

HLFC³PIR: Embedding Computational Private Information Retrieval into Hyperledger Fabric Chaincode for Query Privacy

First A. Author¹, Fellow, IEEE, Second B. Author²,
and Third C. Author Jr.³, Member, IEEE

¹National Institute of Standards and Technology, Boulder, CO 80305 USA

²Department of Physics, Colorado State University, Fort Collins, CO 80523 USA

³Electrical 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* transactions reveal function arguments and read-sets to peers, creating privacy risks for organizations querying sensitive records. We address this gap by presenting the first full integration of Computational Private Information Retrieval (CPIR) within Fabric's chaincode layer. Our design encodes the ledger's key-value table as a packed plaintext polynomial and enables clients to submit encrypted selection vectors evaluated under the Brakerski–Gentry–Vaikuntanathan (BGV) homomorphic encryption scheme. Peers perform ciphertext–plaintext evaluation entirely inside chaincode and return encrypted results, preventing index leakage while preserving Fabric's endorsement and audit semantics. A prototype built with the Lattigo library achieves end-to-end query latency of ≈ 113 ms and peer-side evaluation below 45 ms for databases of up to 128 records, demonstrating practical performance within permissioned environments. Network measurements indicate ≈ 2 MB of peer traffic per query due to the external chaincode model, which can be halved under in-process deployment. Overall, the results confirm the feasibility of native CPIR execution in Hyperledger Fabric and establish a foundation for scalable extensions such as constant-time evaluation, sublinear and sharded PIR, and encrypted-database (*ct \times ct*) computation for richer private analytics.

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

I. INTRODUCTION

A. MOTIVATION AND PROBLEM STATEMENT

Blockchain [1] is a peer-to-peer network protocols that uses cryptographic primitives and consensus mechanism to create and maintain a distributed ledger of transactions, such as Ethereum [2] and Hyperledger Fabric [3].

According to the literature [4]–[7] such systems are widely adopted in various domains, such as Cyber Threat Intelligence (CTI) sharing, where multiple organizations exchange threat indicators and wish to keep their queries confidential. Blockchain guarantees, however, primarily cover data

integrity and auditability, not query privacy. For example, only recently PSE team in Ethereum foundation explicitly highlighted the need for private reads in addition to private writes and private proving [8].

In Hyperledger Fabric [3], 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 [9]. 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) [10] 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 [10]–[14] 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 [15]–[20] relying on cryptographic assumptions and homomorphic encryption to achieve privacy with a single server.

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. Thus, we focus on CPIR, which is more suitable for our threat model.

B. LITERATURE REVIEW

This section surveys prior work at the intersection of Private Information Retrieval (PIR) and distributed systems, with emphasis on smart-contract or chaincode-based implementations for query privacy across different domains, not limited to any specific domain or blockchain platform.

Xie et al. [21] design BIT-PIR, a PIR-based query scheme tailored to Bitcoin light clients: they restructure on-chain data and craft a storage format so lightweight wallets can privately fetch transaction-related items without Bloom-filter leakage. This work is pioneering in applying PIR to public blockchains, but its reliance on Bitcoin’s UTXO model limits generalizability to other platforms.

Chhabra et al. and Kumar introduced a series of PIR - blockchain frameworks across multiple domains, such as PRIBANI, MUDRA, DEBPIR, and BRON. PRIBANI [22] combines PIR with ring signatures and PEKS to enhance query anonymity in transaction systems. MUDRA [23] integrates dual-server Ring-PIR within Hyperledger Fabric to enable private access to healthcare records, while DEBPIR [24] embeds PIR and k-out-of-N OT in Fabric chaincode for privacy-preserving business data retrieval. BRON [25] extends this approach to human resource management, integrating PIR at the smart-contract layer with ZK components to support verifiable private record queries. Across these works, PIR consistently serves as a privacy-preserving layer for permissioned ledgers, yielding favorable throughput and latency. However, their evaluations offer limited cryptographic parameter details, making scalability and precise overhead less transparent.

Xiao et al. [26] propose Cloak, a privacy-preserving blockchain query framework that hides retrieval targets via

two-phase distributed query requests (division and aggregation). By splitting queries across nodes and aggregating partial answers, Cloak reduces exposure of which records are fetched and reports notable query-time improvements over naive single-server approaches. The technique is PIR-inspired rather than a direct PIR scheme; the paper focuses on distributed obfuscation and leaves a comparison against concrete CPIR/IT-PIR baselines (communication vs. server work vs. anonymity) for future study.

Peer2PIR [27] addresses privacy leakage in IPFS by enabling private peer routing, provider discovery, and content retrieval using PIR-backed protocols; the paper also surveys PIR choices in distributed systems and demonstrates reasonable overheads. While not a blockchain system per se, the work is highly relevant to decentralized Web stacks frequently coupled with blockchains, and its modular designs could be adapted to ledger-attached DHTs. A limitation is the absence of end-to-end integration with on-chain settlement/audit, so operational lessons for smart-contract pipelines are indirect.

Targeted gap. Given the above survey, we identify a gap in the literature: while several works embed PIR into permissioned blockchains, none provide a detailed, practical implementation of a lattice-based CPIR scheme directly within Hyperledger Fabric chaincode, with comprehensive parameter analysis and performance benchmarks. One recent survey outlined several open research questions in this area [28], such as: How can PIR be effectively integrated into blockchain to enhance privacy in decentralized applications?

Our work directly contributes to these directions by implementing a HE-based CPIR protocol in Fabric chaincode under *evaluate* calls, ensuring that endorsing peers cannot infer queried indices while also answering the open research question above.

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 under *evaluate* path and achieve practical performance under certain parametrization choices.

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 main-

tains an independent polynomial database and meta-data, 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.

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 performance. 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. Fabric’s multi-channel architecture [29] isolates ledgers across subgroups of organizations, limiting which participants observe which data. However, channel separation controls *who* sees a ledger, query intent remains visible to all endorsers of a channel.

Private Data Collections (PDC). PDCs [30] 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 [31], [32] 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 artifacts in world state that amount up to a 350 KB under typical parameters (see Section IV), LevelDB is sufficient and preferable for performance and simplicity, while CouchDB [33] remains an option to extend from 8 MB to 4 GB if needed.

Ledger history (blockchain log). According to the documentation [34], block size is constrained by the ordering service configuration. By default, the Fabric orderer limits the serialized payload to $AbsoluteMaxBytes = 10\text{ MB}$ (recommended under 49 MB given the gRPC ceiling of 100 MB), and typically aggregates up to $MaxMessageCount = 500$ transactions per block or $PreferredMaxBytes = 2\text{ MB}$. 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 [35] 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)

Homomorphic Encryption. Lattice-based homomorphic encryption schemes, such as Brakerski/ Fan-Vercauteren (BFV), Brakerski-Gentry-Vaikuntanathan (BGV), CKKS and TFHE [36]–[39] have emerged as foundational technologies for practical CPIR implementations.

We base our CPIR construction on the BGV scheme, a lattice-based homomorphic encryption based on the Ring Learning With Errors (RLWE) problem [40], which supports both addition and multiplication over ciphertexts. The 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 [41] and paper [42]. 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 .

CPIR Definition. 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.

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

- $KeyGen(\lambda) \rightarrow (pk, sk)$: On input the security parameter λ , output a public key pk and a secret key sk .
- $Enc_{pk}(\hat{v}_i) \rightarrow ct_q$: 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 .
- $Eval(ct_q, m_{DB}) \rightarrow ct_r$: Given ct_q and the plaintext polynomial database $m_{DB} \in R_T$, homomorphically evaluate the product to obtain an encrypted response $ct_r \in R_Q$.
- $Dec_{sk}(ct_r) \rightarrow d_i$: 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 $i \in [n]$,

$$Dec_{sk}(Eval(Enc_{pk}(\hat{v}_i), m_{DB})) = d_i.$$

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 CPIR into Hyperledger Fabric chaincode using the BGV [37] 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.

TABLE 1. Notation

Symbol	Description
λ	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)

- **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:

- **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.

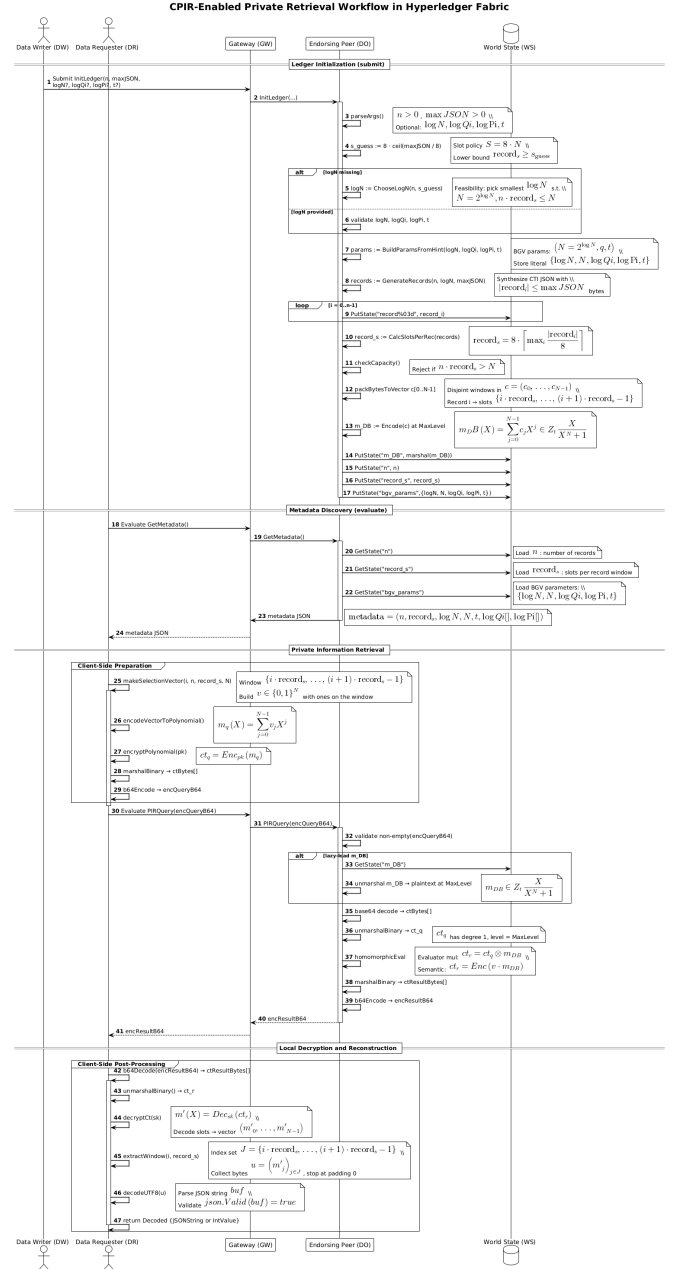


FIGURE 1. "Workflow. *DW* initializes the ledger via *GW*, which triggers chaincode on endorsing peers (*DO*). *DO* executes the protocol and persists state in world state (m_{DB} , metadata, JSON records). *DR* later obtains metadata, submits $ct_q = Enc_{pk}(v_i)$, *DO* evaluates $ct_r = Eval(ct_q,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

- 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 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 $c = (c_0, c_1, \dots, c_{N-1})$, 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 $\lfloor 8192/224 \rfloor = 36$ 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:

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

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

$$\text{Rich: } record_s \geq record_b + 32 + 64 + 15,$$

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:

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

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

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

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 $s \geq record_{\mu, 13} \approx 128 \Rightarrow$ 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 $s \geq record_{\mu, 14} \approx 224 \Rightarrow$ smallest candidate is 224.
- $\text{Disc}(s)$ allows 224, 256, ... but excludes smaller windows.
- $\text{Cap}(16384, 224, n)$ gives $n \leq \lfloor 16384/224 \rfloor = 73$.

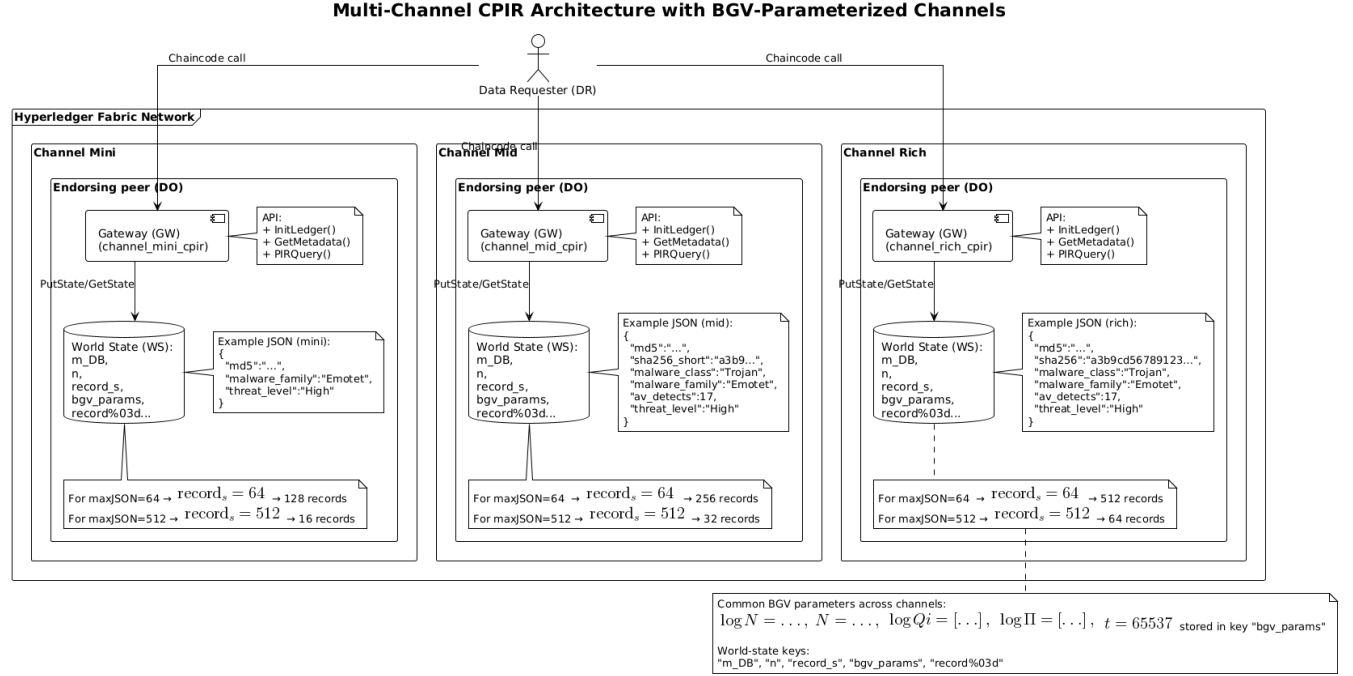


FIGURE 4. Multi-channel CPIR architecture. Each channel instantiates a separate CPIR chaincode and maintains its own m_{DB} polynomial, parameterized by $\log N$. This allows compact, mid-size, and rich CTI records to coexist under the same Fabric network.

elDB 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 *GW* (*submit*) with inputs $(n, record_s^{DW})$ and an optional hint $(\log N, \log Q_i, \log P_i, T)$. Here $record_s^{DW}$ 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 $record_s^{GW} = 8 \cdot \lceil record_s^{DW} / 8 \rceil$. If $\log N$ is absent, the smallest feasible $\log N$ is chosen such that $\text{Cap}(N, record_s^{GW}, n)$ holds. BGV parameters $\{\log N, N, \log Q_i, \log P_i, T\}$ are constructed and stored.
- 3) **DO prepares records.** Records $D = \{d_0, \dots, d_{n-1}\}$ are ingested or synthesized with $|d_i| \leq record_s^{DW}$. 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 $\text{Min}(\log N, record_s)$, $\text{Disc}(record_s)$, $\text{Cap}(N, record_s, n)$.

- 4) **Pack and persist.** Each d_i is placed in a disjoint window $J_i = \{i \cdot record_s, \dots, (i+1)record_s - 1\}$ 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 $(n, record_s, \log N, N, T, \log Q_i, \log P_i)$. This enables reconstruction of the cryptographic context.
- 6) **DR instantiates crypto context.** From metadata, *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 *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 $record_s = 1$).

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.

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

```

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:  $b \leftarrow \text{byte array}$  // init empty buffer for record
7: for  $j \in J_i$  do
8:   if  $u[j] = 0$  then
9:     break // stop at padding zero
10:  end if
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  $d_i$ 

```

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).

Our implementation employs the *Lattigo* v6 library [41] as the homomorphic encryption backend. Lattigo provides a Go-native implementation of the BGV scheme [37], 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 [43]. 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} .

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: (A) block size breakdown, (B) world-state (LevelDB) breakdown, (C) block vs. world-

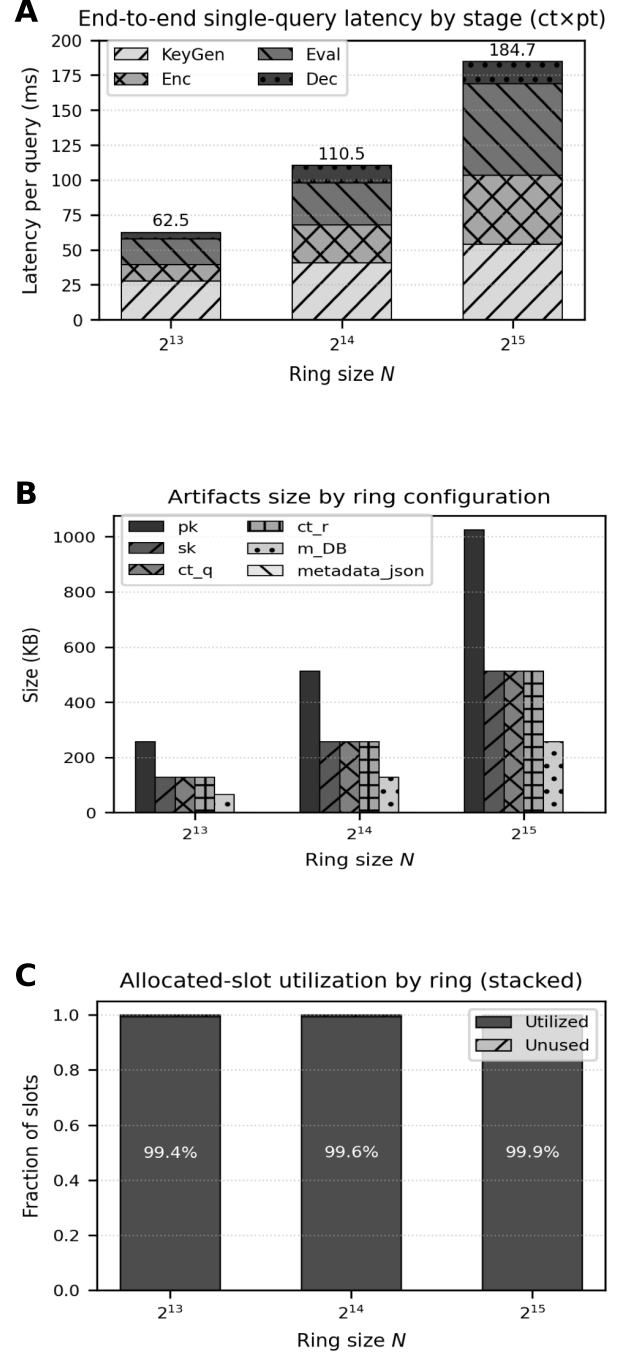


FIGURE 5. Cryptographic performance of the BGV-based CPIR system: (A) latency by algorithmic stage, (B) artifact size by ring configuration, and (C) allocated-slot utilization.

state size, (D) peer CPU utilization, (E) peer memory utilization, and (F) peer network I/O.

Chaincode Execution Timings. Figure 7 and Table 6 summarize the average *server-side execution time* of the main chaincode functions across different ring sizes.

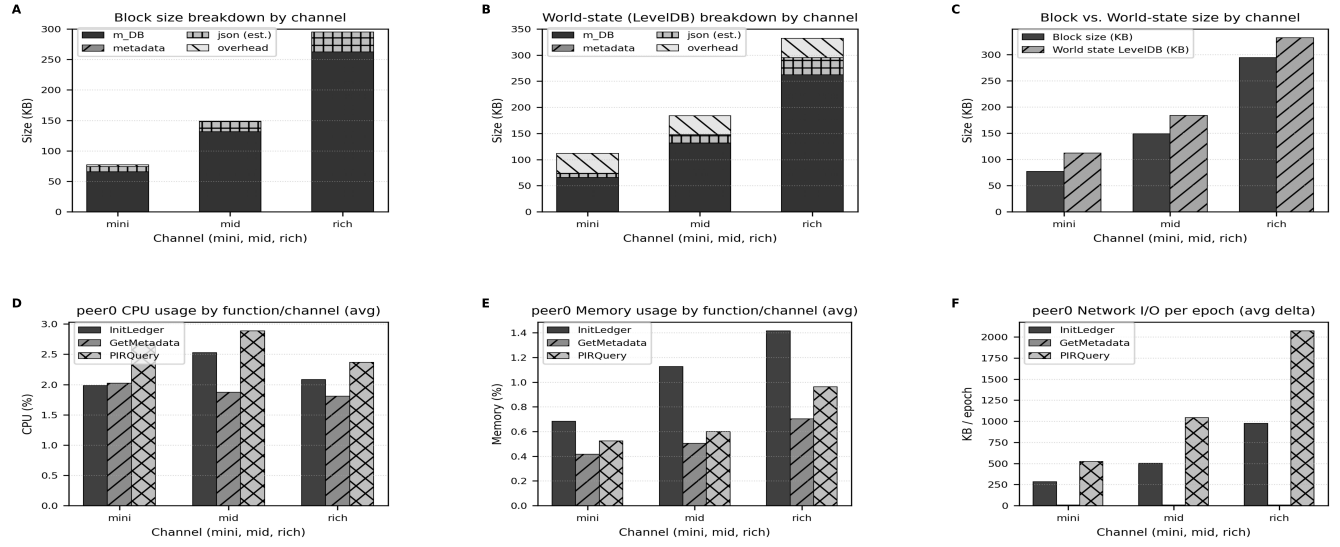


FIGURE 6. Blockchain-side evaluation of the CPIR system across three Fabric channels. (A–C) Storage-level metrics: block and world-state composition. (D–F) Peer-level resource utilization: CPU, memory, and network I/O per function.

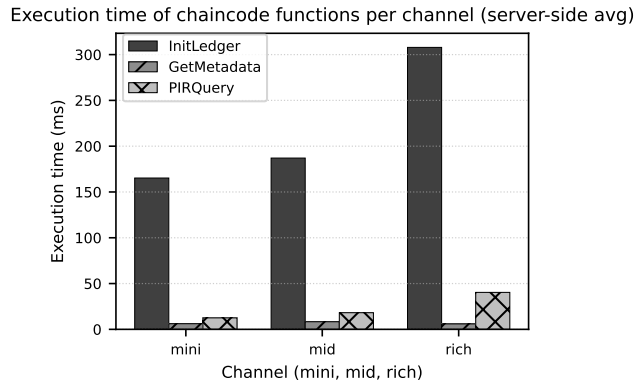


FIGURE 7. Average chaincode execution time by function and ring size. Each bar represents the mean execution time over multiple epochs.

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.

D. OVERALL SYSTEM PERFORMANCE

Table 3 lists the measured execution time of core cryptographic operations (*KeyGen*, *Enc*, *Eval*, and *Dec*) for each evaluated ring dimension $\log N$. Table 4 summarizes the corresponding sizes of primary cryptographic artifacts, including public and secret keys, ciphertexts (ct_q , ct_r), the encoded plaintext database m_{DB} , and auxiliary metadata. On-chain network behavior is reported in Table 5, which presents the average peer-side network I/O per *PIRQuery*

transaction across the three channel configurations. Measured chaincode execution times are provided in Table 6, for *InitLedger*, *GetMetadata*, and *PIRQuery* functions under identical peer conditions. Additionally, Figure 8 illustrates the peer- and client-side execution during a PIR query

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

TABLE 4. 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

TABLE 5. Peer network I/O per *PIRQuery* (avg, KB/tx)

Channel	$\log N$	NET I/O (KB/tx)
mini	13	524.62
mid	14	1042.09
rich	15	2072.88

TABLE 6. 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

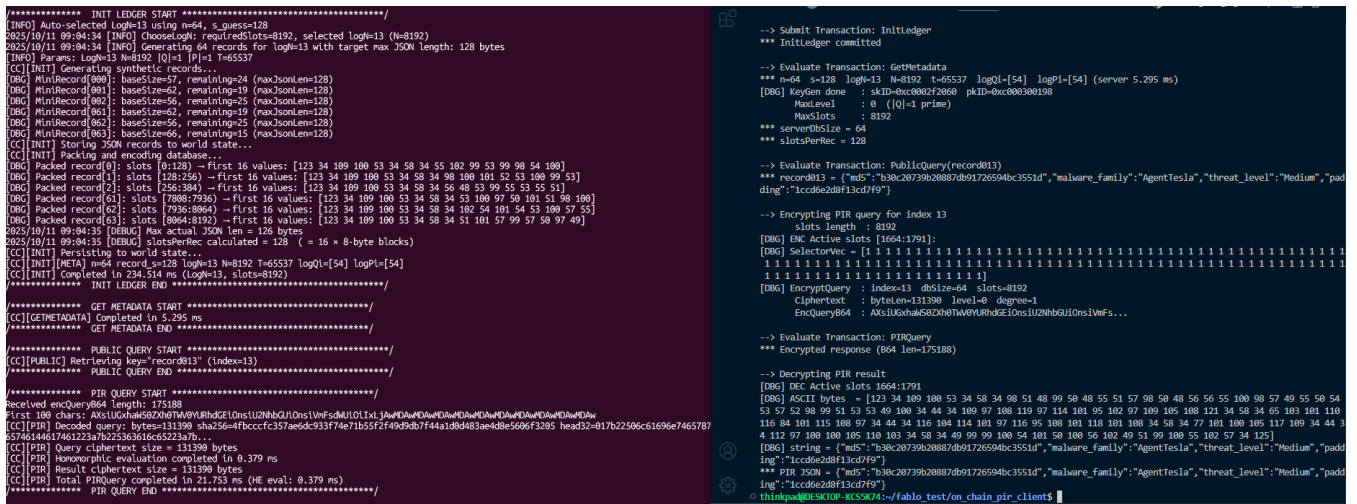


FIGURE 8. Detailed execution logs illustrating the privacy-preserving query workflow. Left: peer-side chaincode logs showing InitLedger, GetMetadata, and PIR evaluation with ciphertext-text multiplication. Right: client-side logs showing metadata retrieval, query encryption, PIR query invocation, and decryption of the result.

End-to-End Workflow Summary. Based on the data in Table 3 and Table 6, we now to summarize two main operational workflows in the system, involving DW and DR , while DO is implicitly involved as the executing peer during private queries.

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

TABLE 7. End-to-End Performance Analysis (ms)

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 ≈ 113 ms on average.

Batched Queries. Among our results, for the ring size $\log N=15$ configuration, issuing 100/1000/10000 private queries incurs $\approx 207\text{MB}$ / 2.07 GB / 20.7 GB of peer-side network I/O with an associated chaincode evaluation time of $\approx 0.07/0.67/6.73$ minutes, respectively, when using our protocol, as shown in Table 8 and Figure 9.

TABLE 8. Projected cost for batched PIR queries at $\log N=15$ (peer perspective)

# Transactions	Total Bandwidth (MB)	Total Time (min)
100	207.3	0.07
1000	2073.0	0.67
10000	20730.0	6.73

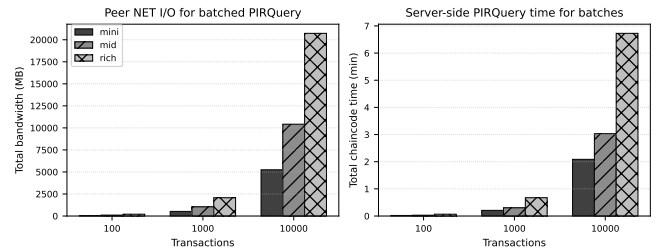


FIGURE 9. Projected cost for batched PIR queries at $\log N=15$ (peer perspective)

Remark (on network I/O measurement). The network I/O values in Table 5 and Table 8 were obtained from *docker stats* snapshots collected for the *peer0.org1.example.com* and *orderer0.group1.orderer.example.com* containers during each benchmark epoch. For every execution of *PIRQuery*, the benchmarking script recorded cumulative network statistics (*NET I/O*) before and after the transaction, and the Python post-processing pipeline computed per-epoch deltas ($\text{bytes}_{\text{in/out}}(t) - \text{bytes}_{\text{in/out}}(t-1)$) to obtain the mean transfer volume per function.

TABLE 9. Comparison with existing blockchain privacy frameworks

Work	Permissioned / Permissionless	Privacy / Technique	Chaincode / Off-chain	Evaluate / Submit Phase	World State / DB Size	E2E Query / Time (ms)
[21]	-	IT-PIR + ZKP	Off-chain service	Off-chain	-	-
[22]	Permissioned	CPIR	Chaincode orchestration	Evaluate	-	-
[23]	Permissioned	CPIR + Differential Privacy	Chaincode + off-chain	Evaluate	-	-
[24]	Permissioned	IT-PIR (OT-based)	Chaincode + off-chain peers	Evaluate	-	-
[25]	Permissioned	CPIR + ZKP	Smart contract	Submit	-	-
[26]	-	Distributed IT-PIR	Hybrid contracts	Submit	-	-
[27]	-	IT-PIR + P2P privacy	Off-chain IPFS layer	Off-chain	-	-
Ours	Permissioned	CPIR (lattice-based, BGV)	Chaincode	Evaluate	Yes	Yes

V. DISCUSSION

A. FUTURE WORK AND OUTLOOK

While this work demonstrates the feasibility of integrating CPIR directly into Hyperledger Fabric’s chaincode for privacy-preserving queries, several challenges remain in performance, scalability, and deployment. Future research should address (i) side-channel resilience through constant-time chaincode evaluation and standardized ciphertext serialization to mitigate timing and size-based leakages. (ii) Scaling beyond moderate database sizes will require sublinear or sharded variants—such as preprocessing-based PIR, offline hinting, or multi-stage initialization—to reduce online complexity. Moreover, (iii) extending the current ciphertext–plaintext ($ct \times pt$) evaluation to fully encrypted-database ($ct \times ct$) computation would enable richer on-chain analytics under encryption, though it demands relinearization, rotation, and modulus-switching capabilities within Fabric’s resource limits. (iv) Dynamic ledger updates (*addRecord*) operation would allow incremental growth without full reinitialization, raising questions around re-encoding and key management. (v) Selective field-level retrieval through adaptive packing could enable varying length records and partial access control. (vi) Support for dual-mode operation—covering both world-state and historical ledger data—would enable privacy-preserving analytics and provenance tracking. (vii) Deployment optimization is another key area. In our setup, the chaincode operated in *development mode* as an external service container, which introduced duplicated gRPC transmissions for each ciphertext (ct_q , ct_r). During every *PIRQuery*, the transaction traversed two network hops—(1) client \leftrightarrow peer and (2) peer \leftrightarrow chaincode—causing each ciphertext to be sent and received twice, once per link. This resulted in roughly four full transmissions per query, explaining the measured aggregate of ≈ 2072 KB for $\log N=15$, consistent with the theoretical estimate 4×512 KB plus gRPC/TLS overhead. If the same chaincode were executed *in-process* within the peer container, only the client–peer exchange would occur, reducing the network footprint to approximately $2 \times 512=1024$ KB per query (about 10 GB for 10,000 queries).

B. RELATED WORK COMPARISON

Table 9 contrasts representative blockchain privacy frameworks that integrate Private Information Retrieval (PIR) or related cryptographic mechanisms to achieve query privacy across different domains. The comparison focuses on six key aspects: (i) whether the system is designed for permissioned or permissionless blockchains, (ii) the underlying privacy primitive (CPIR, IT-PIR, or hybrid), (iii) whether privacy computation occurs on-chain (via smart contracts or chaincode) or off-chain, (iv) the transaction phase in which privacy logic executes (*evaluate* vs. *submit*), (v) typical database or world-state size, and (vi) reported end-to-end (E2E) query latency when available.

VI. CONCLUSION

This work demonstrated that computational private information retrieval (CPIR) can be embedded natively within Hyperledger Fabric’s chaincode to enable privacy-preserving queries without modifying its consensus or ledger logic. Using the BGV homomorphic encryption scheme, query indices remain hidden from endorsing peers while retaining full endorsement and audit semantics. The prototype achieves end-to-end query latency of ≈ 113 ms and peer-side evaluation below 45 ms for databases up to 128 records, with key generation, encryption, and evaluation completing within 55–65 ms.

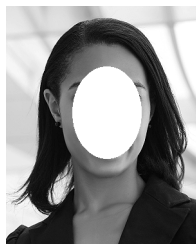
Peer network monitoring indicated an average of ≈ 2 MB per private query due to the dual-hop gRPC communication between client, peer, and external chaincode—equivalent to ≈ 207 MB, 2.07 GB, and 20.7 GB for 100, 1000, and 10,000 queries, respectively. These costs can be halved by deploying the chaincode in-process within the peer, eliminating the hop and bringing cost down to ≈ 1 MB per query.

Overall, the results confirm the practicality of executing CPIR directly within a permissioned blockchain framework, achieving millisecond-scale query performance while preserving modularity and auditability. Future directions include constant-time evaluation and standardized ciphertext serialization for side-channel resilience, sublinear or sharded CPIR for scalability, and fully encrypted ($ct \times ct$) query processing for richer privacy-preserving analytics in distributed ledgers.

REFERENCES

- [1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System."
- [2] D. G. Wood, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER."
- [3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018, event-place: Porto, Portugal. [Online]. Available: <https://doi.org/10.1145/3190508.3190538>
- [4] K. Dunnett, S. Pal, G. D. Putra, Z. Jadidi, and R. Jurdak, "A Trusted, Verifiable and Differential Cyber Threat Intelligence Sharing Framework using Blockchain," in *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Dec. 2022, pp. 1107–1114, iSSN: 2324-9013. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/10063731>
- [5] Y. Allouche, N. Tapas, F. Longo, A. Shabtai, and Y. Wolfsthal, "TRADE: TRusted Anonymous Data Exchange: Threat Sharing Using Blockchain Technology," Mar. 2021, arXiv:2103.13158 [cs]. [Online]. Available: <http://arxiv.org/abs/2103.13158>
- [6] S. Gong and C. Lee, "BLOCIS: Blockchain-Based Cyber Threat Intelligence Sharing Framework for Sybil-Resistance," *Electronics*, vol. 9, no. 3, p. 521, Mar. 2020, number: 3 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: <https://www.mdpi.com/2079-9292/9/3/521>
- [7] P. Huff and Q. Li, "A Distributed Ledger for Non-attributable Cyber Threat Intelligence Exchange," in *Security and Privacy in Communication Networks*, J. Garcia-Alfaro, S. Li, R. Poovendran, H. Debar, and M. Yung, Eds. Cham: Springer International Publishing, 2021, pp. 164–184.
- [8] P. S. o. Ethereum, "PSE Roadmap: 2025 and Beyond | PSE." [Online]. Available: <https://pse.dev/blog/pse-roadmap-2025>
- [9] "Fabric Gateway — hyperledger-fabricdocs main documentation." [Online]. Available: <https://hlf.readthedocs.io/en/stable/gateway.html>
- [10] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, no. 6, pp. 965–981, Nov. 1998, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: <https://doi.org/10.1145/293347.293350>
- [11] D. Demmler, A. Herzberg, and T. Schneider, "RAID-PIR: Practical Multi-Server PIR," in *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security*, ser. CCSW '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 45–56. [Online]. Available: <https://doi.org/10.1145/2664168.2664181>
- [12] I. Goldberg, "Improving the Robustness of Private Information Retrieval," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, May 2007, pp. 131–148, iSSN: 2375-1207. [Online]. Available: <https://ieeexplore.ieee.org/document/4223220>
- [13] A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond, "Breaking the $O(n/\sup{1/(2k-1)})$ barrier for information-theoretic Private Information Retrieval," in *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002. *Proceedings.*, 2002, pp. 261–270.
- [14] C. Devet, I. Goldberg, and N. Heninger, "Optimally Robust Private Information Retrieval," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 269–283. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/devet>
- [15] C. Cachin, S. Micali, and M. Stadler, "Computationally Private Information Retrieval with Polylogarithmic Communication," in *Advances in Cryptology — EUROCRYPT '99*, J. Stern, Ed. Berlin, Heidelberg: Springer, 1999, pp. 402–414.
- [16] Y.-C. Chang, "Single Database Private Information Retrieval with Logarithmic Communication," in *Information Security and Privacy*, H. Wang, J. Pieprzyk, and V. Varadarajan, Eds. Berlin, Heidelberg: Springer, 2004, pp. 50–61.
- [17] C. Gentry and Z. Ramzan, "Single-Database Private Information Retrieval with Constant Communication Rate," in *Automata, Languages and Programming*, L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, Eds. Berlin, Heidelberg: Springer, 2005, pp. 803–815.
- [18] Z. Brakerski and V. Vaikuntanathan, "Efficient Fully Homomorphic Encryption from (Standard) LWE," in *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, Oct. 2011, pp. 97–106, iSSN: 0272-5428. [Online]. Available: <https://ieeexplore.ieee.org/document/6108154>
- [19] C. Dong and L. Chen, "A Fast Single Server Private Information Retrieval Protocol with Low Communication Cost," in *Computer Security - ESORICS 2014*, M. Kutyłowski and J. Vaidya, Eds. Cham: Springer International Publishing, 2014, pp. 380–399.
- [20] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, "XPRI : Private Information Retrieval for Everyone," *Proceedings on Privacy Enhancing Technologies*, 2016. [Online]. Available: <https://petsymposium.org/popets/2016/popets-2016-0010.php>
- [21] Y. Xie, Q. Wang, R. Li, C. Zhang, and L. Wei, "Private Transaction Retrieval for Lightweight Bitcoin Clients," *IEEE Transactions on Services Computing*, vol. 16, no. 5, pp. 3590–3603, 2023.
- [22] A. Chhabra, R. Saha, and G. Kumar, "PRIBANI: a privacy-ensured framework for blockchain transactions with information retrieval," *Cluster Computing*, vol. 27, no. 6, pp. 8189–8206, Sep. 2024. [Online]. Available: <https://doi.org/10.1007/s10586-024-04387-6>
- [23] A. Chhabra, R. Saha, G. Kumar, and T. H. Kim, "MUDRA: A Multi-dimensional Privacy Attainment Framework for Healthcare Blockchain Transactions," *IEEE Transactions on Consumer Electronics*, pp. 1–1, 2025. [Online]. Available: <https://ieeexplore.ieee.org/document/10897831>
- [24] G. Kumar, R. Saha, M. Conti, and T. H. Kim, "DEBPIR: enhancing information privacy in decentralized business modeling," *Complex & Intelligent Systems*, vol. 11, no. 7, p. 290, May 2025. [Online]. Available: <https://doi.org/10.1007/s40747-025-01868-y>
- [25] G. Kumar, R. Saha, M. Gupta, and T. H. Kim, "BRON: A blockchained framework for privacy information retrieval in human resource management," *Heliyon*, vol. 10, no. 13, Jul. 2024, publisher: Elsevier. [Online]. Available: [https://www.cell.com/heliyon/abstract/S2405-8440\(24\)09424-6](https://www.cell.com/heliyon/abstract/S2405-8440(24)09424-6)
- [26] J. Xiao, J. Chang, L. Lin, B. Li, X. Dai, Z. Xiong, K.-K. R. Choo, K. Gai, and H. Jin, "Cloak: Hiding Retrieval Information in Blockchain Systems via Distributed Query Requests," *IEEE Transactions on Services Computing*, vol. 17, no. 06, pp. 3213–3226, Nov. 2024, place: Los Alamitos, CA, USA Publisher: IEEE Computer Society. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/TSC.2024.3411450>
- [27] M. Mazmudar, S. Veitch, and R. A. Mahdavi, "Peer2PIR: Private Queries for IPFS," 2024, _eprint: 2405.17307. [Online]. Available: <https://arxiv.org/abs/2405.17307>
- [28] A. Chhabra, R. Saha, G. Kumar, and T.-H. Kim, "Navigating the Maze: Exploring Blockchain Privacy and Its Information Retrieval," *IEEE Access*, vol. 12, pp. 32 089–32 110, 2024.
- [29] "Channels — Hyperledger Fabric Docs main documentation." [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/latest/channels.html>
- [30] "Private data — Hyperledger Fabric Docs main documentation." [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/latest/private-data/private-data.html>
- [31] "hyperledger/fabric-private-chaincode: FPC enables Confidential Chaincode Execution for Hyperledger Fabric using Intel SGX." [Online]. Available: <https://github.com/hyperledger/fabric-private-chaincode>
- [32] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, "Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric," 2018, _eprint: 1805.08541. [Online]. Available: <https://arxiv.org/abs/1805.08541>
- [33] "CouchDB as the State Database — Hyperledger Fabric Docs main documentation." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/couchdb_as_state_database.html
- [34] "Performance considerations — Hyperledger Fabric Docs main documentation." [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-2.5/performance.html>
- [35] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System," 2014, _eprint: 1407.3561. [Online]. Available: <https://arxiv.org/abs/1407.3561>
- [36] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," 2012, publication info: Published elsewhere. Unknown where it was published. [Online]. Available: <https://eprint.iacr.org/2012/144>

- [37] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “Fully Homomorphic Encryption without Bootstrapping,” 2011, publication info: Published elsewhere. Unknown where it was published. [Online]. Available: <https://eprint.iacr.org/2011/277>
- [38] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic Encryption for Arithmetic of Approximate Numbers,” in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.
- [39] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “TFHE: Fast Fully Homomorphic Encryption over the Torus,” 2018, published: Cryptology ePrint Archive, Paper 2018/421. [Online]. Available: <https://eprint.iacr.org/2018/421>
- [40] C. Peikert, “Lattice Cryptography for the Internet,” in *Post-Quantum Cryptography*, M. Mosca, Ed. Cham: Springer International Publishing, 2014, pp. 197–219.
- [41] “Lattigo v6,” Aug. 2024, published: Online: <https://github.com/tuneinsight/lattigo>. [Online]. Available: <https://github.com/tuneinsight/lattigo>
- [42] C. Mouchet, J. Troncoso-Pastoriza, J.-P. Bossuat, and J.-P. Hubaux, “Multiparty Homomorphic Encryption from Ring-Learning-With-Errors,” 2020, published: Cryptology ePrint Archive, Paper 2020/304. [Online]. Available: <https://eprint.iacr.org/2020/304>
- [43] “client package - github.com/hyperledger/fabric-gateway/pkg/client - Go Packages.” [Online]. Available: <https://pkg.go.dev/github.com/hyperledger/fabric-gateway/pkg/client>



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.