Received XX Month, XXXX; revised XX Month, XXXX; accepted XX Month, XXXX; Date of publication XX Month, XXXX; date of current version XX Month, XXXX.

Digital Object Identifier 10.1109/XXXX.2022.1234567

Blind Chaincode: Enabling Computational Private Information Retrieval for Query Privacy in Hyperledger Fabric

First A. Author[1], Fellow, IEEE, Second B. Author[2], and Third C. Author Jr.[3], Member, IEEE

[1]National Institute of Standards and Technology, Boulder, CO 80305 USA
[2]Department of Physics, Colorado State University, Fort Collins, CO 80523 USA
[3]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 `GetState` 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.

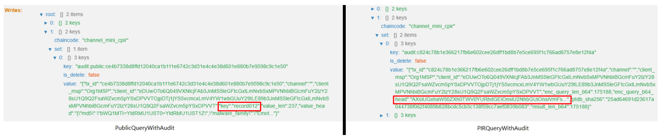
INDEX TERMS Private Information Retrieval (PIR), Homomorphic Encryption, Hyperledger Fabric, Permissioned Blockchains, Private Reads, Query Privacy.

I. INTRODUCTION

A. MOTIVATION AND CONTRIBUTION

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.

Private Information Retrieval (PIR) [1] addresses this challenge by enabling a client to retrieve an item from a database without revealing which item was requested. For a database $D = \{d_0, \ldots, d_{n-1}\}$, the client forms a one-hot selection vector $\hat{v}_i$ with a single "1" at index i. By encrypting $\hat{v}_i$ into $c_q = \text{Enc}_{pk}(\hat{v}_i)$ and sending it to the server, the server computes an encrypted response $c_r = c_q \cdot D = \text{Enc}_{pk}(d_i)$, which decrypts to d_i under the client's secret key. This construction hides the queried index from the server. When integrated with Fabric chaincode, PIR



```
Writes:
    ▼ root:  [] 2 items
      ► 0:  {} 2 keys
      ▼ 1:  {} 2 keys
          chaincode:  "channel_mini_cpir"
        ▼ set:  [] 1 item
          ▼ 0:  {} 3 keys
              key:  "audit:public:ce4b7338d8ffd12040ca1b111e6742c3d31e4c4e38d601e880b7e9598c9c1e50"
              is_delete:  false
              value:  "{"tx_id":"ce4b7338d8ffd12040ca1b111e6742c3d31e4c4e38d601e880b7e9598c9c1e50","channel":"","client
                _msp":"Org1MSP","client_id":"eDUwOTo6Q049VXNlcjFAb3JnMS5leGFtcGxlLmNvbSxMPVNhbiBGcmFuY2lzY2
                8sU1Q9Q2FsaWZvcm5pYSxDPVVTOjpDTj1jYS5vcmcxLmV4YW1wbGUuY29tLE89b3JnMS5leGFtcGxlLmNvbS
                xMPVNhbiBGcmFuY2lzY28sU1Q9Q2FsaWZvcm5pYSxDPVVT","key":"record012","value_len":237,"value_hea
                d":"{\"md5\":\"bWQ1MTI=YldRMU1UST0=YldRMU1UST1Z\",\"malware_family\":\"Emot..."}"
```

PublicQueryWithAudit

```
    ► 0:  {} 2 keys
    ▼ 1:  {} 2 keys
        chaincode:  "channel_mini_cpir"
      ▼ set:  [] 2 items
        ▼ 0:  {} 3 keys
            key:  "audit:c824c78b1e366217fb6e602cee26dff1bd8b7e5ce695f1c766ad8757e8e12f4a"
            is_delete:  false
            value:  "{"tx_id":"c824c78b1e366217fb6e602cee26dff1bd8b7e5ce695f1c766ad8757e8e12f4a","channel":"","client_
              msp":"Org1MSP","client_id":"eDUwOTo6Q049VXNlcjFAb3JnMS5leGFtcGxlLmNvbSxMPVNhbiBGcmFuY2lzY28
              sU1Q9Q2FsaWZvcm5pYSxDPVVTOjpDTj1jYS5vcmcxLmV4YW1wbGUuY29tLE89b3JnMS5leGFtcGxlLmNvbSx
              MPVNhbiBGcmFuY2lzY28sU1Q9Q2FsaWZvcm5pYSxDPVVT","enc_query_len_b64":175188,"enc_query_b64_
              head":"AXsiUGxhaW50ZXh0TWV0YURhdGEiOnsiU2NhbGUiOnsiVmFs...","ptdb_sha256":"25ad64691d23617a
              044738f062f4089b828bcdc5cb3c138f59cc7aef0839b083","result_len_b64":175188}"
        ► 1:  {} 3 keys
```

PIRQueryWithAudit

FIGURE 1. What endorsing peers "see" in audit records. Left: baseline *PublicQueryWithAudit* exposes the queried key (*record012*). Right: CPIR-based *PIRQueryWithAudit* exposes only an opaque encrypted selector (*EncQueryB64*), hiding the queried index.

prevents endorsing peers from linking queries to specific records, while preserving blockchain auditability for writes.

Figure 1 illustrates this contrast. A baseline query call *PublicQueryWithAudit("record012")* explicitly reveals the queried key in the audit log visible to endorsing peers. In contrast, our *PIRQueryWithAudit(encQueryB64)* logs only an opaque Base64-encoded selector, preventing the audit trail or the read-set from exposing query intent.

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 on-chain implementation of Computational PIR (CPIR) based on the BGV scheme, integrated directly into Hyperledger Fabric chaincode.
2) **Evaluate-phase privacy demonstration.** We provide a side-by-side analysis of baseline vs. CPIR queries, showing how endorsing peers' visibility is reduced to opaque ciphertexts while preserving Fabric's normal endorsement and audit mechanisms.
3) **Prototype and benchmarks.** We implement a working system with the Lattigo library and measure client encryption/decryption, peer evaluation, ciphertext size, and end-to-end query latency, demonstrating practical performance for consortium deployments.
4) **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.

B. LITERATURE REVIEW

Privacy in permissioned blockchains has been studied from several angles, but query privacy remains underexplored.

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 often rely on off-chain servers or specialized cryptographic protocols.

On-Chain CPIR Gap. Existing solutions for Fabric focus on access restriction (channels, PDC) or enclave-based confidentiality (FPC). Recent PIR-based proposals either target off-chain databases or prototype OT-based protocols. To our knowledge, no prior system has directly integrated a lattice-based CPIR scheme into Fabric chaincode. Our work closes this 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.

C. ORGANIZATION

The remainder of this paper is organized as follows: Section II surveys related work on PIR and blockchain privacy. Section III presents the system model and threat assumptions. Section IV details the design and implementation of CPIR in Fabric. Section V evaluates performance. Section VI discusses limitations and future directions, and Section VII concludes the paper.

II. PRELIMINARIES

A. TECHNOLOGY BACKGROUND

a) Fabric Native Privacy Techniques: Hyperledger Fabric separates roles among endorsing peers, committing peers, and the ordering service. Endorsing peers execute chaincode proposals and therefore observe function arguments, logs, and read-sets, making their administrators the natural adversaries for read privacy. Protecting reads in Fabric thus requires hiding query intent from endorsers.

Fabric already offers several native privacy mechanisms, each addressing a different dimension of confidentiality:

- *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.

In summary, Fabric's native privacy tools govern data visibility and execution confidentiality. They are orthogonal to Private Information Retrieval (PIR): PDC and FPC restrict who can see data, while PIR hides what data is queried.

b) Private Information Retrieval Basics: PIR protocols enable a client to retrieve a record without revealing which record was requested. They fall into two categories:

Information-Theoretic PIR (IT-PIR). Provides unconditional privacy by distributing the database across multiple non-colluding servers. A client queries subsets of servers such that no single server learns the selection index.

Computational PIR (CPIR). Achieves privacy with a single server, relying on hardness assumptions and homomorphic encryption. For a database $D = \{d_0, \ldots, d_{n-1}\}$ and a one-hot selection vector $\hat{v}_i$, the client computes

$$ct_q = \mathrm{Enc}_{pk}(\hat{v}_i), \qquad ct_r = ct_q \cdot D = \mathrm{Enc}_{pk}(d_i).$$

The server returns c_r, which the client decrypts as $d_i = \mathrm{Dec}_{sk}(c_r)$. Thus the queried index i remains hidden from the server.

CPIR avoids the need for multiple servers, making it attractive in blockchain settings where peers cannot be assumed non-colluding.

c) 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 is defined over polynomial rings modulo a large ciphertext modulus and 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 m_{DB}, allowing efficient evaluation of structured PIR queries inside chaincode and underpins the polynomial database representation used in our Fabric integration.

B. NOTATION

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

C. CRYPTOGRAPHIC PRIMITIVES

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.

TABLE 1. Notation

Symbol	Description
λ	Security parameter
n	Database size; index domain $[n] = \{0, \ldots, n-1\}$
$D = \{d_0, \ldots, d_{n-1}\}$	Database records (serialized bytes/words)
m_{DB}	Plaintext polynomial representation of D (BGV batching)
$\hat{v}_i$	One-hot selector for index i (single 1, rest 0)
v_i	Windowed selector for index i with $record_s$ contiguous ones (retrieves full record window)
pk, sk	Public / secret keys (BGV)
$ct_q = \mathsf{Enc}_{pk}(\hat{v}_i)$	Encrypted query
$ct_r = \mathsf{Eval}(ct_q, m_{\mathrm{DB}})$	Encrypted response
$d_i = \mathsf{Dec}_{sk}(ct_r)$	Decrypted record d_i retrieved by client
$\mathsf{KeyGen}(\lambda) \to (pk, sk)$	Key generation
$\mathsf{Enc}_{pk}(\cdot)$, $\mathsf{Dec}_{sk}(\cdot)$	Encrypt / Decrypt
$\mathsf{Eval}(\cdot)$	Homomorphic evaluation (ct–pt multiply)
$N = 2^{\log N}$	Ring dimension (polynomial degree of the scheme)
$\log Q_i$	Bit-lengths of primes forming modulus chain Q (ciphertext levels)
$\log P_i$	Bit-lengths of special primes P (used for key switching / relinearization)
T	Plaintext modulus (NTT-friendly prime; Lattigo `PlaintextModulus`)
$record_s$	Slots allocated per record (slot window size)
$record_b$	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 (implementation policy)
$c = (c_0, \ldots, c_{N-1})$	Coefficient vector of the polynomial encoding (slots of m_{DB})
$\lvert \cdot \rvert$, size$(\cdot)$	Length in elements / size in bytes
$\mathsf{Cap}(N, s, n)$	Capacity predicate: $n \cdot s \le N$
$\mathsf{Min}(\log N, s)$	Template predicate: $s \ge record_{\mu,\log N}$
$\mathsf{Disc}(s)$	Discrete predicate: $s \in \mathcal{S}$
$\mathsf{Cap} \wedge \mathsf{Min} \wedge \mathsf{Disc}$	Feasibility condition for $(\log N, n, record_s)$
$\mathcal{DO}$, $\mathcal{DW}$, $\mathcal{DR}$, $\mathcal{GW}$	Data Owner; Data Writer; Data Requester; Gateway
evaluate, submit	Fabric read / write transaction phases
$\mathcal{L}$	Leakage considered (ciphertext size, protocol timing)

We instantiate Computational PIR as a tuple of probabilistic polynomial-time algorithms

$$\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec}),$$

defined as follows:

- $\mathsf{KeyGen}(\lambda) \to (pk, sk)$: On input the security parameter λ, output a public key pk and a secret key sk.

- $\mathsf{Enc}_{pk}(\hat{v}_i) \to ct_q$: 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.
- $\mathsf{Eval}(ct_q, m_{\mathrm{DB}}) \to ct_r$: Given ct_q and the plaintext polynomial database $m_{\mathrm{DB}} \in R_T$, homomorphically evaluate the product to obtain an encrypted response $ct_r \in R_Q$.
- $\mathsf{Dec}_{sk}(ct_r) \to d_i$: 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 $i \in [n]$,

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

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.

III. PROPOSED SYSTEM

The following subsections detail (i) the system and threat models, (ii) a system overview, (iii) the polynomial database construction, (iv) feasibility constraints for parameter selection, (v) multi-channel architecture, (vi) workflow details and main 5 algorithms.

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 (CPIR) 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 ($\mathcal{DO}$):** Endorsing peers that hold the current plaintext polynomial m_{DB} in world state and execute PIR during evaluate. $\mathcal{DO}$ is honest-but-curious.
- **Data Writer ($\mathcal{DW}$):** A client organization that provisions or refreshes the database. $\mathcal{DW}$ invokes submit to initialize the ledger (e.g., set n and template bounds). Chaincode computes $record_s$, packs $D = \{d_0, \ldots, d_{n-1}\}$, encodes it into m_{DB}, and persists it.
- **Data Requester ($\mathcal{DR}$):** A client that privately retrieves a record. $\mathcal{DR}$ runs $\mathsf{KeyGen}(\lambda) \to (pk, sk)$, forms $ct_q = \mathsf{Enc}_{pk}(v_i)$, calls evaluate PIRQuery, and later decrypts ct_r.
- **Gateway ($\mathcal{GW}$):** The Fabric client/chaincode interface used by $\mathcal{DW}$ and $\mathcal{DR}$ 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:

- **Endorsing peers ($\mathcal{DO}$).** 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 ($\mathcal{DW}$).** Issues initialization writes via submit. $\mathcal{DW}$ is not trusted with decryption keys and learns nothing about $\mathcal{DR}$'s queries. We assume $\mathcal{DW}$ 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 $\mathcal{DO}$ 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. 2.

1) **Ledger initialization.** $\mathcal{DW}$ invokes InitLedger via $\mathcal{GW}$ using submit. Chaincode derives $record_s$ from $record_b$, packs D into $c = (c_0, \ldots, c_{N-1})$, encodes m_{DB}, and stores m_{DB} and metadata in world state held by $\mathcal{DO}$.
2) **Metadata discovery.** $\mathcal{DR}$ calls GetMetadata via evaluate to obtain n, $record_s$, and BGV parameters needed to form a valid query.

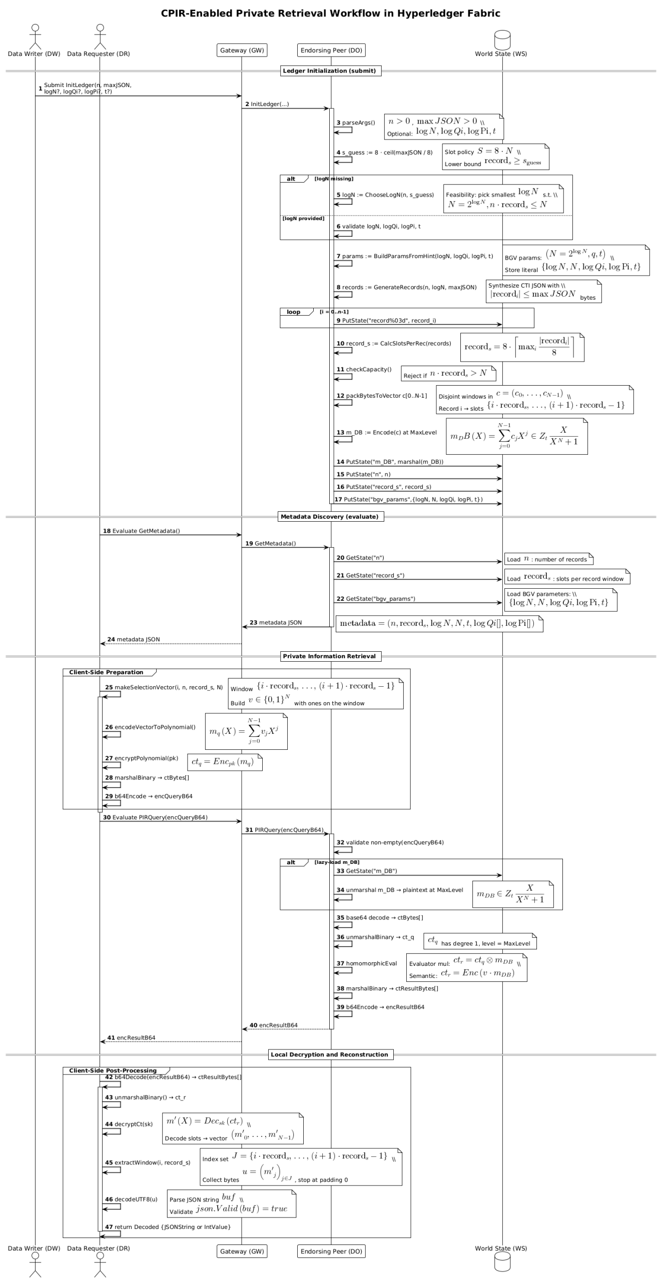

CPIR-Enabled Private Retrieval Workflow in Hyperledger Fabric
Data Writer (DW)
Data Requester (DR)
Gateway (GW)
Endorsing Peer (DO)
World State (WS)
Ledger Initialization (submit)
1 Submit InitLedger(n, maxJSON, logN?, logQi?, logPi?, t?)
2 InitLedger(...)
3 parseArgs()
$n > 0$, $\max JSON > 0$ \\ Optional: $\log N, \log Qi, \log \mathrm{Pi}, t$
4 s_guess := 8 · ceil(maxJSON / 8)
Slot policy $S = 8 \cdot N$ \\ Lower bound $\mathrm{record}_s \geq s_{\mathrm{guess}}$
alt [logN missing]
5 logN := ChooseLogN(n, s_guess)
Feasibility: pick smallest $\log N$ s.t. \\ $N = 2^{\log N}, n \cdot \mathrm{record}_s \leq N$
[logN provided]
6 validate logN, logQi, logPi, t
7 params := BuildParamsFromHint(logN, logQi, logPi, t)
BGV params: $(N = 2^{\log N}, q, t)$ \\ Store literal $\{\log N, N, \log Qi, \log \mathrm{Pi}, t\}$
8 records := GenerateRecords(n, logN, maxJSON)
Synthesize CTI JSON with \\ $|\mathrm{record}_i| \leq \max JSON$ bytes
loop [i = 0..n-1]
9 PutState("record%03d", record_i)
10 record_s := CalcSlotsPerRec(records)
$\mathrm{record}_s = 8 \cdot \left\lceil \max_i \frac{|\mathrm{record}_i|}{8} \right\rceil$
11 checkCapacity()
Reject if $n \cdot \mathrm{record}_s > N$
12 packBytesToVector c[0..N-1]
Disjoint windows in $c = (c_0, \ldots, c_{N-1})$ \\ Record i → slots $\{i \cdot \mathrm{record}_s, \ldots, (i+1) \cdot \mathrm{record}_s - 1\}$
13 m_DB := Encode(c) at MaxLevel
$m_D B(X) = \sum_{j=0}^{N-1} c_j X^j \in Z_t \frac{X}{X^N+1}$
14 PutState("m_DB", marshal(m_DB))
15 PutState("n", n)
16 PutState("record_s", record_s)
17 PutState("bgv_params",{logN, N, logQi, logPi, t})
Metadata Discovery (evaluate)
18 Evaluate GetMetadata()
19 GetMetadata()
20 GetState("n")
Load n : number of records
21 GetState("record_s")
Load record_s : slots per record window
22 GetState("bgv_params")
Load BGV parameters: \\ $\{\log N, N, \log Qi, \log \mathrm{Pi}, t\}$
23 metadata JSON
$\mathrm{metadata} = (n, \mathrm{record}_s, \log N, N, t, \log Qi[], \log \mathrm{Pi}[])$
24 metadata JSON
Private Information Retrieval
Client-Side Preparation
25 makeSelectionVector(i, n, record_s, N)
Window $\{i \cdot \mathrm{record}_s, \ldots, (i+1) \cdot \mathrm{record}_s - 1\}$
Build $v \in \{0,1\}^N$ with ones on the window
26 encodeVectorToPolynomial()
$m_q(X) = \sum_{j=0}^{N-1} v_j X^j$
27 encryptPolynomial(pk)
$ct_q = Enc_{pk}(m_q)$
28 marshalBinary → ctBytes[]
29 b64Encode → encQueryB64
30 Evaluate PIRQuery(encQueryB64)
31 PIRQuery(encQueryB64)
32 validate non-empty(encQueryB64)
alt [lazy-load m_DB]
33 GetState("m_DB")
34 unmarshal m_DB → plaintext at MaxLevel
$m_{DB} \in Z_t \frac{X}{X^N+1}$
35 base64 decode → ctBytes[]
36 unmarshalBinary → ct_q
ct_q has degree 1, level = MaxLevel
37 homomorphicEval
Evaluator mul: $ct_r = ct_q \otimes m_{DB}$ \\ Semantic: $ct_r = Enc(v \cdot m_{DB})$
38 marshalBinary → ctResultBytes[]
39 b64Encode → encResultB64
40 encResultB64
41 encResultB64
Local Decryption and Reconstruction
Client-Side Post-Processing
42 b64Decode(encResultB64) → ctResultBytes[]
43 unmarshalBinary() → ct_r
44 decryptCt(sk)
$m'(X) = Dec_{sk}(ct_r)$ \\ Decode slots → vector $(m'_0, \ldots, m'_{N-1})$
45 extractWindow(i, record_s)
Index set $J = \{i \cdot \mathrm{record}_s, \ldots, (i+1) \cdot \mathrm{record}_s - 1\}$ \\ Collect bytes $u = (m'_j)_{j \in J}$, stop at padding 0
46 decodeUTF8(u)
Parse JSON string buf \\ Validate $json.Valid(buf) = true$
47 return Decoded {JSONString or IntValue}

FIGURE 2. **"Workflow. $\mathcal{DW}$ initializes the ledger via $\mathcal{GW}$, which triggers chaincode on endorsing peers ($\mathcal{DO}$). $\mathcal{DO}$ executes the protocol and persists state in world state (m_{DB}, metadata, JSON records). $\mathcal{DR}$ later obtains metadata, submits $ct_q = \mathsf{Enc}pk(v_i)$, $\mathcal{DO}$ evaluates $ct_r = \mathsf{Eval}(ct_q, m\mathrm{DB})$ against world state, and $\mathcal{DR}$ decrypts to d_i.**

3) **Private retrieval.** $\mathcal{DR}$ constructs $ct_q = \mathsf{Enc}_{pk}(v_i)$ and invokes PIRQuery via evaluate. $\mathcal{DO}$ computes $ct_r = \mathsf{Eval}(ct_q, m_{\mathrm{DB}})$ and returns it.
4) **Decryption.** $\mathcal{DR}$ decrypts ct_r to recover $d_i = \mathsf{Dec}_{sk}(ct_r)$.

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. 9, 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 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 $c = (c_0, c_1, \ldots, c_{N-1})$, 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:

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

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 $\log N =$

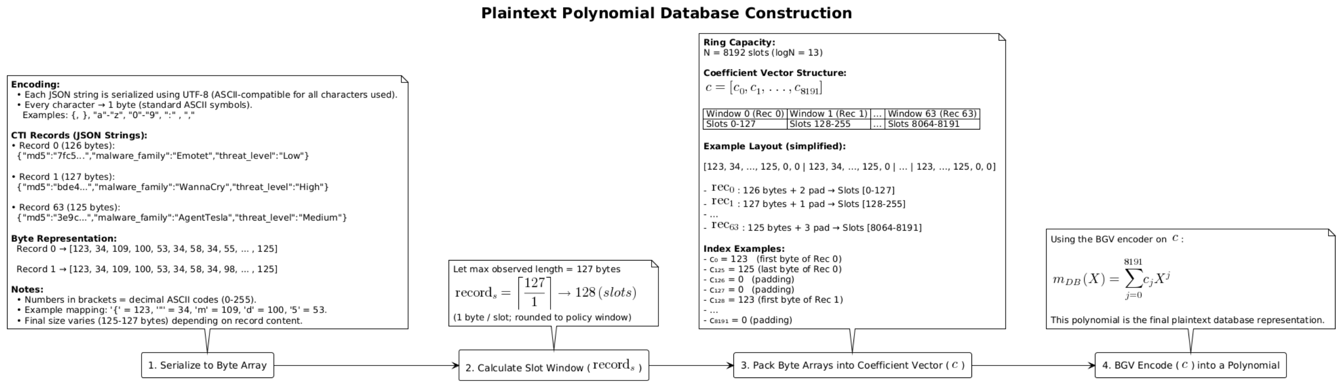

Plaintext Polynomial Database Construction
Encoding:
• Each JSON string is serialized using UTF-8 (ASCII-compatible for all characters used).
• Every character → 1 byte (standard ASCII symbols).
Examples: {, }, "a"-"z", "0"-"9", ":" , ","
CTI Records (JSON Strings):
• Record 0 (126 bytes):
{"md5":"7fc5...","malware_family":"Emotet","threat_level":"Low"}
• Record 1 (127 bytes):
{"md5":"bde4...","malware_family":"WannaCry","threat_level":"High"}
• Record 63 (125 bytes):
{"md5":"3e9c...","malware_family":"AgentTesla","threat_level":"Medium"}
Byte Representation:
Record 0 → [123, 34, 109, 100, 53, 34, 58, 34, 55, ... , 125]
Record 1 → [123, 34, 109, 100, 53, 34, 58, 34, 98, ... , 125]
Notes:
• Numbers in brackets = decimal ASCII codes (0-255).
• Example mapping: '{' = 123, '"' = 34, 'm' = 109, 'd' = 100, '5' = 53.
• Final size varies (125-127 bytes) depending on record content.
Let max observed length = 127 bytes
$\mathrm{record}_s = \left\lceil \frac{127}{1} \right\rceil \rightarrow 128\,(slots)$
(1 byte / slot; rounded to policy window)
Ring Capacity:
N = 8192 slots (logN = 13)
Coefficient Vector Structure:
$c = [c_0, c_1, \ldots, c_{8191}]$
Window 0 (Rec 0) | Window 1 (Rec 1) | ... | Window 63 (Rec 63)
Slots 0-127 | Slots 128-255 | ... | Slots 8064-8191
Example Layout (simplified):
[123, 34, ..., 125, 0, 0 | 123, 34, ..., 125, 0 | ... | 123, ..., 125, 0, 0]
- rec_0 : 126 bytes + 2 pad → Slots [0-127]
- rec_1 : 127 bytes + 1 pad → Slots [128-255]
- ...
- rec_{63} : 125 bytes + 3 pad → Slots [8064-8191]
Index Examples:
- c_0 = 123 (first byte of Rec 0)
- c_{125} = 125 (last byte of Rec 0)
- c_{126} = 0 (padding)
- c_{127} = 0 (padding)
- c_{128} = 123 (first byte of Rec 1)
- ...
- c_{8191} = 0 (padding)
Using the BGV encoder on c :
$m_{DB}(X) = \sum_{j=0}^{8191} c_j X^j$
This polynomial is the final plaintext database representation.
1. Serialize to Byte Array
2. Calculate Slot Window (record_s)
3. Pack Byte Arrays into Coefficient Vector (c)
4. BGV Encode (c) into a Polynomial

FIGURE 3. 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.**

13 ($N = 8192$) and $record_s = 224$, at most $\lfloor 8192/224 \rfloor = 36$ 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:

$$\begin{aligned} \text{Mini:}\quad & record_s \ \geq\ record_b + 32 + 15, \\ \text{Mid:}\quad & record_s \ \geq\ record_b + 32 + 16 + 15, \\ \text{Rich:}\quad & record_s \ \geq\ record_b + 32 + 64 + 15, \end{aligned}$$

where $record_b \approx$ 113–125 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:

$$\begin{aligned} \mathsf{Cap}(N, s, n) &: n \cdot s \leq N \quad \text{(ring capacity)} \\ \mathsf{Min}(\log N, s) &: s \geq record_{\mu,\log N} \quad \text{(template minimum)} \\ \mathsf{Disc}(s) &: s \in \mathcal{S} \quad \text{(discrete allocation).} \end{aligned}$$

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

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

Examples. The interaction of these predicates is illustrated in Fig. 4. We provide three concrete examples below:

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

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

b) Polynomial degree $\log N = 14$ (Mid):

- $\mathsf{Min}(14, s)$ enforces $s \geq record_{\mu,14} \approx 224 \Rightarrow$ smallest candidate is 224.
- $\mathsf{Disc}(s)$ allows $224, 256, \ldots$ but excludes smaller windows.
- $\mathsf{Cap}(16384, 224, n)$ gives $n \leq \lfloor 16384/224 \rfloor = 73$.
- *Feasible:* $(\log N = 14, s \in \{224, 256, \ldots\}, n \leq 73)$.

c) Polynomial degree $\log N = 15$ (Rich):

- $\mathsf{Min}(15, s)$ enforces $s \geq record_{\mu,15} \approx 256$.
- $\mathsf{Disc}(s)$ excludes $64, 128, 224$; smallest valid is 256.
- $\mathsf{Cap}(32768, 256, n)$ 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 $record_s$ if given richer templates. Feasible configurations occur only where all 3 predicates are met, guiding practical parameter selection.

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

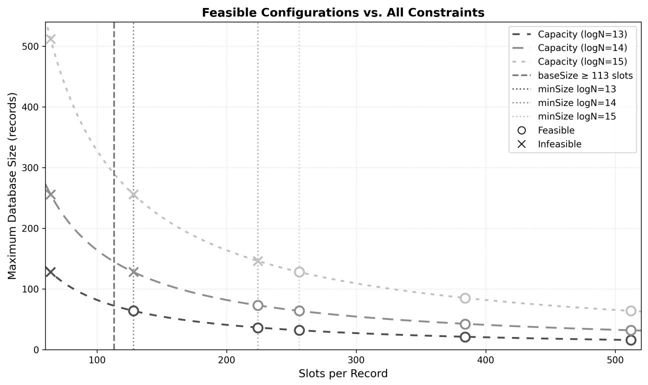


FIGURE 4. Feasible configurations under the joint constraints Cap, Min, **and** Disc. **Dashed curves show ring-capacity limits for** $\log N \in \{13, 14, 15\}$**, vertical lines mark template-driven minima** $record_{\mu,\log N}$**, and x-axis ticks correspond to discrete slot sizes** $\mathcal{S}$**. Circles indicate feasible triples** $(\log N, record_s, n)$**.**

(Fig. 5), 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 ≤ 64 bytes or 32 records at ≤ 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 ≤ 64 bytes or 64 records at ≤ 512 bytes.

Channel semantics. As shown in Fig. 5, 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 $record_s$,
  - `"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.

a) 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.

b) 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. 4) 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.

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 2 provides the high-level overview. A Data Writer ($\mathcal{DW}$) provisions the database $D = \{d_0, \ldots, d_{n-1}\}$, a Data Owner ($\mathcal{DO}$) maintains the polynomial m_{DB} in world state and executes PIR evaluations, and a Data Requester ($\mathcal{DR}$) retrieves d_i privately using homomorphic encryption. The detailed steps are as follows:

1) **DW submits initialization.** $\mathcal{DW}$ calls InitLedger (Alg. 1) via $\mathcal{GW}$ (submit) with inputs $(n, record_s^{\mathcal{DW}})$ and an optional hint $(\log N, \log Q_i, \log P_i, T)$. Here $record_s^{\mathcal{DW}}$ denotes the maximum JSON size anticipated by the writer.
2) **DO validates and derives parameters.** $\mathcal{DO}$ rounds the writer's input to a discrete slot size $record_s^{\mathcal{GW}} =$

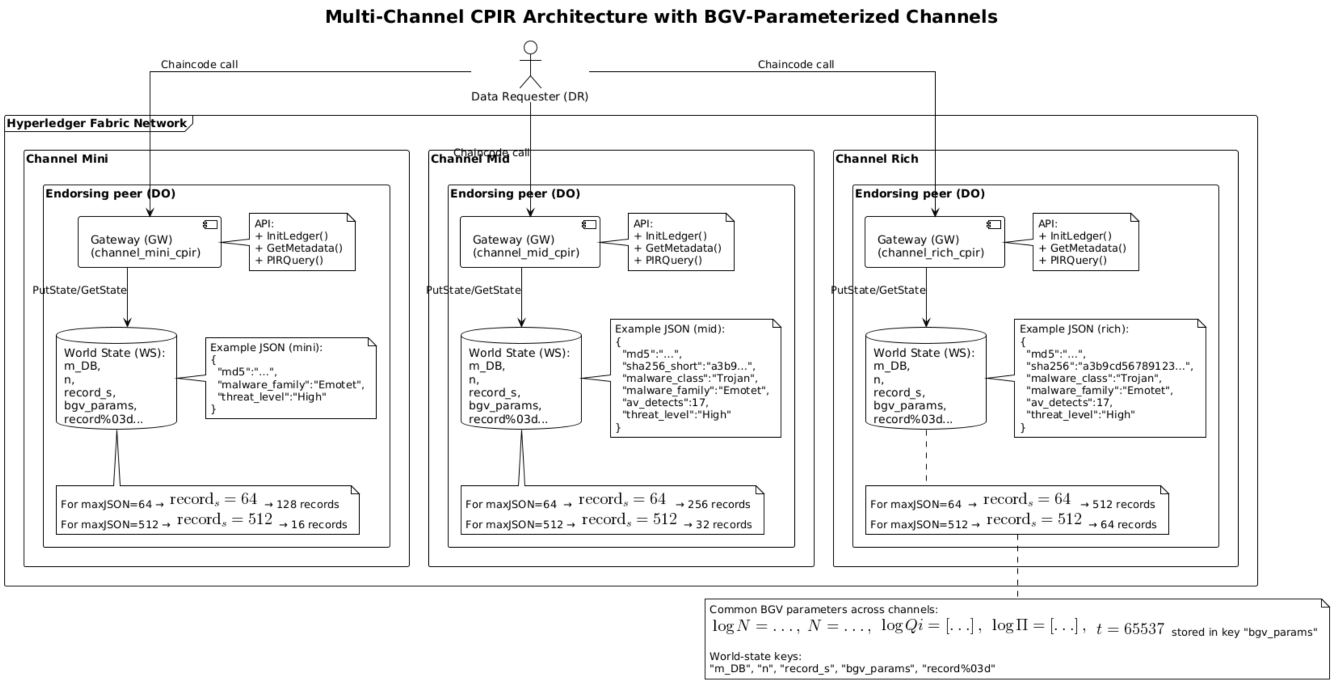

Multi-Channel CPIR Architecture with BGV-Parameterized Channels
Chaincode call
Data Requester (DR)
Chaincode call
Hyperledger Fabric Network
Channel Mini
Channel Mid
Chaincode call
Channel Rich
Endorsing peer (DO)
Gateway (GW) (channel_mini_cpir)
Gateway (GW) (channel_mid_cpir)
Gateway (GW) (channel_rich_cpir)
API: + InitLedger() + GetMetadata() + PIRQuery()
PutState/GetState
World State (WS): m_DB, n, record_s, bgv_params, record%03d...
Example JSON (mini): { "md5":"...", "malware_family":"Emotet", "threat_level":"High" }
Example JSON (mid): { "md5":"...", "sha256_short":"a3b9...", "malware_class":"Trojan", "malware_family":"Emotet", "av_detects":17, "threat_level":"High" }
Example JSON (rich): { "md5":"...", "sha256":"a3b9cd56789123...", "malware_class":"Trojan", "malware_family":"Emotet", "av_detects":17, "threat_level":"High" }
For maxJSON=64 → $\mathrm{record}_s = 64$ → 128 records
For maxJSON=512 → $\mathrm{record}_s = 512$ → 16 records
For maxJSON=64 → $\mathrm{record}_s = 64$ → 256 records
For maxJSON=512 → $\mathrm{record}_s = 512$ → 32 records
For maxJSON=64 → $\mathrm{record}_s = 64$ → 512 records
For maxJSON=512 → $\mathrm{record}_s = 512$ → 64 records
Common BGV parameters across channels:
$\log N = \ldots,\ N = \ldots,\ \log Qi = [\ldots],\ \log \Pi = [\ldots],\ t = 65537$ stored in key "bgv_params"
World-state keys:
"m_DB", "n", "record_s", "bgv_params", "record%03d"

FIGURE 5. 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.

$8 \cdot \lceil record_s^{\mathcal{DW}}/8 \rceil$. If $\log N$ is absent, the smallest feasible $\log N$ is chosen such that $\mathsf{Cap}(N, record_s^{\mathcal{GW}}, n)$ holds. BGV parameters $\{\log N, N, \log Q_i, \log P_i, T\}$ are constructed and stored.

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

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, $\mathcal{GW}$ still returns the same metadata blob; clients rebuild context idempotently.

IV. PERFORMANCE EVALUATION

The following subsections detail (i) experimental setup and cryptographic benchmarks, (ii) fabric integration becnhmarks, (iii) discussion and comparison with related work.

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.

Algorithm 1 InitLedger (chaincode)

```
Require: $n$; $record_s^{\mathcal{DW}}$; op: hint
 1: $\log N \leftarrow \mathsf{selMinLogN}\Big(\big(n,\ 8\cdot\big\lceil \frac{record_s^{\mathcal{DW}}}{8}\big\rceil\big)\Big)$
 2: if $\log N = \varnothing$ then
 3:     return $\perp$
 4: end if
 5: $bgvParams \leftarrow \mathsf{selParams}((\log N,\ op : \text{hint}))$
 6: $D \leftarrow \mathsf{genRecords}\big((n,\ record_s^{\mathcal{DW}})\big)$
 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 $\neg\mathsf{feasible}(\log N, n, record_s)$ 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:             $c[J_i[k]] \leftarrow \mathsf{byte}(d_i[k])$    // copy byte of record
20:         else
21:             $c[J_i[k]] \leftarrow 0$    // padding
22:         end if
23:     end for
24: end for
25: $m_{\text{DB}}(X) \leftarrow \mathsf{enc}^{\text{poly}}(c) \in \mathbb{Z}_T[X]/(X^N+1)$
26: $worldState \leftarrow \{m_{\text{DB}},\ n,\ record_s,\ bgvParams,\ op : D\}$
27: return OK
```

Algorithm 2 GetMetadata (chaincode)

```
Require: $\varnothing$
 1: $n \leftarrow worldState.n$
 2: $record_s \leftarrow worldState.record_s$
 3: $paramsMeta \leftarrow worldState.bgvParams$
 4: if $n = \varnothing \ \vee\ record_s = \varnothing \ \vee\ paramsMeta = \varnothing$ 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 $metadata$
```

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 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: $m_q(X) \leftarrow \mathsf{enc}^{\text{poly}}(v_i)$    // polynomial encode at max level
13: $ct_q \leftarrow \mathsf{Enc}_{pk}(m_q)$
14: $ct_q^{B64} \leftarrow \mathsf{enc}^{\text{B64}}\big(\mathsf{ser}^{\text{bin}}(ct_q)\big)$
15: return $ct_q^{B64}$    // Base64(marshalled ciphertext)
```

Algorithm 4 PIRQuery (chaincode)

```
Require: $ct_q^{B64}$
 1: if $ct_q^{B64} = \varnothing$ then
 2:     return $\perp$
 3: end if
 4: $ct_q \leftarrow \mathsf{des}^{\text{bin}}\Big(\mathsf{dec}^{\text{B64}}\big((ct_q^{B64})\big)\Big)$
 5: if $m_{\text{DB}}$ not cached in memory then
 6:     $m_{\text{DB}} \leftarrow worldState.m_{\text{DB}}$
 7: end if
 8: $ct_r \leftarrow \mathsf{Eval}(ct_q, m_{\text{DB}})$
 9: $ct_r^{B64} \leftarrow \mathsf{enc}^{\text{B64}}\big(\mathsf{ser}^{\text{bin}}((ct_r))\big)$
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: $ct_r \leftarrow \mathsf{des}^{\text{bin}}\Big(\mathsf{dec}^{\text{B64}}\big((ct_r^{B64})\big)\Big)$
 4: $u \in \mathbb{Z}_T^N \leftarrow \mathsf{dec}^{\text{poly}}(\ m'(X)\ ) \leftarrow \mathsf{Dec}_{sk}(ct_r)$
 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:     $b.\mathsf{append}(u[j])$
12: end for
13: if $record_s = 1$ then return $u[i \cdot record_s]$
14: $d_i \leftarrow dec^{UTF8}(b)$
15: return $d_i$
```

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*.

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 lists the main cryptographic parameters used in the evaluation, including the ring dimension N, modulus chain $(\log Q_i, \log P_i)$, plaintext modulus T, slot allocation $record_s$, number of records n, and selector type.

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

End-to-End Latency by Stage. Figure 6 presents the breakdown of single-query latency across cryptographic operations for each ring size.

Artifact Size Analysis. Figure 7 reports the serialized size (in kilobytes) of the main cryptographic artifacts: the public and secret keys, the encrypted selector ct_q, the encrypted response ct_r, the plaintext database polynomial m_{DB}, and the metadata JSON object.

Slot Utilization. Figure 8 illustrates the proportion of utilized versus unutilized slots across different ring sizes $N = 2^{\log N}$. Utilization $u = (n \cdot record_s)/N$ measures how efficiently plaintext slots are allocated within the polynomial database m_{DB}.

Having characterized the cryptographic performance of the BGV-based CPIR scheme in isolation—covering latency, artifact size, and slot utilization—we now turn to its **on-chain execution behavior within Hyperledger Fabric**. The following subsection measures the performance of the chaincode functions (InitLedger, GetMetadata, and PIRQuery) across channels, focusing on peer-side latency, throughput, and world-state footprint.

C. BLOCKCHAIN PERFORMANCE

a) Chaincode Execution Timings. Figure 10 and Table 3 summarize the average *server-side execution time* of the main chaincode functions across different ring sizes $N = 2^{\log N}$.

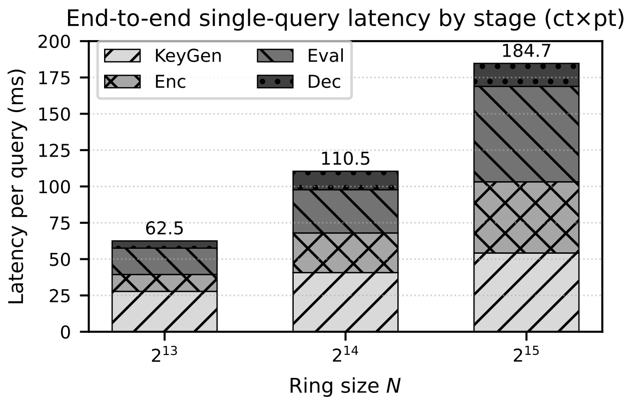


FIGURE 6. End-to-end single-query latency by stage; `ct×pt` path.

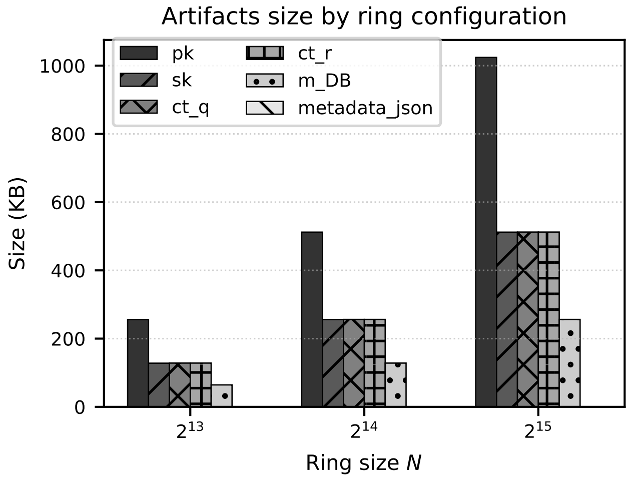


FIGURE 7. Size of main cryptographic artifacts.

TABLE 3. Average chaincode execution time (ms) per function and ring size

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

Remark (Submittion paths reminder). Transactions that modify world state (e.g., InitLedger) are issued via the submit path and commit through Raft consensus, while read-only operations (GetMetadata, PIRQuery) use the evaluate path, which bypasses ordering and block creation.

D. DISCUSSION

The results, summarized in Table 4, highlight a nearly linear growth trend in total latency with respect to $\log N$.

The *KeyGen* dominates at small scales due to keypair initialization and parameter validation in `Lattigo`. *Enc*

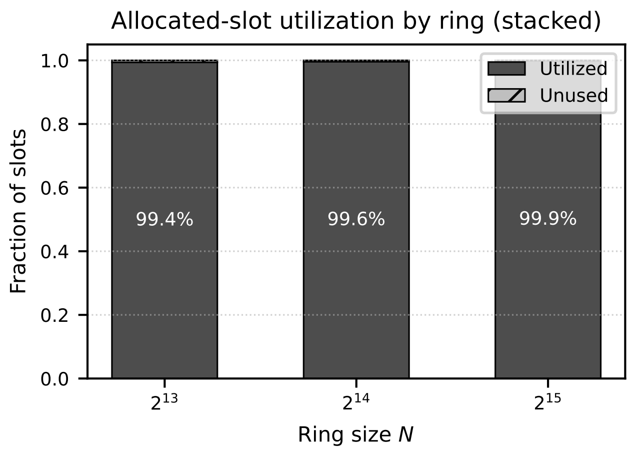


FIGURE 8. Allocated-slot utilization by ring. Each bar shows the fraction of utilized versus unutilized slots in m_{DB}.

TABLE 4. 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

and *Eval* exhibit moderate scaling with ring size, reflecting the $O(N \log N)$ cost of NTT-based polynomial arithmetic. *Dec* remains lightweight, averaging under 16 ms even for $N = 2^{15}$, confirming that decryption and slot extraction overheads are negligible. Overall, end-to-end query latency remains below 200 ms for all configurations, demonstrating that CPIR can be feasibly embedded within Fabric's evaluate transaction path.

Table 5 lists the corresponding averages for each $\log N$ configuration.

TABLE 5. Size of Main Cryptographic Artifacts (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

As expected, artifact sizes scale linearly with the ring dimension $N = 2^{\log N}$. 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). Base64 serialization introduces a $\sim$33% wire overhead but does not affect the underlying cryptographic representation.

`InitLedger` exhibits the largest execution time, reflecting the cost of initializing and inserting a large batch of encoded CTI records into the ledger world state. `PIRQuery` remains sublinear with respect to initialization, reflecting the cost of single-query PIR evaluation, which depends on homomorphic multiplication and slot decoding for the given ring dimension. `GetMetadata`, in contrast, executes quickly (< 10 ms on average) since it only retrieves pre-initialized ledger parameters without heavy computation. As expected, increasing $\log N$ leads to a monotonic rise in execution time for the PIR-related operations, with the most pronounced growth between $\log N = 14$ and $\log N = 15$.

Across all configurations, utilization remains above 98%, indicating near-optimal packing under the feasibility constraints of Fig. 4. Slight underutilization (e.g., $u \approx 0.984$) arises from rounding of slot windows to the nearest discrete size in $\mathcal{S} = 64, 128, 224, 256, 384, 512$, ensuring consistent record alignment and deterministic indexing.

V. CONCLUSION

The conclusion goes here.

APPENDIX

Appendixes, if needed, appear before the acknowledgment.

ACKNOWLEDGMENT

The preferred spelling of the word "acknowledgment" in American English is without an "e" after the "g." Use the singular heading even if you have many acknowledgments. Avoid expressions such as "One of us (S.B.A.) would like to thank . . . ." Instead, write "F. A. Author thanks . . . ." In most cases, sponsor and financial support acknowledgments are placed in the unnumbered footnote on the first page, not 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.

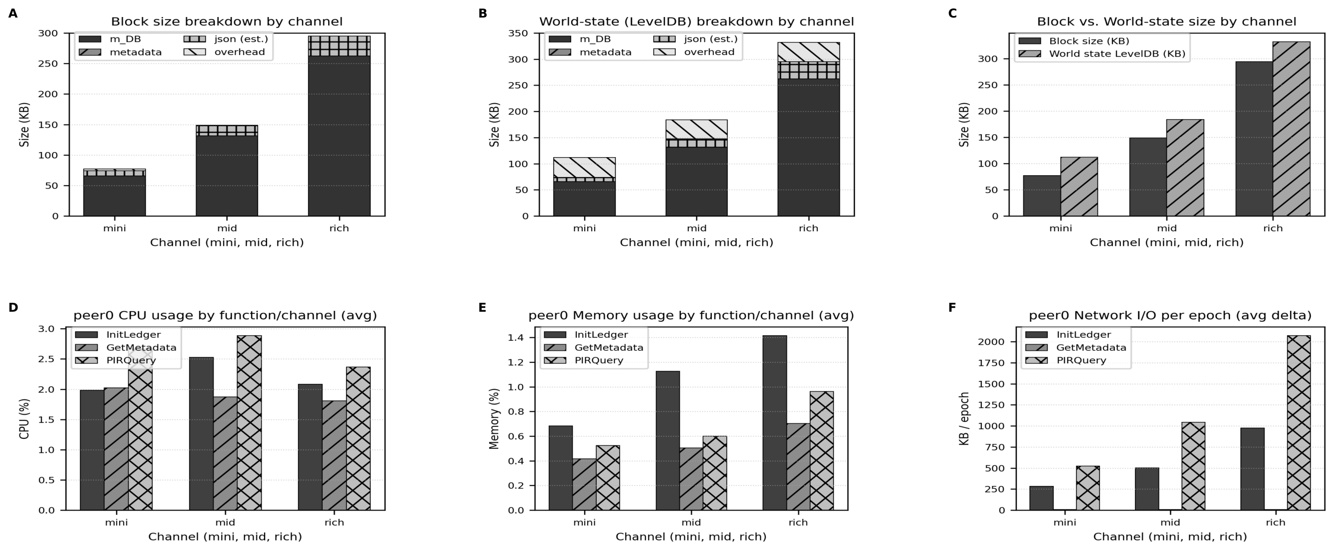


FIGURE 9. m_{DB} **construction from JSON to plaintext polynomial. Each record is serialized to bytes, mapped into a fixed slot window** $record_s$**, and packed into a coefficient vector** c**. The vector is then encoded as a BGV plaintext polynomial** m_{DB}**, which is stored in the Fabric world state.**

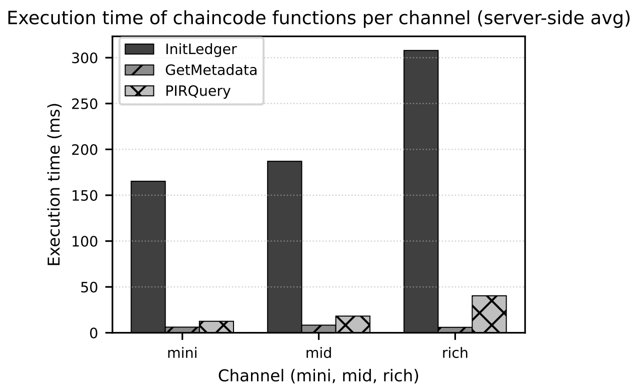


FIGURE 10. **Average chaincode execution time by function and ring size. Each bar represents the mean execution time over multiple epochs.**

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.