailability | Redundancy              | Low           |
| DoS on blockchain      | Fees deter, multi-L2    | Medium        |

## 6.4 Replay Attack Prevention

Nonces provide replay protection:

**Property 1** (Unique Nonce Consumption). *For each patient, each nonce can be used exactly once. Signature  $\sigma$  for  $(rid, addr_R, expiration, W_R, nonce)$  is valid only if nonce has not been previously used. After successful grant, nonce is marked as consumed, invalidating all future signatures using the same nonce.*

**Time-Based Replay:** Expiration timestamps prevent long-term replay. Even if adversary captures signature, using it after expiration fails `require(expiration > block.timestamp)` check.

## 6.5 Auditability

**Property 2** (Authorization Audit Trail Completeness). *All authorization-changing operations emit events: `RecordAdded`, `PermissionGranted`, `PermissionRevoked`, `RecordUpdated`, `EmergencyAccessGranted`. Events are permanently stored in blockchain logs, queryable by any observer. The system provides complete authorization history (who was granted access), not complete access history (who actually retrieved/viewed records) unless optional `logAccess` is used.*

Blockchain immutability ensures events cannot be deleted or modified after confirmation. Patients, regulators, or auditors can reconstruct complete authorization history by filtering events for specific records or addresses.

## 6.6 Threat Analysis Summary

Table 1 summarizes threat coverage.

Table 2: Cryptographic Operation Latency

| Operation            | Mean (ms) | 95th % (ms) |
|----------------------|-----------|-------------|
| AES-GCM Enc (1 KB)   | 0.42      | 0.58        |
| AES-GCM Enc (100 KB) | 2.1       | 2.7         |
| AES-GCM Enc (1 MB)   | 18.3      | 22.1        |
| AES-GCM Enc (10 MB)  | 181.5     | 205.3       |
| AES-GCM Dec (1 MB)   | 16.8      | 20.5        |
| ECIES Key Wrap       | 3.2       | 4.1         |
| ECIES Key Unwrap     | 3.5       | 4.3         |
| ECDSA Sign (EIP-712) | 2.8       | 3.6         |
| ECDSA Verify         | 3.1       | 3.9         |
| SHA-256 (1 MB)       | 12.1      | 14.8        |

## 7 Performance Evaluation

### 7.1 Experimental Setup

**Blockchain:** Ethereum Sepolia testnet, September–October 2024. Transactions via Web3.js v4.2.1.

**Storage:** IPFS (go-ipfs v0.18) on 8-core, 16 GB RAM server. AWS S3 for comparison. Client in Dhaka, Bangladesh; IPFS nodes in Singapore; S3 in us-east-1.

**Client:** Intel Core i7-1165G7 @ 2.80GHz, 16 GB RAM, Ubuntu 22.04. Web Crypto API for AES-GCM, eth-crypto for ECIES/ECDSA.

**Workload:** Synthea v3.2.0 generated FHIR R4 resources. Record sizes: 1 KB (observations) to 10 MB (imaging reports). 50 trials per measurement for mean and 95th percentile.

### 7.2 Cryptographic Operations

Table 2 shows operation latencies.

AES-GCM achieves ~55 MB/s throughput, linear in plaintext size. Hardware AES acceleration (AES-NI) provides these speeds on modern processors. Public-key operations (ECIES, ECDSA) take 3–4 ms regardless of record size—operate on fixed-size keys/hashes. For typical sharing (one signature, one key wrap), total cryptographic overhead  $\leq 10$  ms, negligible vs network latency.

### 7.3 On-Chain Gas Costs

Table 3 reports gas consumption.

Layer-2 solutions reduce gas consumption 10–13× through batching and off-chain computation. However, data availability charges for posting calldata to L1 can dominate actual costs during high congestion periods. Fees on rollups are dominated by L1 data-availability; published ‘gas used’ reductions (Table 3) do not directly translate linearly to USD costs.

**Cost Breakdown:** `grantPermissionBySig` costs: signature verification ( $\sim 3,000$  gas for `ecrecover` precompile), storage writes ( $\sim 40,000$  gas for new permission),

Table 3: Smart Contract Gas Consumption

| Operation                     | Gas       |
|-------------------------------|-----------|
| <i>Ethereum Mainnet (L1)</i>  |           |
| Contract Deployment           | 2,341,829 |
| addRecord (first)             | 183,742   |
| addRecord (subsequent)        | 166,542   |
| grantPermissionBySig          | 78,331    |
| revokePermission              | 31,204    |
| updateRecord                  | 45,123    |
| emergencyGrantAccess          | 156,432   |
| confirmEmergencyAccess        | 35,211    |
| <i>Layer-2 (Arbitrum One)</i> |           |
| addRecord                     | 14,392    |
| grantPermissionBySig          | 6,127     |
| <i>Layer-2 (zkSync Era)</i>   |           |
| addRecord                     | 11,243    |
| grantPermissionBySig          | 5,894     |

Table 4: End-to-End Access Latency (1 MB Records)

| Component                        | Mean (ms)    | 95th % (ms)  |
|----------------------------------|--------------|--------------|
| Blockchain query                 | 245          | 312          |
| IPFS retrieval                   | 1,087        | 1,523        |
| Integrity verification (SHA-256) | 12           | 15           |
| Key unwrapping (ECIES)           | 4            | 5            |
| AES-GCM decryption               | 17           | 21           |
| <b>Total (IPFS)</b>              | <b>1,365</b> | <b>1,876</b> |
| S3 retrieval                     | 423          | 589          |
| <b>Total (S3)</b>                | <b>701</b>   | <b>942</b>   |

state updates ( $\sim 20,000$  gas), events ( $\sim 10,000$  gas), execution overhead ( $\sim 5,000$  gas).

**Economic Viability:** At \$3,000/ETH and 30 gwei gas price, permission grant costs  $\sim \$7$  on L1,  $\sim \$0.54$  on L2. Healthcare institutions or insurance can subsidize L1 costs; L2 enables direct patient payment models with substantially lower costs.

### 7.4 End-to-End Access Latency

Table 4 shows total record access time.

Storage retrieval dominates latency. IPFS exhibits higher variance due to distributed nature—retrieval from distant/slow peers. S3 provides consistent performance through CDN. Cryptographic operations contribute  $\leq 5\%$  total latency. The abstract reports mean values (0.7–1.4 s) for typical performance expectations.

**Scalability:** For 10 MB records, total latency increases to  $\sim 3.5$  s (IPFS) or  $\sim 2.1$  s (S3). Encryption/decryption scale linearly but remain small fraction. For typical clinical documents (10–500 KB), latency stays  $< 1$  s.

## 7.5 Storage Overhead

Encryption overhead: AES-GCM adds 12 bytes (nonce) + 16 bytes (tag) = 28 bytes per record. For 1 MB plaintext: overhead 0.003%. Content-addressing (IPFS CID) adds 32-byte identifier. Total storage penalty negligible.

On-chain storage per record: 32 bytes (digest) + 50 bytes (storage pointer string) + 110-150 bytes (wrapped key, implementation-dependent with ECIES ephemeral public key + IV + MAC + ciphertext)  $\approx$  192-232 bytes. Persistent storage on Ethereum is charged per 32-byte storage slot (20,000 gas for first-write). Dynamic types (strings/bytes) span multiple slots. In practice, `addRecord` costs  $\sim$ 166-184k gas on L1 (Table 3), dominated by storage writes and event emission; this measured figure is more representative than per-byte estimates.

## 8 Privacy and Regulatory Compliance

### 8.1 Metadata Privacy

**Public Information:** Storage pointers `ptr` and content digests `d` are public on-chain, visible in events and contract storage. This transparency is *by design*—pointers reference exclusively *encrypted* content. Without wrapped keys, adversaries obtain only ciphertext.

**Metadata Leakage:** On-chain data reveals: (1) Which addresses participate in health data sharing; (2) How many records each patient has; (3) When records are created/accessed; (4) Which recipients have permissions.

**Mitigation:** Recipients concerned about linkability should use fresh addresses per patient via HD wallet derivation (BIP-32/BIP-44). Patients can use mixing services or privacy-preserving layer-2s (zkSync) to obscure transaction origins. However, complete metadata privacy contradicts auditability—trade-off between transparency and privacy must be balanced per deployment requirements.

### 8.2 HIPAA Compliance

Health Insurance Portability and Accountability Act (HIPAA) mandates safeguards for protected health information (PHI). Our system addresses HIPAA requirements:

**Administrative Safeguards:** Patients define access policies through cryptographic permissions. Audit logs track all authorization events. Risk analysis identifies vulnerabilities (key management, storage availability).

**Physical Safeguards:** Encrypted storage prevents PHI exposure during theft/breach. Hardware wallets protect private keys.

**Technical Safeguards:** Authentication (ECDSA signatures), encryption (AES-256), integrity (SHA-256), audit trails (blockchain events for authorization history), transmission security (TLS for off-chain communication).

**Access Control:** Identity-based access through permission grants. Patient controls who accesses what, when. Emergency access mechanisms (Section 6.8) balance safety with privacy.

**Audit Logs (§164.312(b)):** Blockchain events create tamper-proof audit trail of all authorization events. The system provides complete authorization history (who was granted access) though not access history (who actually retrieved records) unless optional `logAccess` is used.

### 8.3 GDPR Compliance

General Data Protection Regulation (GDPR) grants data subjects rights over personal data. Our architecture supports GDPR requirements:

**Right to Access:** Patients always have permission to their records via `getOwnerWrappedKey`. Can query blockchain for complete metadata and retrieve encrypted data anytime.

**Right to Portability:** FHIR format enables standards-based data export. Patients can download encrypted records, decrypt with their keys, and transfer to other systems.

**Right to Erasure:** Complex due to blockchain immutability. We implement a three-tier approach:

1. Off-chain data can be deleted from storage
2. On-chain pointers can be nullified through contract updates
3. Permission/audit events remain as legally-required logs

Organizations should establish legal basis treating blockchain entries as audit logs potentially exempt from erasure under regulatory compliance requirements (GDPR Article 17(3)(b)).

**Data Minimization:** Only encrypted pointers and hashes stored on-chain. No protected health information appears in blockchain. Metadata in associated data minimized to version identifiers.

**Consent Management:** EIP-712 signatures provide explicit, informed, unambiguous consent for each data sharing event with cryptographic proof.

### 8.4 De-Identified Data for Research

Healthcare research requires large datasets while protecting privacy. We support de-identification pipelines creating research-safe datasets.

**De-Identification Methods:** We implement Safe Harbor method (HIPAA) removing 18 identifier categories: names, addresses (except ZIP regions), dates (except year), phone numbers, emails, SSNs, medical record numbers, account numbers, certificate numbers, vehicle identifiers, device identifiers, URLs, IP addresses, biometric identifiers, full-face photos, and other unique identifiers.

**Pipeline Architecture:** Patients select records for research contribution. De-identification service: (1) Decrypts records using patient-authorized temporary key; (2) Applies transformations (identifier removal, quasi-identifier generalization, date shifting); (3) Re-encrypts with research consortium key; (4) Uploads to research database with new permissions.

**Performance:** Evaluation on 1,000 Synthea FHIR bundles (mean 127 KB): de-identification takes mean 47.3 ms/record (95th: 68.5 ms). Processing 1M records requires  $\sim$ 13 hours sequential, parallelizable to <1 hour with 32-core cluster.

**Re-Identification Risk:** Even de-identified data carries re-identification risk through quasi-identifiers. We apply  $k$ -anonymity ( $k \geq 5$ ) [25] and evaluate with ARX Data Anonymization Tool. Results show:

- Maximum risk: 0.18% (highest risk individual)
- Average risk: 0.04% (across all records)
- Population uniqueness: 0.09%

These metrics meet standard thresholds for de-identified data release while maintaining research utility.

## 9 Discussion and Future Work

### 9.1 Deployment Barriers

**Integration Challenges:** Most EHR systems don't natively support blockchain. Middleware or API gateways can bridge, but introduce operational dependencies. Standards development and vendor cooperation are essential.

**User Experience:** Non-technical patients need intuitive interfaces hiding cryptographic complexity. Wallet software must balance security (key protection) with usability (avoiding lock-out). Social recovery mechanisms—where trusted contacts help restore access—show promise but need careful design to avoid vulnerabilities.

**Economic Models:** Gas costs require sustainable funding. Options: (1) Healthcare institutions subsidize as infrastructure cost; (2) Insurance covers as benefit; (3) Patients pay directly (raises equity concerns); (4) Tiered models with basic access funded by institutions. Contract deployment costs ( $\sim$ 2.3M gas on L1,  $\sim$ \$200 at typical prices) can be amortized over patient lifetime.

### 9.2 Institutional Key Management

Per-recipient key wrapping becomes impractical for large institutions (hospitals with hundreds of staff). We propose **institutional guardian keys**:

**Design:** Organizations represented by single encryption key managed institutionally. Patient wraps  $SymmK$  once for hospital's key. Hospital maintains internal access control determining which staff can decrypt. Reduces on-chain operations from  $O(n)$  per institution (where  $n = \text{staff count}$ ) to  $O(1)$ .

**Implementation:** Institution deploys guardian contract: (1) Registers encryption public key; (2) Maintains staff roster; (3) Provides `requestDecryption(rid, patientContract)` for staff; (4) Verifies caller is authorized staff; (5) Uses institutional private key (in HSM) to unwrap  $SymmK$  and re-wrap for requesting clinician.

**Trade-offs:** Trades patient-controlled fine-grained access for scalability/usability. Patients trust institution to enforce internal policies rather than controlling individual clinicians directly. Matches real-world practice: patients authorize "my hospital" rather than enumerating every provider. On-chain audit still records institutional-level grants/revocations.

### 9.3 Advanced Cryptographic Enhancements

**Proxy Re-Encryption:** Allows patients to delegate re-encryption to semi-trusted proxy, enabling efficient re-sharing without patient remaining online or re-encrypting. Introduces operational complexity but could improve usability for frequent re-sharing.

**Attribute-Based Encryption:** Encodes access policies directly in ciphertexts, automatically enforcing conditions like "any cardiologist in my hospital." Introduces computational overhead and complex key management.

**Zero-Knowledge Proofs:** Enables proving permission validity without revealing access control policy. Provider could prove they have valid access without disclosing which permission grant they use, obscuring metadata about sharing patterns.

### 9.4 Cross-Chain Interoperability

Healthcare is global involving diverse stakeholders potentially operating on different blockchains. Cross-chain interoperability protocols could enable permission grants on one blockchain recognized on another, or aggregate records across multiple blockchains.

Polkadot [26] provides heterogeneous multi-chain framework through relay chains and parachains. Cosmos offers Inter-Blockchain Communication for sovereign blockchain interoperation. Atomic swaps or blockchain bridges could enable interoperability. However, each approach introduces complexity and trust assumptions requiring careful evaluation for sensitive medical data.

## 9.5 Formal Verification

Smart contracts manage safety-critical assets (health data). Bugs have severe consequences. Formal verification tools (Certrora, K Framework, Coq) can mathematically prove contracts satisfy specified properties, providing stronger assurance than testing. Future work includes formal verification of our contract against confidentiality, integrity, and authorization properties.

## 9.6 Long-Term Data Stewardship

Digital estate planning mechanisms could allow patients to designate heirs or archival repositories for records. Medical research could benefit from posthumous data donation through advance directives recorded on-chain. Decentralized autonomous organizations (DAOs) could distribute governance across stakeholders with on-chain voting on protocol upgrades.

## 10 Limitations

**Key Compromise:** If patient’s  $SK_P$  is stolen, adversary can sign arbitrary authorizations. Multi-signature wallets, hardware wallets, and social recovery reduce risk but add complexity.

**Revocation Limits:** Revocation prevents future access but cannot retract already-decrypted plain-text—fundamental limitation of cryptographic access control.

**Storage Availability:** Depends on off-chain infrastructure. Storage provider failure makes records inaccessible until restored. Redundant storage mitigates but increases cost.

**Transaction Costs:** Gas prices fluctuate. High congestion makes L1 expensive. L2 solutions reduce costs but introduce additional trust assumptions (optimistic rollup fraud proofs, zkRollup trusted setup).

**Scalability:** Per-recipient key wrapping doesn’t scale to very large recipient lists (e.g., sharing with 1,000+ researchers). Institutional guardian keys and proxy re-encryption address some cases, but massive-scale sharing requires additional techniques.

**Metadata Privacy:** On-chain transparency reveals sharing patterns. While not exposing PHI, metadata can enable inference attacks (frequency analysis, timing correlation). Complete metadata privacy contradicts auditability—fundamental trade-off requiring deployment-specific balance.

**Denial of Service:** While fees deter spam, coordinated attacks or network congestion can delay critical healthcare operations. Multi-chain deployment with automatic failover provides resilience.

## 11 Conclusion

We presented a patient-centric blockchain architecture for health record management that cryptographically enforces access control while maintaining practical performance. By separating encrypted storage from on-chain authorization and deploying one contract per patient, we achieve confidentiality against curious storage providers, tamper-evident audit trails, and true patient sovereignty over medical data.

Our Ethereum implementation demonstrates feasibility with reasonable gas costs (78,000 gas per permission grant on L1, 6,000 gas on L2 though DA charges dominate actual costs) and acceptable latency (0.7–1.4 s mean end-to-end for 1 MB records on S3 and IPFS respectively). Security analysis establishes that standard cryptographic primitives composed correctly provide desired properties. The system provides complete authorization history (who was granted access), not complete access history (who actually retrieved/viewed records) unless optional read receipts via `logAccess` are used.

Critical implementation details—explicit nonce management for parallel grants, owner key retrieval paths, metadata privacy through minimal associated data, and comprehensive update/emergency access patterns—ensure the system is deployable in real clinical settings. Integration with FHIR standards and compliance mapping to HIPAA/GDPR requirements demonstrate regulatory viability.

The architecture addresses key challenges in healthcare data management: eliminates trusted intermediaries, provides cryptographic rather than policy-based access control, creates immutable audit trails, and restores patient agency over sensitive medical information. While not solving all healthcare IT problems (key management, emergency access, metadata privacy require ongoing research), this work establishes a foundation for truly patient-controlled health information exchange.

Future work includes formal verification of smart contracts, enhanced privacy through zero-knowledge proofs, cross-chain interoperability for global health data networks, and user studies evaluating real-world adoption barriers. The path forward requires collaboration among cryptographers, healthcare informaticists, policymakers, and patient advocates to realize the vision of patient-empowered, secure, and interoperable health data infrastructure.

## Acknowledgments

This work was supported by the Department of Computer Science & Engineering at Bangladesh University of Engineering & Technology. We thank the Ethereum Foundation and IPFS community for open-source tools enabling this research.

## References

- [1] J. R. Vest and L. D. Gamm, "Health information exchange: Persistent challenges and new strategies," *J. Am. Med. Inform. Assoc.*, vol. 17, no. 3, pp. 288–294, May 2010.
- [2] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "MedRec: Using blockchain for medical data access and permission management," in *Proc. IEEE OBD*, Aug. 2016, pp. 25–30.
- [3] A. Ekblaw, A. Azaria, J. D. Halamka, and A. Lippman, "A case study for blockchain in healthcare: MedRec prototype," in *Proc. IEEE OBD*, Aug. 2016.
- [4] K. Peterson, R. Deeduvanu, P. Kanjamala, and K. Boles, "A blockchain-based approach to health information exchange networks," in *Proc. NIST Workshop Blockchain Healthcare*, 2016.
- [5] X. Yue, H. Wang, D. Jin, M. Li, and W. Jiang, "Healthcare data gateways: Found healthcare intelligence on blockchain," *J. Med. Syst.*, vol. 40, no. 10, p. 218, Oct. 2016.
- [6] K. N. Griggs et al., "Healthcare blockchain system using smart contracts," *J. Med. Syst.*, vol. 42, no. 7, p. 130, June 2018.
- [7] K. M. Hasib et al., "Blockchain-based electronic health records management: A comprehensive review," *IEEE Access*, vol. 10, pp. 11411–11434, 2022.
- [8] C. C. Agbo, Q. H. Mahmoud, and J. M. Eklund, "Blockchain technology in healthcare: A systematic review," *Healthcare*, vol. 7, no. 2, p. 56, Apr. 2019.
- [9] T.-T. Kuo, H.-E. Kim, and L. Ohno-Machado, "Blockchain distributed ledger technologies for biomedical and health care applications," *J. Am. Med. Inform. Assoc.*, vol. 24, no. 6, pp. 1211–1220, Nov. 2017.
- [10] J. Benaloh, M. Chase, E. Horvitz, and K. Lauter, "Patient controlled encryption: Ensuring privacy of electronic medical records," in *Proc. ACM CCSW*, Nov. 2009, pp. 103–114.
- [11] B. Fisher et al., "PGP-HCCA: Pretty good privacy for health care client applications," in *Proc. IEEE ICHI*, Sept. 2013.
- [12] S. Wang, Y. Zhang, and Y. Zhang, "A blockchain-based framework for data sharing with fine-grained access control," *IEEE Access*, vol. 6, pp. 38437–38450, 2018.
- [13] A. A. Omar et al., "Privacy-friendly platform for healthcare data in cloud based on blockchain," *Future Gener. Comput. Syst.*, vol. 95, pp. 511–521, June 2019.
- [14] P. Zhang, J. White, D. C. Schmidt, G. Lenz, and S. T. Rosenbloom, "FHIRChain: Applying blockchain to securely share clinical data," *Comput. Struct. Biotechnol. J.*, vol. 16, pp. 267–278, 2018.
- [15] P. Zhang and D. C. Schmidt, "Blockchain for health data and its potential use in health IT and health care," *JAMA*, vol. 319, no. 23, pp. 2363–2364, June 2018.
- [16] D. C. Nguyen et al., "ACTION-EHR: Patient-centric blockchain-based electronic health record data management," *J. Netw. Comput. Appl.*, vol. 178, p. 102987, Mar. 2021.
- [17] NIST, "Advanced Encryption Standard (AES)," FIPS PUB 197, Nov. 2001.
- [18] M. Dworkin, "Recommendation for block cipher modes: GCM and GMAC," NIST SP 800-38D, Nov. 2007.
- [19] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis," in *Proc. ASIACRYPT*, Dec. 2000, pp. 531–545.
- [20] M. Abdalla, M. Bellare, and P. Rogaway, "The oracle Diffie-Hellman assumptions and DHIES," in *Proc. CT-RSA*, Apr. 2001, pp. 143–158.
- [21] V. Shoup, "A proposal for an ISO standard for public key encryption," ISO/IEC JTC 1/SC27, Dec. 2001.
- [22] R. Schiff et al., "EIP-712: Typed structured data hashing and signing," Ethereum Improvement Proposal, Sept. 2017.
- [23] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Yellow Paper, 2014.
- [24] J. Benet, "IPFS - content addressed, versioned, P2P file system," *arXiv:1407.3561*, July 2014.
- [25] L. Sweeney, "k-anonymity: A model for protecting privacy," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 10, no. 5, pp. 557–570, Oct. 2002.
- [26] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," White Paper, 2016.