

# HLFCPIR: Embedding Computational Private Information Retrieval into Hyperledger Fabric Chaincode for Query Privacy

First A. Author<sup>1</sup>, Fellow, IEEE, Second B. Author<sup>2</sup>,  
and Third C. Author Jr.<sup>3</sup>, Member, IEEE

<sup>1</sup>National Institute of Standards and Technology, Boulder, CO 80305 USA

<sup>2</sup>Department of Physics, Colorado State University, Fort Collins, CO 80523 USA

<sup>3</sup>Electrical Engineering Department, University of Colorado, Boulder, CO 80309 USA

Corresponding author: First A. Author (email: author@boulder.nist.gov).

This paragraph of the first footnote will contain support information, including sponsor and financial support acknowledgment. For example, "This work was supported in part by the U.S. Department of Commerce under Grant 123456."

**ABSTRACT** Permissioned blockchains ensure integrity and auditability of shared data but expose query parameters to endorsing peers during read operations. In Hyperledger Fabric, evaluate calls are executed by peers who observe function arguments and read-sets, creating privacy risks for organizations querying sensitive records. We address this gap by presenting the first practical integration of Computational Private Information Retrieval (CPIR) into Fabric chaincode. Our design encodes the ledger's key-value table as a plaintext polynomial and allows clients to submit encrypted selection vectors, evaluated under the Brakerski–Gentry–Vaikuntanathan (BGV) homomorphic encryption scheme. Peers return only encrypted responses, preventing index leakage while preserving normal Fabric endorsement and audit flows. We prototype the system with the Lattigo library and benchmark client-side encryption/decryption, peer-side evaluation, ciphertext size, and end-to-end query latency. Results show that single-query latencies remain practical for typical Fabric deployments, while eliminating the privacy leakage of baseline operations. This work demonstrates the feasibility of embedding CPIR directly into permissioned blockchains and provides a foundation for future enhancements such as post-quantum schemes, zero-knowledge proofs, and sublinear retrieval.

**INDEX TERMS** Private Information Retrieval (PIR), Homomorphic Encryption, Hyperledger Fabric, Permissioned Blockchains, Private Reads, Query Privacy.

## I. INTRODUCTION

### A. MOTIVATION AND PROBLEM STATEMENT

Permissioned blockchains such as Hyperledger Fabric are widely adopted for tamper-evident and auditable data management across consortiums. Their guarantees, however, primarily cover writes. In Fabric, the separation of *evaluate* and *submit* makes the read-privacy gap explicit: an *evaluate* call is a read-only proposal sent to endorsing peers, which execute the chaincode and return results without committing to the ledger. Crucially, these peers still observe all function arguments and read-sets. Thus, in multi-organization settings, the dominant privacy risk arises not from the

immutable ledger, but from endorsing peers who can log or infer sensitive query information.

This motivates us to target *query* or *read* privacy issue in Fabric, defined as the ability to hide which record is being queried from endorsing peers during *evaluate* type calls. Given problem description, an obvious approach is to construct and use Private Information Retrieval (PIR) [1] protocol, which would enable client to retrieve an item from a world state database without revealing to peer which item was requested. There are two main types of PIR schemes:

Earlier Information-Theoretic (IT-PIR) schemes [1] provide unconditional privacy by distributing the database

across multiple non-colluding servers, a client then queries subsets of servers such that no single server learns the selection index.

Computational PIR (CPIR) schemes relying on hardness assumptions and homomorphic encryption achieve privacy with a single server. Model the database as a vector  $D = \{d_0, \dots, d_{n-1}\}$ . To retrieve desired record  $d_i$  without revealing  $i$ , the client forms a one-hot selection vector  $\hat{v}_i$  defined as  $\hat{v}_i = (0, \dots, 0, 1, 0, \dots, 0)$  where 1 corresponds to the desired record at index  $i$ . He then encrypts it as  $ct_q = \text{Enc}_{pk}(\hat{v}_i)$  under a public key  $pk$  and sends  $ct_q$  to the server, which computes  $ct_r = (ct_q \cdot D) = \text{Enc}_{pk}(d_i)$ , returning  $ct_r$  to the client for decryption  $d_i = \text{Dec}_{sk}(ct_r)$  using his secret key  $sk$ , thus obtaining the desired record  $d_i$  without revealing  $i$  to the server, or peer in our case.

However, consortium settings exclude IT-PIR due to multiple non-colluding servers requirement, since all peers are typically considered honest-but-curious, and collusion is a realistic threat.

## B. LITERATURE REVIEW

We now examine existing attempts to address mentioned problem and identify the gap that our work fills.

**Blockchain Privacy Mechanisms.** Early efforts have emphasized access control and anonymity. Token-based authentication schemes [?] and access-control contracts [?] prevent unauthorized reads or hide participant identities. Differential-sharing frameworks [?] allow producers to regulate how much content is revealed. While effective at controlling *who* sees data, these mechanisms do not conceal *which* records are queried. Queries themselves remain visible to endorsing peers.

**PIR and FHE Applications.** A line of work has applied Private Information Retrieval (PIR) or Fully Homomorphic Encryption (FHE) to protect data access. Tan et al. [?] use CPIR to hide vehicular location queries. Chakraborty et al. [?] propose BRON, combining PIR with zero-knowledge proofs for human-resource data. Mazmudar et al. [?] integrate PIR with IPFS for private queries in distributed file sharing, while Hameed et al. [?] present DEBPIR, embedding an Oblivious Transfer-based PIR into Fabric smart contracts. These works demonstrate the feasibility of PIR in distributed settings but rely on extra cryptographic layers, multiple servers or off-chain components.

**Targeted gap.** To our knowledge, no prior work provided a systematic framework for embedding CPIR directly into Fabric chaincode nor has directly integrated a lattice-based CPIR scheme into Fabric chaincode to hide query indices from endorsing peers. We believe that this gap persists partly due to a misconception: blockchain and PIR are often seen as serving opposite functions —auditability versus privacy— when in fact they can be complementary.

This statement is backed by a recently published Privacy-Preserving Roadmap by PSE team in Ethereum foundation, which highlights private reads as a key track alongside pri-

vate writes and private proving for Ethereum’s development [?].

Combining Fabric with CPIR thus yields a dual guarantee: (1) writes remain auditable and attributable; (2) reads become unlinkable to specific keys—even to endorsing peers.

## C. KEY OBJECTIVE AND CONTRIBUTIONS

In this work, we demonstrate that CPIR can be implemented natively in chaincode to provide query privacy for Fabric reads and achieve practical performance under certain parametrization choices. For our consortium setting we choose a typical Cyber Threat Intelligence (CTI) sharing scenario often implemented on Fabric [?], where organizations exchange JSON-formatted threat indicators (hashes, IPs, metadata) and wish to keep their queries confidential.

Our work closes the aforementioned gap by embedding a BGV-based PIR workflow directly in Fabric’s evaluate path, ensuring that endorsing peers cannot infer queried indices while preserving normal endorsement and auditability.

Hence, the main contributions of this work are:

- 1) **BGV-based CPIR as Fabric chaincode.** We present, to the best of our knowledge, the first fully functional integration of a lattice-based CPIR scheme into Fabric chaincode for query privacy.
- 2) **Feasibility constraints.** We derive necessary and sufficient conditions on the cryptographic parameters, database size, and record structure to ensure that polynomial packing is possible, facilitating practical deployment for CTI sharing and similar use cases.
- 3) **Multi-channel architecture.** We design a multi-channel Fabric architecture where each channel maintains an independent polynomial database and metadata, allowing different organizations to host separate datasets while reusing the same CPIR chaincode.
- 4) **Comprehensive evaluation.** We implement a prototype using the Lattigo library and benchmark cryptographic operations, blockchain interactions, and end-to-end query latency under various configurations. Our results show that single-query latencies remain practical for typical Fabric deployments.
- 5) **Open-Source Release.** To encourage reproducibility and use by other researchers, we open-source the complete CPIR-on-Blockchain system, including chaincode, client, and experimental setup. The repository is available at: [https://github.com/artias13/2\\_2\\_HLF\\_CPIR](https://github.com/artias13/2_2_HLF_CPIR).

## D. ORGANIZATION

The remainder of this paper is organized as follows: Section II reviews Fabric privacy features, PIR fundamentals, and BGV encryption. Section III describes our system and threat models, polynomial database construction, feasibility constraints, multi-channel architecture, workflow and algorithms. Section IV shows experimental setup, cryptographic and blockchain benchmarks, and overall system perfor-

mance. Section V discusses limitations and future directions, and Section VI concludes the paper.

## II. PRELIMINARIES

### A. FABRIC NATIVE PRIVACY FEATURES

We hereby acknowledge several native solutions for privacy that Hyperledger Fabric provides and explain why they do not fully address the query privacy problem.

**Separate Channels.** Multi-channel partitioning isolates ledgers across subgroups of organizations, limiting which participants observe which data. However, channel separation controls *who* sees a ledger, not *what* is accessed inside that ledger. Query intent remains visible to all endorsers of a channel.

**Private Data Collections (PDC).** PDCs restrict which organizations store and access private key–value pairs. The shared ledger records only hashes, while members of the collection hold plaintext. PDCs provide access control but still expose function arguments to endorsers inside the collection, leaving query patterns observable.

**Fabric Private Chaincode (FPC).** FPC executes chaincode within Intel SGX enclaves. Arguments and state are protected even from peer operators, but this requires Trusted Execution Environments (TEEs) and attestation, introducing additional hardware and trust assumptions.

### B. FABRIC STORAGE LIMITS

We here address the question of how large a database can be stored in Fabric world state and ledger history, as this impacts the practical limits of our PIR construction.

**World state (LevelDB/CouchDB).** Fabric imposes no hard limit on the number of key–value entries in world state; capacity depends on available disk space and peer I/O throughput. In our implementation, each channel maintains a few small keys: "m\_DB" (64–265 KB), "n", "record\_s", "bgv\_params", and optional "record%03d" JSON records. These sizes are well within the default LevelDB storage profile, which is optimized for high-speed local key–value operations. Although CouchDB can be used as an alternative (for rich JSON queries), it enforces a configurable `max_document_size` limit of 8 MB by default (up to 4 GB in recent versions of CouchDB 3.0+). Since our  $m_{DB}$  values are below 300 KB, LevelDB is sufficient and preferable for performance and simplicity, while CouchDB remains compatible for future extensions that require JSON-based indexing.

**Ledger history (blockchain log).** Block size is constrained by the ordering service configuration. By default, the Fabric orderer limits the serialized payload to `AbsoluteMaxBytes` = 10 MB (recommended under 49 MB given the gRPC ceiling of 100 MB), and typically aggregates up to `MaxMessageCount` = 500 transactions per block or `PreferredMaxBytes` = 2 MB. In our system, these limits affect only *submit* transactions such as *InitLedger* or record updates. *Evaluate* transactions (including

*PIRQuery*) are read-only and do not generate blocks, thus unaffected by ordering or batching constraints.

**Implication.** The effective capacity of a channel is governed primarily by cryptographic feasibility (Fig. 3) and the size of a single world-state value (i.e.,  $m_{DB}$ ), rather than by Fabric's block or database limits. For large or binary CTI objects (e.g., malware samples or full PCAPs), an optional extension is to store them off-chain in IPFS while persisting only their content identifiers (CIDs) in world state, keeping the polynomial  $m_{DB}$  as the structured component used for private retrieval.

### C. PRIVATE INFORMATION RETRIEVAL (PIR)

**BGV Homomorphic Encryption:** Our construction relies on the Brakerski–Gentry–Vaikuntanathan (BGV) scheme, a lattice-based homomorphic encryption system supporting both addition and multiplication over ciphertexts. BGV enables *batching*, where multiple plaintext elements are packed into a single ciphertext. In our design, this batching is used to embed the ledger's key–value table into a single polynomial database representation, allowing efficient evaluation of structured PIR queries inside chaincode and underpins the polynomial database representation used in our Fabric integration.

The Brakerski–Gentry–Vaikuntanathan (BGV) scheme defines operations over two polynomial rings: a ciphertext ring  $R_Q = \mathbb{Z}_Q[X]/(X^N + 1)$  and a plaintext ring  $R_T = \mathbb{Z}_T[X]/(X^N + 1)$ , both sharing the same dimension  $N = 2^{\log N}$ . In our implementation, these rings are jointly specified by a single parameter literal  $(\log N, \log Q_i, \log P_i, T)$  as provided by the Lattigo library. The field  $T$  determines  $R_T$ , while the modulus chain  $(Q, P)$  and their bit-lengths  $(\log Q_i, \log P_i)$  determine  $R_Q$ .

**Our CPiR Instantiation:** We instantiate Computational PIR as a tuple of probabilistic polynomial-time algorithms (*KeyGen*, *Enc*, *Eval*, *Dec*):

- $\text{KeyGen}(\lambda) \rightarrow (pk, sk)$ : On input the security parameter  $\lambda$ , output a public key  $pk$  and a secret key  $sk$ .
- $\text{Enc}_{pk}(\hat{v}_i) \rightarrow ct_q$ : Given a windowed selection vector  $\hat{v}_i \in \{0, 1\}^N$ , encode it into the plaintext ring  $R_T$  and encrypt to a query ciphertext  $ct_q$  under  $pk$ .
- $\text{Eval}(ct_q, m_{DB}) \rightarrow ct_r$ : Given  $ct_q$  and the plaintext polynomial database  $m_{DB} \in R_T$ , homomorphically evaluate the product to obtain an encrypted response  $ct_r \in R_Q$ .
- $\text{Dec}_{sk}(ct_r) \rightarrow d_i$ : Using the secret key  $sk$ , decrypt the response ciphertext  $ct_r$  to recover the desired record  $d_i$ .

**Protocol objective.** Correctness requires that for all  $i \in [n]$ ,

$$\text{Dec}_{sk}(\text{Eval}(\text{Enc}_{pk}(\hat{v}_i), m_{DB})) = d_i.$$

*Remark (restricted operation set).* The full BGV scheme also provides *EvalKeyGen* to generate relinearization and rotation keys, supporting ciphertext–ciphertext multiplication, automorphisms, and modulus switching. Our PIR construction requires only ciphertext–plaintext multiplication ( $ct \times pt$ ),

since the database polynomial  $m_{DB}$  is kept in plaintext within world state. This avoids degree growth and level management, so we do not expose EvalKeyGen in chaincode. Extending to ciphertext–ciphertext evaluation would require additional on-chain artifacts (relinearization keys, Galois keys, encrypted  $m_{DB}$ ), larger world-state footprint, and higher evaluation cost, which we leave as future work.

#### D. NOTATION

We summarize the main notation used throughout the paper in Table 1.

### III. PROPOSED SYSTEM

#### A. SYSTEM MODEL

We introduce a blockchain-based query privacy system designed for permissioned ledgers. The system enables clients to privately retrieve from the ledger while endorsing peers can evaluate read-only queries over encrypted inputs without learning which record was accessed. The novelty of our approach lies in the integration of computational Private Information Retrieval (PIR) into Hyperledger Fabric chaincode using the Brakerski–Gentry–Vaikuntanathan (BGV) homomorphic encryption scheme. This approach ensures that clients remain the sole holders of decryption keys, while peers perform only black-box computations, thereby enhancing overall privacy without requiring trusted hardware or protocol modifications.

Our system is composed of the following entities:

- **Data Owner (DO):** Endorsing peers that hold the current plaintext polynomial  $m_{DB}$  in world state and execute PIR during *evaluate*. DO is honest-but-curious.
- **Data Writer (DW):** A client organization that provisions or refreshes the database. DW invokes *submit* to initialize the ledger (e.g., set  $n$  and template bounds). Chaincode computes  $record_s$ , packs  $D = \{d_0, \dots, d_{n-1}\}$ , encodes it into  $m_{DB}$ , and persists it.
- **Data Requester (DR):** A client that privately retrieves a record. DR runs  $KeyGen(\lambda) \rightarrow (pk, sk)$ , forms  $ct_q = Enc_{pk}(v_i)$ , calls *evaluate PIRQuery*, and later decrypts  $ct_r$ .
- **Gateway (GW):** The Fabric client/chaincode interface used by DW and DR to invoke *InitLedger*, *GetMetadata*, and *PIRQuery*. It follows standard Fabric semantics; no extra trust is assumed.

*Remark (world-state scope).* In Fabric, the “ledger” comprises the blockchain log and the world state. Our CPIR operates on the world state:  $m_{DB}$  encodes the latest key–value snapshot, not the historical transaction logs.

#### B. THREAT MODEL

Our design follows the standard *honest-but-curious* adversarial model. We explicitly consider the following assumptions and threats:

TABLE 1. Notation

Symbol	Description
$\lambda$	Security parameter
$n$	Database size; index domain $[n] = \{0, \dots, n-1\}$
$D = \{d_0, \dots, d_{n-1}\}$	Database records
$m_{DB}$	Plaintext polynomial representation of $D$
$\hat{v}_i$	One-hot selector for index $i$
$v_i$	Windowed selector for index $i$ with $record_s$ contiguous ones
$pk, sk$	Public / secret keys (BGV)
$ct_q = Enc_{pk}(\hat{v}_i)$	Encrypted query
$ct_r = Eval(ct_q, m_{DB})$	Encrypted response
$d_i = Dec_{sk}(ct_r)$	Decrypted record $d_i$ retrieved by client
$KeyGen(\lambda) \rightarrow (pk, sk)$	Key generation
$Enc_{pk}(\cdot), Dec_{sk}(\cdot)$	Encrypt / Decrypt
$Eval(\cdot)$	Homomorphic evaluation (ct–pt multiply)
$N = 2^{\log N}$	Ring dimension
$\log Q_i$	Bit-lengths of primes forming modulus chain $Q$
$\log P_i$	Bit-lengths of special primes $P$
$T$	Plaintext modulus
$record_s$	Slots allocated per record
$record_b$	Base serialized size of a record in bytes
$record_{\mu, \log N}$	Template-specific minimum record size at security level $\log N$
$\mathcal{S}$	Allowed discrete slot sizes
$c = (c_0, \dots, c_{N-1})$	Coefficient vector of the polynomial encoding (slots of $m_{DB}$ )
$ \cdot , \text{size}(\cdot)$	Length in elements / size in bytes
$\text{Cap}(N, s, n)$	Capacity predicate: $n \cdot s \leq N$
$\text{Min}(\log N, s)$	Template predicate: $s \geq record_{\mu, \log N}$
$\text{Disc}(s)$	Discrete predicate: $s \in \mathcal{S}$
$\text{Cap} \wedge \text{Min} \wedge \text{Disc}$	Feasibility condition
$DO, DW, DR, GW$	Data Owner; Data Writer; Data Requester; Gateway
<i>evaluate, submit</i>	Fabric read / write transaction phases
$\mathcal{L}$	Leakage considered (ciphertext size, protocol timing)

- **Endorsing peers (DO).** Execute chaincode correctly but may try to infer the queried index from *evaluate* inputs or logs. They see  $ct_q$ , metadata, and  $m_{DB}$ .
- **Data Writer (DW).** Issues initialization writes via *submit*. DW is not trusted with decryption keys and

learns nothing about  $DR$ 's queries. We assume  $DW$  follows the write protocol but is not relied upon for privacy.

- **External observers.** May eavesdrop on client-peer traffic. Without  $sk$ ,  $ct_q$  and  $ct_r$  reveal nothing under BGV assumptions.
- **Out of scope.** Traffic analysis and timing side-channels; the only permitted leakage is  $\mathcal{L}$  (ciphertext size and protocol timing).

**Security objective.** For any  $i \in [n]$ , neither  $DO$  nor external observers can distinguish which  $d_i$  is requested from  $ct_q$  and  $ct_r$ . The only permissible leakage is ciphertext size and protocol timing, denoted collectively as  $\mathcal{L}$ .

### C. SYSTEM OVERVIEW

The proposed system integrates computational Private Information Retrieval (CPIR) directly into Hyperledger Fabric chaincode. Its purpose is to ensure that query indices remain hidden from endorsing peers while preserving Fabric's endorsement and audit workflow. At a high level, the workflow consists of four stages, illustrated in Fig. 1.

- 1) **Ledger initialization.**  $DW$  invokes *InitLedger* via  $GW$  using *submit*. Chaincode derives  $record_s$  from  $record_b$ , packs  $D$  into  $c = (c_0, \dots, c_{N-1})$ , encodes  $m_{DB}$ , and stores  $m_{DB}$  and metadata in world state held by  $DO$ .
- 2) **Metadata discovery.**  $DR$  calls *GetMetadata* via *evaluate* to obtain  $n$ ,  $record_s$ , and BGV parameters needed to form a valid query.
- 3) **Private retrieval.**  $DR$  constructs  $ct_q = Enc_{pk}(v_i)$  and invokes *PIRQuery* via *evaluate*.  $DO$  computes  $ct_r = Eval(ct_q, m_{DB})$  and returns it.
- 4) **Decryption.**  $DR$  decrypts  $ct_r$  to recover  $d_i = Dec_{sk}(ct_r)$ .

### D. POLYNOMIAL DATABASE CONSTRUCTION

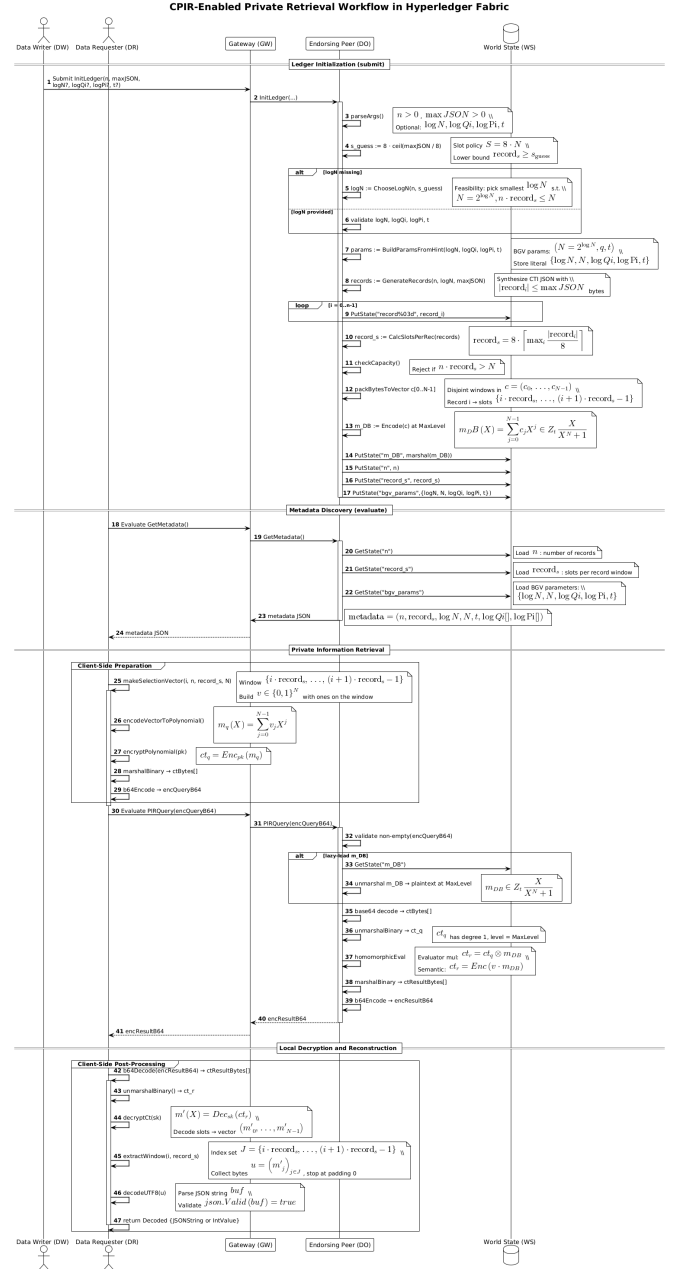
To enable PIR queries over structured ledger data, we must embed records into a plaintext polynomial  $m_{DB}$  suitable for BGV evaluation. Our prototype adopts a fixed-width packing strategy, illustrated in Fig. 2, which proceeds in four steps.

**Step 1: Serialize to Byte Array.** Each ledger record  $d_i$  is serialized as a UTF-8 byte array. In our motivating use case of Cyber Threat Intelligence (CTI) sharing, JSON objects containing fields such as hash digests and threat levels are flattened into byte sequences. Every character is represented by its ASCII code in  $[0, 255]$ . This ensures that arbitrary structured records can be embedded in the polynomial without loss of information.

**Step 2: Calculate Slot Window.** We determine the slot allocation per record as:

$$record_s = \left\lceil \frac{record_b}{bytesPerSlot} \right\rceil,$$

where  $record_b$  is the maximum serialized record length observed in bytes. In our prototype, each slot stores exactly

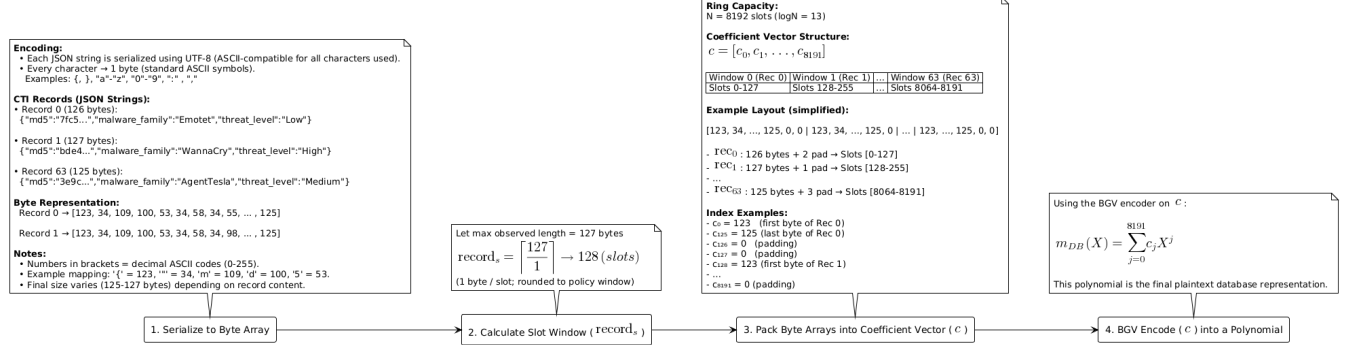


**FIGURE 1.** “Workflow.  $DW$  initializes the ledger via  $GW$ , which triggers chaincode on endorsing peers ( $DO$ ).  $DO$  executes the protocol and persists state in world state ( $m_{DB}$ , metadata, JSON records).  $DR$  later obtains metadata, submits  $ct_q = Enc_{pk}(v_i)$ ,  $DO$  evaluates  $ct_r = Eval(ct_q, m_{DB})$  against world state, and  $DR$  decrypts to  $d_i$ .

one byte. For example, if the largest record is 126 bytes, then  $record_s = \lceil 126/1 \rceil = 126$ , which rounds up to 128 slots due to the discrete window policy. This guarantees uniform slot windows across all records, simplifying query construction at the cost of potential padding overhead.

**Step 3: Pack into Coefficient Vector.** Serialized byte arrays are inserted into disjoint slot windows of length  $record_s$  within a coefficient vector  $c = (c_0, c_1, \dots, c_{N-1})$ ,

### Plaintext Polynomial Database Construction



**FIGURE 2.**  $m_{DB}$  construction from JSON to plaintext polynomial. Each record is serialized to bytes, mapped into a fixed slot window  $record_s$ , and packed into a coefficient vector  $c$ . The vector is then encoded as a BGV plaintext polynomial  $m_{DB}$ , which is stored in the Fabric world state.

where  $N = 2^{\log N}$  is the ring capacity of the BGV scheme. Padding zeros are added if a record is shorter than  $record_s$ . Thus each record  $d_i$  occupies a contiguous slot interval that can be privately retrieved through PIR.

**Step 4: Encode into Polynomial.** Finally, the coefficient vector  $d$  is encoded into a plaintext polynomial:

$$m_{DB}(X) = \sum_{j=0}^{N-1} c_j X^j \in R_t,$$

where  $R_t = \mathbb{Z}_t[X]/(X^N + 1)$ . This polynomial serves as the plaintext database representation stored in the Fabric world state. Endorsing peers operate over  $m_{DB}$  during PIR queries, while clients recover only the slots corresponding to their requested record.

### E. FEASIBILITY CONSTRAINTS

Embedding records into the plaintext polynomial  $m_{DB}$  is feasible only for parameter triples  $(\log N, n, record_s)$  that satisfy *all* of the following constraints. These constraints form a hierarchical relationship: template requirements dominate discrete allocation, and both are ultimately bounded by ring capacity.

**Constraint 1: Ring capacity.** The total number of occupied slots cannot exceed the ring size:

$$n \cdot record_s \leq N.$$

This represents the fundamental mathematical limit imposed by the cryptographic parameters. For example, with  $\log N = 13$  ( $N = 8192$ ) and  $record_s = 224$ , at most  $\lceil 8192/224 \rceil = 36$  records can be packed.

**Constraint 2: Template-specific minima.** Each security level ( $\log N$ ) corresponds to a record template with mandatory fields that impose a minimum slot requirement  $record_{\mu, \log N}$ . Examples include:

- **Mini records** ( $\log N = 13$ ): MD5 hash + malware family + threat level  $\Rightarrow record_{\mu,13} \approx 128$  bytes.

- **Mid records** ( $\log N = 14$ ): MD5 + SHA-256 short + malware class + AV detects  $\Rightarrow record_{\mu,14} \approx 224$  bytes.
- **Rich records** ( $\log N = 15$ ): MD5 + full SHA-256 + all metadata  $\Rightarrow record_{\mu,15} \approx 256$  bytes.

These minima arise from generator checks of the form:

$$\text{Mini: } record_s \geq record_b + 32 + 15,$$

Mid:  $record_s \geq record_b + 32 + 16 + 15,$

**Rich:**  $record_s \geq record_b + 32 + 64 + 15,$

where  $record_b \approx 113\text{--}125$  bytes denotes the base JSON structure and numeric terms represent mandatory hash fields and serialization overhead.

**Constraint 3: Discrete allocation policy.** Operationally, we restrict the slot window  $record_s$  to a discrete set for implementation simplicity:

$$\mathcal{S} = \{64, 128, 224, 256, 384, 512\} \text{ bytes.}$$

This means that even if Constraints 1 and 2 are satisfied, the configuration is rejected unless  $record_s \in \mathcal{S}$ .

**Constraint hierarchy and feasibility.** Feasibility of a configuration  $(\log N, n, record_s)$  is defined by three predicates:

$$\text{Cap}(N, s, n) : n \cdot s \leq N \quad (\text{ring capacity})$$

$$\text{Min}(\log N, s) : s \geq \text{record}_{\mu, \log N} \quad (\text{template minimum})$$

$$\text{Disc}(s) : s \in \mathcal{S} \quad (\text{discrete allocation}).$$

where  $s = record_s$  and  $N = 2^{\log N}$ . A configuration is feasible iff:

$$\text{Cap}(N, s, n) \wedge \text{Min}(\log N, s) \wedge \text{Disc}(s).$$

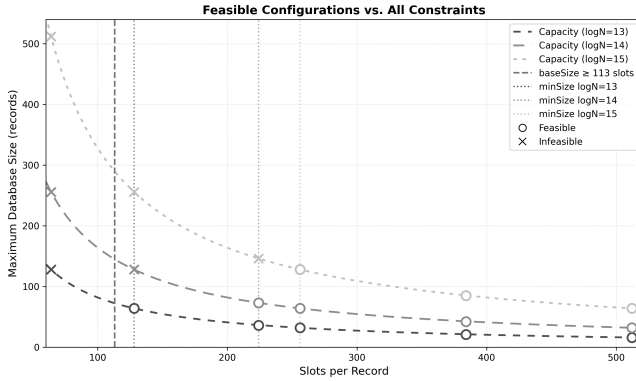
**Examples.** The interaction of these predicates is illustrated in Fig. 3. We provide three concrete examples below:

a) *Polynomial degree*  $\log N = 13$  (Mini):

- $\text{Min}(13, s)$  enforces  $s \geq \text{record}_{\mu, 13} \approx 128 \Rightarrow$  the smallest candidate is  $s = 128$ .
- $\text{Disc}(s)$  requires  $s \in \mathcal{S} \Rightarrow 128$  is allowed (while 64 is invalid).
- $\text{Cap}(8192, 128, n)$  gives  $n \leq \lfloor 8192/128 \rfloor = 64$ .

- *Feasible*: ( $\log N = 13, s = 128, n \leq 64$ ).
- b) *Polynomial degree*  $\log N = 14$  (Mid):
- $\text{Min}(14, s)$  enforces  $s \geq \text{record}_{\mu,14} \approx 224 \Rightarrow$  smallest candidate is 224.
  - $\text{Disc}(s)$  allows 224, 256, ... but excludes smaller windows.
  - $\text{Cap}(16384, 224, n)$  gives  $n \leq \lfloor 16384/224 \rfloor = 73$ .
  - *Feasible*: ( $\log N = 14, s \in \{224, 256, \dots\}, n \leq 73$ ).
- c) *Polynomial degree*  $\log N = 15$  (Rich):
- $\text{Min}(15, s)$  enforces  $s \geq \text{record}_{\mu,15} \approx 256$ .
  - $\text{Disc}(s)$  excludes 64, 128, 224; smallest valid is 256.
  - $\text{Cap}(32768, 256, n)$  gives  $n \leq \lfloor 32768/256 \rfloor = 128$ .
  - *Feasible*: ( $\log N = 15, s \geq 256, n \leq 128$ ).

**Implications.** Increasing  $\log N$  raises ring capacity  $N$  and thus  $n$ , but also requires larger  $\text{record}_s$  if given richer templates. Feasible configurations occur only where all 3 predicates are met, guiding practical parameter selection.



**FIGURE 3.** Feasible configurations under the joint constraints Cap, Min, and Disc. Dashed curves show ring-capacity limits for  $\log N \in \{13, 14, 15\}$ , vertical lines mark template-driven minima  $\text{record}_{\mu, \log N}$ , and x-axis ticks correspond to discrete slot sizes  $S$ . Circles indicate feasible triples  $(\log N, \text{record}_s, n)$ .

## F. MULTI-CHANNEL ARCHITECTURE

The packing strategy and feasibility constraints highlight an important observation: no single homomorphic parameter set can efficiently support the full diversity of Cyber Threat Intelligence (CTI) record formats. Compact records fit comfortably under smaller rings, while full JSON objects with long cryptographic hashes exceed the slot budget of these configurations. To balance scalability and expressiveness, we design a *multi-channel architecture* in Hyperledger Fabric (Fig. 4), where each channel is provisioned with a distinct BGV parameter set and record template.

a) **Channel Mini** ( $\log N = 13$ ). Supports compact CTI records (e.g., MD5, malware family, threat level) with maximum scalability and lowest query latency. For example, with  $N = 8192$  slots, the system accommodates up to 128

records when  $\max_i |d_i| \leq 64$  bytes, and 16 records when  $\max_i |d_i| \leq 512$  bytes.

b) **Channel Mid** ( $\log N = 14$ ). Targets medium-sized records that include MD5 and truncated SHA-256 fields alongside classification metadata. With  $N = 16384$  slots, the system supports up to 256 records at  $\leq 64$  bytes or 32 records at  $\leq 512$  bytes.

c) **Channel Rich** ( $\log N = 15$ ). Handles the most detailed records, including full-length hashes and multiple metadata fields. Here,  $N = 32768$  slots allow up to 512 records at  $\leq 64$  bytes or 64 records at  $\leq 512$  bytes.

**Channel semantics.** As shown in Fig. 4, each channel maintains its own PIR chaincode instance and world state. The world state contains:

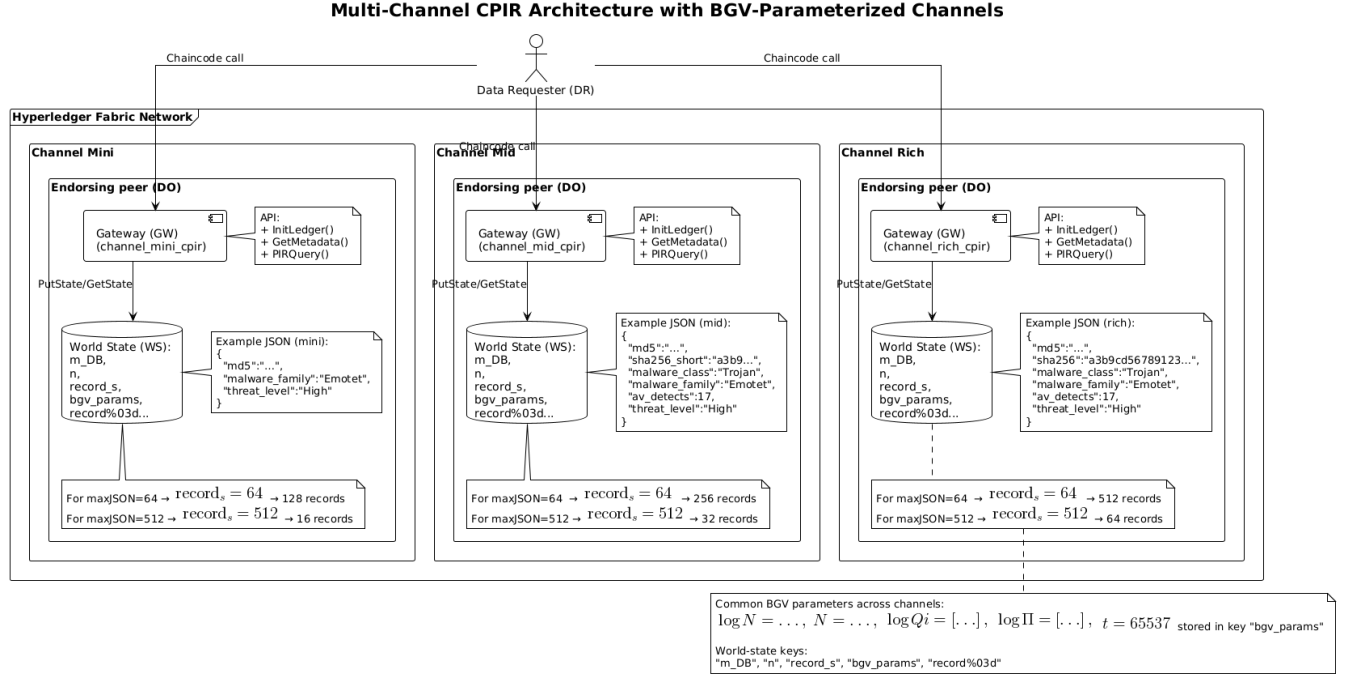
- The *polynomial view*: the packed plaintext polynomial  $m_{DB}$  under key "m\_DB".
- The *normal view*: JSON records stored individually under keys "record%03d" for auditability and interoperability with non-PIR chaincode.
- *Metadata*:
  - "n": number of records  $n$ ,
  - "record\_s": slots per record  $\text{record}_s$ ,
  - "bgv\_params":  $\{\log N, N, \log Q_i, \log P_i, T\}$ .

*Remark (ledger capacity).* A practical concern is the maximum size of the ledger when channels store both JSON records and the polynomial  $m_{DB}$ . In Hyperledger Fabric, two layers impose size-related limits: the *world state* (LevelDB or CouchDB) and the *blockchain history* managed by the ordering service, refer to Section II for details.

## G. WORKFLOW DETAILS

The complete workflow of our CPIR-enabled Fabric system is driven by five algorithms (Alg. 1–5), corresponding to the chaincode interface (*InitLedger*, *GetMetadata*, *PIRQuery*) and the client-side routines (*FormSelectionVector*, *DecryptResult*). Figure 1 provides the high-level overview. A Data Writer (DW) provisions the database  $D = \{d_0, \dots, d_{n-1}\}$ , a Data Owner (DO) maintains the polynomial  $m_{DB}$  in world state and executes PIR evaluations, and a Data Requester (DR) retrieves  $d_i$  privately using homomorphic encryption. The detailed steps are as follows:

- 1) **DW submits initialization.** DW calls *InitLedger* (Alg. 1) via  $\mathcal{GW}$  (*submit*) with inputs  $(n, \text{record}_s^{DW})$  and an optional hint  $(\log N, \log Q_i, \log P_i, T)$ . Here  $\text{record}_s^{DW}$  denotes the maximum JSON size anticipated by the writer.
- 2) **DO validates and derives parameters.** DO rounds the writer's input to a discrete slot size  $\text{record}_s^{GW} = 8 \cdot \lceil \text{record}_s^{DW} / 8 \rceil$ . If  $\log N$  is absent, the smallest feasible  $\log N$  is chosen such that  $\text{Cap}(N, \text{record}_s^{GW}, n)$  holds. BGV parameters  $\{\log N, N, \log Q_i, \log P_i, T\}$  are constructed and stored.



**FIGURE 4. Multi-channel CPIR architecture.** Each channel instantiates a separate CPIR chaincode and maintains its own  $m_{DB}$  polynomial, parameterized by  $\log N$ . This allows compact, mid-size, and rich CTI records to coexist under the same Fabric network.

- 3) **DO prepares records.** Records  $D = \{d_0, \dots, d_{n-1}\}$  are ingested or synthesized with  $|d_i| \leq \text{record}_s^{DW}$ . The definitive slot allocation is then fixed as  $\text{record}_s = 8 \cdot \lceil \max_i |d_i| / 8 \rceil$  (discrete policy), checked against feasibility predicates  $\text{Min}(\log N, \text{record}_s)$ ,  $\text{Disc}(\text{record}_s)$ ,  $\text{Cap}(N, \text{record}_s, n)$ .
- 4) **Pack and persist.** Each  $d_i$  is placed in a disjoint window  $J_i = \{i \cdot \text{record}_s, \dots, (i+1)\text{record}_s - 1\}$  of  $c = (c_0, \dots, c_{N-1})$ , zeros pad unused slots, and the polynomial  $m_{DB}(X) = \sum c_j X^j$  is encoded at max level. World state stores "m\_DB", "n", "record\_s", and "bgv\_params" plus optional "record%03d" entries.
- 5) **DR discovers metadata.** DR calls *GetMetadata* (Alg. 2) via *evaluate* to obtain  $(n, \text{record}_s, \log N, N, T, \log Qi, \log Pi)$ . This enables reconstruction of the cryptographic context.
- 6) **DR instantiates crypto context.** From metadata, DR builds parameters, executes  $\text{KeyGen}(\lambda) \rightarrow (pk, sk)$ , and prepares encoder/encryptor objects.
- 7) **Form and encrypt query.** For index  $i \in [n]$ , DR runs *FormSelectionVector* (Alg. 3): define  $J_i$ , set  $v_i$  with ones on  $J_i$ , encode to  $m_q(X)$ , encrypt as  $ct_q = \text{Enc}_{pk}(v_i)$ , and serialize/Base64-encode.
- 8) **PIR query evaluation.** DR issues  $\text{PIRQuery}(ct_q^{B64})$  (Alg. 4) via *evaluate*. DO decodes, reloads  $m_{DB}$  if necessary, and computes  $ct_r = \text{Eval}(ct_q, m_{DB})$ , returning the Base64-encoded ciphertext.

- 9) **Decryption and reconstruction.** DR runs *DecryptResult* (Alg. 5) to recover  $m'(X)$ , extract bytes from  $J_i$ , stop at padding zero, and reconstruct  $d_i$  as a valid JSON object (or a scalar if  $\text{record}_s = 1$ ).

*Remark (levels, noise, determinism).* Queries are encoded and encrypted at *max level*. Chaincode evaluates a single ct-pt multiply (no relinearization). This keeps noise growth minimal and evaluation deterministic across endorsers, which is important for Fabric endorsement. If  $m_{DB}$  is re-encoded or refreshed, GW still returns the same metadata blob; clients rebuild context idempotently.

## IV. PERFORMANCE EVALUATION

### A. EXPERIMENTAL SETUP

All experiments were executed on a local Ubuntu 24.04 host running under WSL2 on an Intel Core i5-3380M CPU (2 cores/4 threads, 2.90 GHz) with 7.7 GB of RAM and a 1 TB SSD. During evaluation, the average available memory was 6.9 GB with a 2 GB swap partition, and the root filesystem reported 946 GB of free space.

The software stack consisted of Go 1.24.1, Docker 27.4.0, and Docker Compose v2.31.0, hosting Hyperledger Fabric v2.5 with a single Raft orderer and LevelDB as the world-state database. Fabric's ordering service used the recommended parameters (BatchSize.AbsoluteMaxBytes=99 MB, PreferredMaxBytes=2 MB per block).

**Algorithm 1** InitLedger (chaincode)

---

**Require:**  $n$ ;  $record_s^{\mathcal{DW}}$ ;  $op$ : hint

- 1:  $\log N \leftarrow \text{selMinLogN}\left(\left(n, 8 \cdot \left\lceil \frac{record_s^{\mathcal{DW}}}{8} \right\rceil\right)\right)$
- 2: **if**  $\log N = \emptyset$  **then**
- 3:   **return**  $\perp$
- 4: **end if**
- 5:  $bgvParams \leftarrow \text{selParams}((\log N, op : \text{hint}))$
- 6:  $D \leftarrow \text{genRecords}((n, record_s^{\mathcal{DW}}))$
- 7: **if**  $\exists i : |d_i| > record_s^{\mathcal{DW}}$  **then**
- 8:   **return**  $\perp$
- 9: **end if**
- 10:  $record_s \leftarrow 8 \cdot \left\lceil \frac{\max_i |d_i|}{8} \right\rceil$
- 11: **if**  $\neg \text{feasible}(\log N, n, record_s)$  **then**
- 12:   **return**  $\perp$  // infeasible configuration
- 13: **end if**
- 14:  $c \leftarrow [0, \dots, 0] \in \mathbb{Z}_T^N$  // init coefficient vector
- 15: **for**  $i \in [0, n-1]$  **do**
- 16:    $J_i \leftarrow \{i \cdot record_s, \dots, (i+1) \cdot record_s - 1\}$  // slot window for  $d_i$
- 17:   **for**  $k = 0$  **to**  $record_s - 1$  **do**
- 18:     **if**  $k < |d_i|$  **then**
- 19:        $c[J_i[k]] \leftarrow \text{byte}(d_i[k])$  // copy byte of record
- 20:     **else**
- 21:        $c[J_i[k]] \leftarrow 0$  // padding
- 22:     **end if**
- 23:   **end for**
- 24: **end for**
- 25:  $m_{DB}(X) \leftarrow \text{enc}^{\text{poly}}(c) \in \mathbb{Z}_T[X]/(X^N + 1)$
- 26:  $worldState \leftarrow \{m_{DB}, n, record_s, bgvParams, op : D\}$
- 27: **return** OK

---

**Algorithm 2** GetMetadata (chaincode)

---

**Require:**  $\emptyset$

- 1:  $n \leftarrow worldState.n$
- 2:  $record_s \leftarrow worldState.record_s$
- 3:  $paramsMeta \leftarrow worldState.bgvParams$
- 4: **if**  $n = \emptyset \vee record_s = \emptyset \vee paramsMeta = \emptyset$  **then**
- 5:   **return**  $\perp$
- 6: **end if**
- 7:  $paramsMeta = (\log N, N, \log Q_i[], \log P_i[], T)$
- 8:  $metadata \leftarrow (n, record_s, paramsMeta)$
- 9: **return**  $metadata$

---

Our implementation employs the *Lattigo* v6 library [?] as the homomorphic encryption backend. *Lattigo* provides a Go-native implementation of the Brakerski–Gentry–Vaikuntanathan (BGV) scheme [?], whose security and correctness have been validated in prior literature. Accordingly, our focus is on evaluating its *practical performance within a permissioned blockchain environment*.

**Algorithm 3** FormSelectionVector (client)

---

**Require:**  $pk$ ;  $i \in [n]$ ;  $record_s$ ;  $N$

- 1: **if**  $i < 0 \vee i \geq n$  **then**
- 2:   **return**  $\perp$
- 3: **end if**
- 4: **if**  $n \cdot record_s > N$  **then**
- 5:   **return**  $\perp$
- 6: **end if**
- 7:  $J_i \leftarrow \{i \cdot record_s, \dots, (i+1) \cdot record_s - 1\}$
- 8:  $v_i \in \{0, 1\}^N \leftarrow \mathbf{0}$
- 9: **for**  $j \in J_i$  **do**
- 10:    $v_i[j] \leftarrow 1$
- 11: **end for** // windowed selector
- 12:  $m_q(X) \leftarrow \text{enc}^{\text{poly}}(v_i)$  // polynomial encode at max level
- 13:  $ct_q \leftarrow \text{Enc}_{pk}(m_q)$
- 14:  $ct_q^{B64} \leftarrow \text{enc}^{B64}(\text{ser}^{\text{bin}}(ct_q))$
- 15: **return**  $ct_q^{B64}$  // Base64(marshalled ciphertext)

---

**Algorithm 4** PIRQuery (chaincode)

---

**Require:**  $ct_q^{B64}$

- 1: **if**  $ct_q^{B64} = \emptyset$  **then**
- 2:   **return**  $\perp$
- 3: **end if**
- 4:  $ct_q \leftarrow \text{des}^{\text{bin}}(\text{dec}^{B64}((ct_q^{B64})))$
- 5: **if**  $m_{DB}$  not cached in memory **then**
- 6:    $m_{DB} \leftarrow worldState.m_{DB}$
- 7: **end if**
- 8:  $ct_r \leftarrow \text{Eval}(ct_q, m_{DB})$
- 9:  $ct_r^{B64} \leftarrow \text{enc}^{B64}(\text{ser}^{\text{bin}}((ct_r)))$
- 10: **return**  $ct_r^{B64}$

---

**Algorithm 5** DecryptResult (client)

---

**Require:**  $ct_r^{B64}$ ;  $sk$ ;  $i \in [n]$ ;  $record_s$ ;  $n$

- 1: **if**  $i < 0$  **or**  $i \geq n$  **then return**  $\perp$
- 2: **if**  $n \cdot record_s > N$  **then return**  $\perp$  // sanity
- 3:  $ct_r \leftarrow \text{des}^{\text{bin}}(\text{dec}^{B64}((ct_r^{B64})))$
- 4:  $u \in \mathbb{Z}_T^N \leftarrow \text{dec}^{\text{poly}}(m'(X)) \leftarrow \text{Dec}_{sk}(ct_r)$
- 5:  $J_i \leftarrow \{i \cdot record_s, \dots, (i+1) \cdot record_s - 1\}$
- 6:  $b \leftarrow$  byte array // init empty buffer for record
- 7: **for**  $j \in J_i$  **do**
- 8:   **if**  $u[j] = 0$  **then**
- 9:     **break**
- 10:   **end if** // stop at padding zero
- 11:    $b.append(u[j])$
- 12: **end for**
- 13: **if**  $record_s = 1$  **then return**  $u[i \cdot record_s]$
- 14:  $d_i \leftarrow \text{dec}^{UTF8}(b)$
- 15: **return**  $d_i$

---

Client–peer communication was performed through the Fabric Go SDK (Gateway API). The network configuration

comprised a single organization with one peer per channel, sufficient for privacy evaluation since PIR execution occurs solely at endorsing peers and ordering nodes do not access the world state. Unless otherwise stated, each reported value represents the mean of 20 executions under both cold and warm cache conditions, cross-verified against peer logs for consistency.

### B. CRYPTOGRAPHIC PERFORMANCE

**Parameter Configuration.** Table 2 summarizes the BGV parameter sets used in our evaluation, including the ring dimension  $N$ , ciphertext modulus chain  $(\log Q_i, \log P_i)$ , plaintext modulus  $T$ , slot allocation  $record_s$ , and database size  $n$ . Each configuration corresponds to one Fabric channel and maintains a feasible packing ratio as defined in Section III.

**TABLE 2.** Default BGV Parameter Configuration per Channel

$N$	$\log Q_i$	$\log P_i$	$T$	$record_s$	$n$
$2^{13}$	[54]	[54]	65537	128	64
$2^{14}$	[54]	[54]	65537	224	73
$2^{15}$	[54]	[54]	65537	256	128

**Overall Results.** Figure 5 consolidates the main cryptographic evaluation metrics: (A) end-to-end latency by algorithmic stage, (B) serialized artifact size, and (C) slot utilization ratio within the packed database polynomial  $m_{DB}$ .

(A) *End-to-end latency by stage.* Each bar shows the average latency per query at a given ring size  $N = 2^{\log N}$ , divided into four stages: *KeyGen* (parameter setup and keypair generation), *Enc* (selector encoding and encryption), *Eval* (homomorphic multiplication  $ct \times pt$ ), and *Dec* (decryption and record reconstruction). All measurements were averaged over 20 executions.

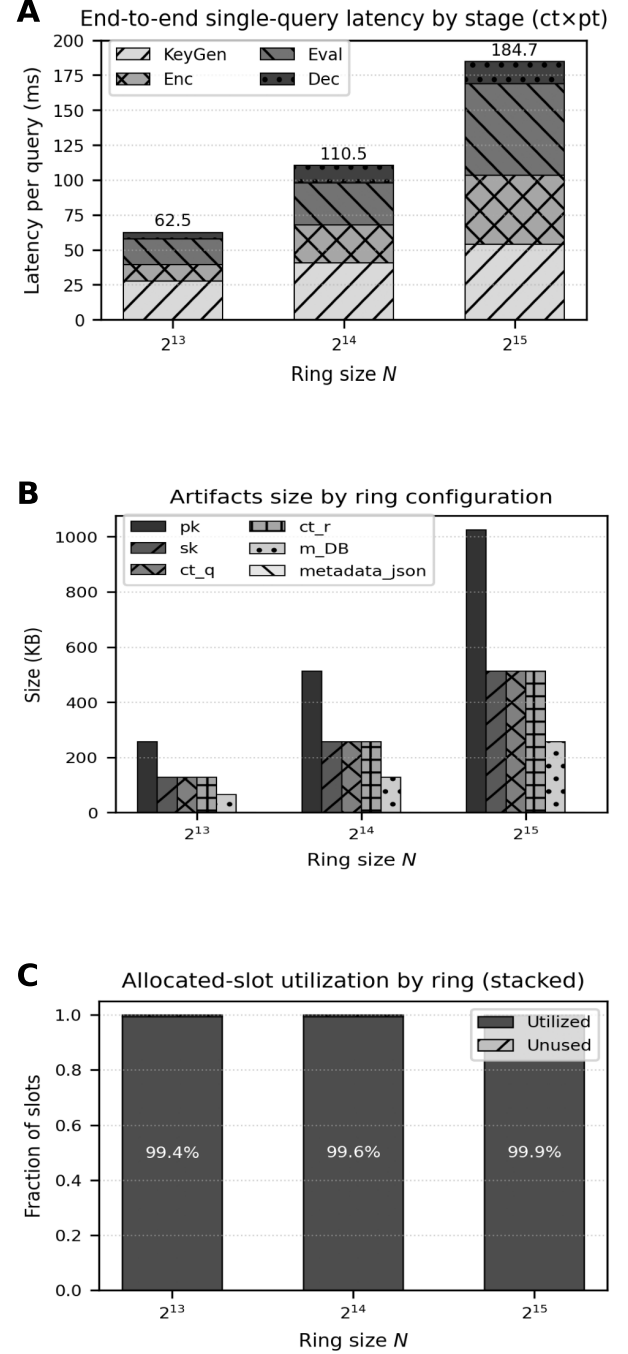
(B) *Artifact size.* This plot reports the serialized sizes (in KB) of core artifacts: the public key, secret key, encrypted selector  $ct_q$ , encrypted response  $ct_r$ , plaintext database polynomial  $m_{DB}$ , and the metadata JSON object. Base64 serialization introduces an additional  $\sim 33\%$  overhead, which is reflected in the reported wire sizes.

(C) *Slot utilization.* The final plot illustrates the fraction of utilized versus unutilized plaintext slots across all tested rings. Utilization  $u = (n \cdot record_s)/N$  quantifies packing efficiency within  $m_{DB}$ , which remains above 99% across configurations.

### C. BLOCKCHAIN PERFORMANCE

**Blockchain performance.** Figure 6 summarizes the performance of the CPIR chaincode within Hyperledger Fabric across three channels, each corresponding to a different ring size  $N$  and record configuration. The six subplots report storage breakdowns and peer-level resource utilization, capturing both ledger-side and runtime characteristics of the system.

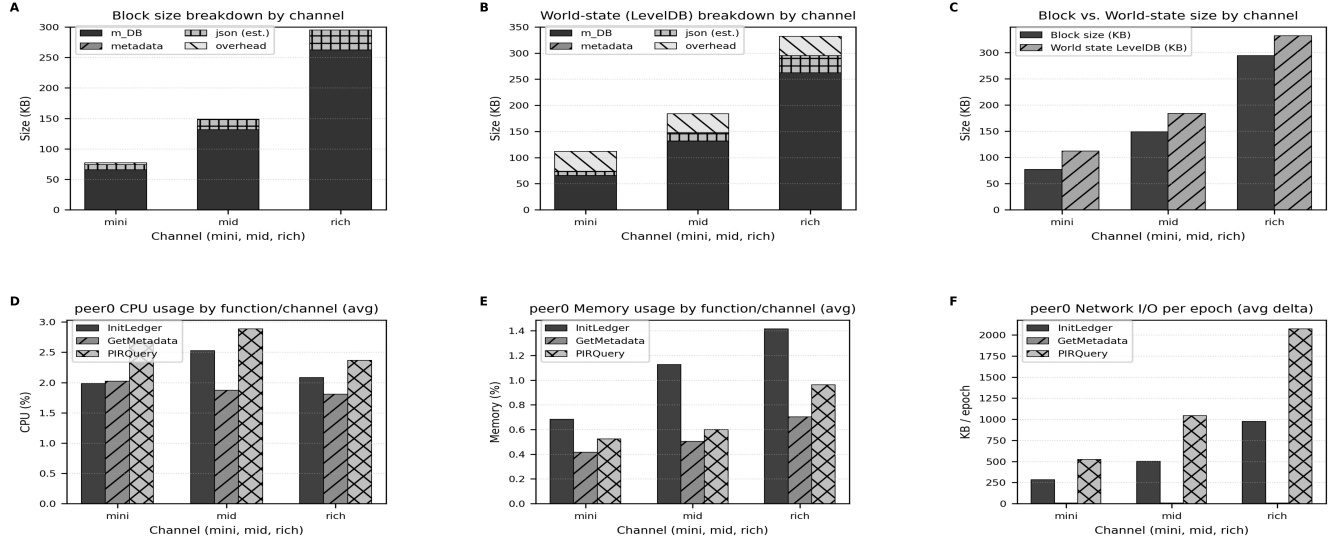
(A) *Block size breakdown.* This plot shows the total block payload produced by *InitLedger*, decomposed into the serial-



**FIGURE 5.** Cryptographic performance of the BGV-based CPIR system: (A) latency by algorithmic stage, (B) artifact size by ring configuration, and (C) allocated-slot utilization.

ized polynomial  $m_{DB}$ , metadata, JSON entries (estimated), and Fabric overhead. Only *InitLedger* produces blocks since *GetMetadata* and *PIRQuery* are evaluate-type transactions.

(B) *World-state (LevelDB) breakdown.* The on-chain world-state footprint mirrors the block structure, storing each entry as a key–value pair. The main contributors are the



**FIGURE 6.** Blockchain-side evaluation of the CPIR system across three Fabric channels. (A–C) Storage-level metrics: block and world-state composition. (D–F) Peer-level resource utilization: CPU, memory, and network I/O per function.

packed database polynomial  $m_{DB}$  and per-record metadata entries, with negligible LevelDB index overhead.

(C) *Block vs. world-state size.* This comparison illustrates the relative proportions of block payloads and their persisted world-state representations. The near-linear growth across channels reflects the increased plaintext vector size required to encode larger  $record_s$  and  $n$  configurations.

(D) *Peer CPU utilization.* Average CPU usage for each function (*InitLedger*, *GetMetadata*, *PIRQuery*) is shown per channel. *PIRQuery* exhibits the highest CPU activity due to homomorphic evaluation and ciphertext serialization, while *InitLedger* remains dominant for write-heavy workloads.

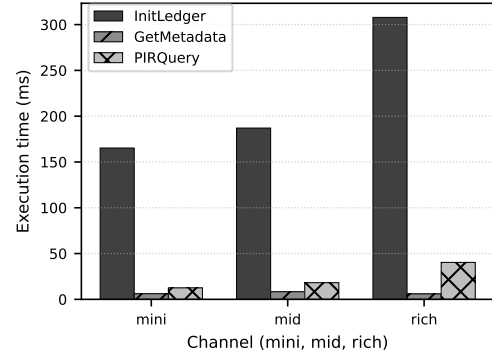
(E) *Peer memory utilization.* Memory usage stays below 1.5% across all cases. Peak values correspond to *InitLedger* in *mid* and *rich* channels, where large plaintext arrays and encoding buffers are instantiated during database packing and state commits.

(F) *Peer network I/O.* This plot reports the average data exchanged between the peer and client per epoch. *PIRQuery* incurs the largest network transfer, driven by the Base64-encoded ciphertexts ( $ct_q$ ,  $ct_r$ ), whereas metadata retrieval and ledger initialization are comparatively light.

**Chaincode Execution Timings.** Figure 7 and Table 5 summarize the average *server-side execution time* of the main chaincode functions across different ring sizes.

*Remark (execution paths).* Transactions that modify world state (e.g., *InitLedger*) are issued via the *submit* path and committed through Raft consensus, while read-only operations (*GetMetadata*, *PIRQuery*) use the *evaluate* path, bypassing block creation and ordering. All metrics are averaged over 20 executions under identical peer configurations.

Execution time of chaincode functions per channel (server-side avg)



**FIGURE 7.** Average chaincode execution time by function and ring size. Each bar represents the mean execution time over multiple epochs.

#### D. OVERALL SYSTEM PERFORMANCE

The experimental results collectively demonstrate that the proposed BGV-based CPIR design achieves consistent end-to-end performance within practical bounds for permissioned blockchains. Cryptographic evaluation (Table 3) shows that as  $\log N$  increases from 13 to 15, KeyGen latency roughly doubles and encryption and evaluation times grow proportionally to the ring size, while decryption remains below 16 ms.

**TABLE 3.** Execution Time of Cryptographic Operations (ms)

$\log N$	KeyGen	Enc	Eval	Dec
13	27.7	11.7	16.9	5.1
14	42.7	27.7	30.7	11.9
15	55.0	49.8	64.8	15.6

Artifact size in (Fig. 5B) and Table 4 scales linearly with the ring dimension  $N$ , public and secret keys exhibit a  $2\times$  growth per level, consistent with BGV’s key structure. Ciphertexts  $(ct_q, ct_r)$  approximately double in size with each increment in  $\log N$ , while the plaintext polynomial  $m_{DB}$  occupies roughly half of a ciphertext’s footprint. Metadata remains negligible in size ( $< 0.1$  KB).

$\log N$	pk	sk	$ct_q$	$ct_r$	$m_{DB}$	Metadata
13	256.1	128.0	128.3	128.3	64.3	0.08
14	512.1	256.0	256.3	256.3	128.3	0.08
15	1024.1	512.0	512.3	512.3	256.3	0.08

**TABLE 5. Average Chaincode Execution Time (ms)**

$\log N$	InitLedger	GetMetadata	PIRQuery
13	165.24	6.16	12.53
14	187.03	8.31	18.17
15	307.87	5.94	40.36

The first workflow corresponds to  $\mathcal{DW}$  initializing the ledger through *InitLedger*, which packs and encodes the plaintext database  $m_{DB}$  into world state.

The second workflow corresponds to  $\mathcal{DR}$  performing a private query by sequentially executing  $GetMetadata \rightarrow KeyGen \rightarrow Enc \rightarrow PIRQuery \rightarrow Dec$ . The PIR evaluation itself is performed by  $\mathcal{DO}$  (endorsing peer) using ciphertext-plaintext multiplication during the evaluate transaction.

Table 6 summarizes the cryptographic and blockchain timings for both workflows, averaged across all three channel configurations.

Workflow	Cryptographic Operations (ms)	Blockchain Operations (ms)	Total Time (ms)
$\mathcal{DW}$ 's Upload	0.0	220.0	220.0
$\mathcal{DR}$ 's Query	82.4	30.5	112.9

While the *InitLedger* operation incurs higher cost due to state updates and ordering, the end-to-end query workflow completes within  $\approx 113$  ms on average. Additionally, Figure 8 illustrates the peer- and client-side execution during a PIR query. While the endorsing peer observes only the ciphertext size and ring parameters, the actual queried index remains hidden.

These results demonstrate the practicality of integrating CPIR based on the BGV scheme into Hyperledger Fabric chaincode for privacy-preserving reads from world state database using the ciphertext-plaintext multiplication in evaluate type transactions.

Our proposed framework integrates a BGV-based Computational PIR (CPIR) mechanism directly into Hyperledger Fabric chaincode. By leveraging ciphertext-plaintext mul-

tiplication during evaluate transactions, it enables privacy-preserving reads from the ledger without modifying Fabric's transaction flow or consensus layer. This design allows query privacy to be achieved while maintaining low on-chain computation overhead and compatibility with existing Fabric deployments.

- additional operation in workflow (addRecord) + consequences for the design (refresh, re-encoding, etc) - selective field-level retrieval (adaptive packing with variable slot windows) - Lemma/theorem layer (constraints, correctness, security). - world-state vs. IPFS mention - dual mode for cpir (world state vs. blockchain history data) - relinearization for ct x ct evaluation

## VI. CONCLUSION

The conclusion goes here.

## REFERENCES

- [1] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, no. 6, pp. 965–981, nov 1 1998.



**FIRST A. AUTHOR** (Fellow, IEEE) and all authors may include biographies. Biographies are often not included in conference-related papers. This author is an IEEE Fellow. The first paragraph may contain a place and/or date of birth (list place, then date). Next, the author's educational background is listed. The degrees should be listed with type of degree in what field, which institution, city, state, and country, and year the degree was earned. The author's major field of study should be lower-cased.

The second paragraph uses the pronoun of the person (he or she) and not the author's last name. It lists military and work experience, including summer and fellowship jobs. Job titles are capitalized. The current job must have a location; previous positions may be listed without one. Information concerning previous publications may be included. Try not to list more than three books or published articles. The format for listing publishers of a book within the biography is: title of book (publisher name, year) similar to a reference. Current and previous research interests end the paragraph.

The third paragraph begins with the author's title and last name (e.g., Dr. Smith, Prof. Jones, Mr. Kajor, Ms. Hunter). List any memberships in professional societies other than the IEEE. Finally, list any awards and work for IEEE committees and publications. If a photograph is provided, it should be of good quality, and professional-looking.

**SECOND B. AUTHOR**, photograph and biography not available at the time of publication.

**THIRD C. AUTHOR JR.** (Member, IEEE), photograph and biography not available at the time of publication.