CVPIR: Verifiable PIR with Efficient Online Communication Complexity

Seung Geol Choi[1], Alexis Galan[a,2], Daniel S. Roche[b,1] and Mayank Varia[c,3]

[1] United States Naval Academy, Computer Science Department, Annapolis, Maryland, U.S.A.[d]
[2] Université Grenoble Alpes, Laboratoire Jean Kuntzmann, Grenoble, France
[3] Boston University, Faculty of Computing & Data Sciences, Boston, Massachusetts, U.S.A.

Abstract. Verifiable private information retrieval (vPIR) enables clients to query a database server with query privacy and correctness, and without selective failures. This work contributes *Compact and Verified PIR* (CVPIR), the first vPIR construction with single server whose communication complexity grows logarithmically in the database size N. We implement CVPIR using OpenFHE and show that it outperforms prior works (with $O(\sqrt{N})$ communication) for databases greater than 32 GB, with around 5× improvement for a 1 TB database. We also contribute the first simulation-secure security analysis of verifiable PIR: we specify an ideal functionality $\mathcal{F}_{\text{PIR}}$, show how it also satisfies the definition of a proof of retrievability, and provide a UC security analysis that CVPIR realizes $\mathcal{F}_{\text{PIR}}$.

1 Introduction

Private Information Retrieval (PIR) allows a client to query for a database record without revealing the search query. As a result, PIR hides both the actions of individual clients and the access patterns (i.e., popularity) of individual records. PIR has been a topic of study for 30 years [CGKS95], and it has found applications in certificate transparency [Rya14, LG15, HHC+23], checking for credential compromise [TPY+19, ALP+21], private contact discovery [KRS+19, HSW23], software updates [Cap13, AIVG22], and more. This paper focuses on the setting of PIR with a single server and computational security.

Verifiable PIR. There are two waves of *verifiable PIR* (vPIR) constructions that add integrity guarantees on top of PIR's query privacy guarantee.

The first guarantee is that the client can verify that the server correctly responds to its query [ZS14, BDKP22]. Constructions that satisfy this correctness guarantee were introduced in the multi-server setting by Zhang and Safavi-Naini [ZS14] and in the single-server setting by Ben-David et al. [BDKP22]. The single server setting requires a preprocessing stage in which the server broadcasts a commitment to its database to all clients, so that the "correct response" is well-defined.

E-mail: choi@usna.edu (Seung Geol Choi), alexis.galan@univ-grenoble-aples.fr (Alexis Galan), roche@usna.edu (Daniel S. Roche), varia@bu.edu (Mayank Varia)

[a]The authors thank the IDEX for funding A.G. mobility grant.
[b]This work was started while the author was on sabbatical at Boston University.
[c]Work supported by the DARPA SIEVE program under Agreement No. HR00112020021 and by the National Science Foundation under Grant No. 2209194.

[d]The views expressed in this document are those of the authors and do not reflect the official policy or position of the U.S. Naval Academy, Department of the Navy, the Department of Defense, or the U.S. Government.

Recent vPIR constructions add a second, stronger guarantee that the client can *also* verify that the server honestly responded to their query q using the correct database. As a result, the client is assured that the server's response also would have been correct even if the client had queried a different location q'. Hence, the server cannot run a *selective failure attack* [AB25] to learn about the client's query by (for instance) mauling some records of the database and waiting to see if the client complains about an incorrect response. Colombo et al. [CNC+23] contributed the first vPIR construction satisfying this stronger guarantee, albeit with the assumption that the server is honest during preprocessing. Recent works have removed this assumption [WLZ+23, dCL24, DT24, RLP25].

Communication complexity. There is an interesting dichotomy between the two waves of vPIR constructions, in terms of the communication complexity required between the server and querying client.

- Constructions that achieve only the weaker guarantee [BDKP22] or operate in the multi-server setting [ZKN+25] have communication complexity $O_\lambda(\mathsf{polylog}(N))$. Here, N denotes the number of entries in the database, and the notation O_λ indicates that the asymptotic dependency on the soundness parameter λ is omitted.
- Constructions that achieve the selective failure resistance guarantee in the single server setting have communication complexity $\Omega_\lambda(\sqrt{N})$ [CNC+23, dCL24, DT24, RLP25].

This raises the following questions:

1. Is it possible to construct the stronger form of verifiable PIR with communication complexity $O_\lambda(\mathsf{polylog}(N))$ in the single server setting?
2. If so, is the communication between the server and querying client concretely smaller than prior works, for reasonable database sizes in the gigabyte to terabyte range?

1.1 This Work

In this work, we answer both questions in the affirmative. We contribute a new verifiable PIR construction in the single server setting called *Compact and Verified Private Information Retrieval* (CVPIR) with $\log(N)$ communication complexity (Figs. 11 to 12). (Note that for the rest of this work, all references to verifiable PIR always mean the stronger notion that disallows selective failures.) Additionally, we implement CVPIR and show that its asymptotic advantage 'kicks in' and produces concretely lower communication for realistic database sizes (Fig. 17).

Along the way, we also make three other innovations in verifiable PIR: a composable definition, a reduction to proofs of retrievability, and the use of verifiable linear algebra. We describe each of these innovations in more detail below.

Composition. This work contributes, for the first time, an ideal functionality $\mathcal{F}_{\text{PIR}}$ for single server PIR (in Fig. 1). By contrast, all prior works on verifiable PIR use game-based security definitions, but Alon and Beimel observe in their recent work on multi-server PIR [AB25] that "40 years of experience in studying secure multiparty protocols taught us that defining the security of protocols by a list of required properties is problematic."

Our ideal functionality $\mathcal{F}_{\text{PIR}}$ includes three methods:

- a one-time server setup GetDigest to commit to the database,
- a per-client setup Register to receive parameters from the server, and
- an online Lookup for the client to make a query and receive a response.

We emphasize that *verifiability* is a crucial pre-requisite for an ideal functionality $\mathcal{F}_{\text{PIR}}$ for PIR to be meaningful. An ordinary PIR construction (that provides confidentiality but not integrity) has no meaningful ideal functionality since the server can return any response it wants in response to a client's query lookup.

We also contribute a simulation-based analysis of our vPIR construction π_{CVPIR} using Canetti's universally composable security framework [Can01]. Since PIR is a common building block used in security-critical applications like credential compromise and private contact discovery, a modular and composable analysis ensures that vPIR's guarantees (correctness and resistance to selective failure) continue to hold even when the vPIR is incorporated into a larger protocol.

Connection to PoR. We also relate vPIR to another crypto primitive: Proofs of Retrievability (PoR), which allow a client to store data remotely and to efficiently ensure that the entirety of that data is still intact on the server. In this work, we observe that any vPIR also provides the guarantees of a PoR—concretely, a client can query any entry in the database and verify that the server's response is correct. Because a vPIR withstands selective failure, the server must also have been able to respond correctly to lookups of all other records; hence, the server must know the entire database.

We make this connection explicit in our ideal functionality $\mathcal{F}_{\text{PIR}}$ by forcing the server to re-enter the entire database every time the client makes a Lookup request, and checking that it matches the database initially used in preprocessing (see Fig. 1 for details). In more detail, both vPIR and PoR require the existence of an extractor algorithm, so it suffices for our vPIR queries to serve as PoR audits and for the vPIR extractor to function as the PoR extractor. To perform the extraction, our universally composable security analysis (that π_{CVPIR} UC-realizes $\mathcal{F}_{\text{PIR}}$) uses the generic group model $\mathcal{G}$-oGG. We note that UC security analyses require *some* trusted setup [Can01], and the generic group model provides a convenient way for our simulator to extract the database required during online lookups from the commitment provided in the server setup phase.

Verifiable linear algebra. Finally, we provide a protocol π_{VLA} and functionality $\mathcal{F}_{\text{VLA}}$ for verifiable linear algebra; we use it both during precomputation and for a client to check that it receives the correct result to a query, and it may also be of independent interest. Generally, $\mathcal{F}_{\text{VLA}}$ provides a way to commit to a matrix $\mathbf{D}$ and later perform two kinds of operations:

- An interactive protocol VectMatProd, where a client submits a vector $\mathbf{u}$ and receives the vector-matrix product $\mathbf{u}^\intercal\mathbf{D}$, which is verified to be correct consistent with the commitment to $\mathbf{D}$, without revealing anything about $\mathbf{u}$ to the server
- A non-interactive check VerifyColumn to confirm that a given vector is in fact the i'th column of the committed matrix $\mathbf{D}$ provided a short *tag* structure t_i.

One of the logistical challenges in our design stems from the need to connect linear algebra operations in the FHE plaintext space of order p with cryptographic operations over an elliptic curve group of larger order q. See Sections 5 to 6 for details.

1.2 Contributions

In summary, we make four contributions in this work.

- We formally specify ideal functionalities for private information retrieval $\mathcal{F}_{\text{PIR}}$ that also serves as a proof of retrievability, for verifiable linear algebra $\mathcal{F}_{\text{VLA}}$, and for polynomial commitments $\mathcal{F}_{\text{TPC}}$ (Section 4).
- We construct the first verifiable PIR protocol π_{CVPIR} with logarithmic communication complexity (Section 7). We also give the first simulation-based security analysis of single server PIR using universally composable security (Appendix A), in order to show that π_{CVPIR} UC-realizes $\mathcal{F}_{\text{PIR}}$ in the generic group model (Appendix C).
- Our protocol contains two subroutines that may be of independent interest: a real-ideal specification of KZG polynomial commitments π_{TPC} (Section 5), and verifiable two-party computation of matrix-vector operations π_{VLA} (Section 6).

Table 1: Comparison of verifiable PIR constructions in the single server setting. In this table: N denotes the database size, all runtimes and communication are asymptotic, and they ignore polylogarithmic factors and dependencies on the security parameter λ. For instance, '1' means that the property scales as $O_\lambda(\mathsf{polylog}(N))$.

	Client preprocessing			Online time		Online
Scheme	Time	Comm	Storage	Client	Server	Comm
Crust [WLZ+23]	N	N	$\sqrt{N}$	$\sqrt{N}$	$\sqrt{N}$	$\sqrt{N}$
Dietz and Tessaro [DT24]	N	$\sqrt{N}$	1	$\sqrt{N}$	N	$\sqrt{N}$
VeriSimplePIR [dCL24]	$\sqrt{N}$	$\sqrt{N}$	$\sqrt{N}$	$\sqrt{N}$	N	$\sqrt{N}$
Rathee et al. [RLP25]	$\sqrt{N}$	$\sqrt{N}$	$\sqrt{N}$	$\sqrt{N}$	N	$\sqrt{N}$
This work π_{CVPIR}	N	N	1	1	N	1

- We implement π_{CVPIR} and show that, for ∼gigabyte-sized databases, our concrete communication is lower than that of prior works (Section 8).

Our construction. We briefly summarize the design of our vPIR protocol π_{CVPIR} here and provide a longer overview in Section 2. We start from a Kushilevitz-Ostrovsky style PIR [KO97]: the server represents the N-bit database as a square matrix $\mathbf{D}$ of size $\sqrt{N} \times \sqrt{N}$ and broadcasts a vector commitment to $\mathbf{D}$. The client builds a one-hot vector $\mathbf{a}$ that equals 1 at the desired index and 0 elsewhere. The server computes the column vector $\mathbf{Da}$, and the client looks at the relevant index to find its desired database record. To protect query privacy, we use BFV homomorphic encryption [Bra12, FV12a] to encrypt the query vector $\mathbf{a}$ and evaluate the response vector $\mathbf{Da}$. However, this initial idea has two major issues: it has inefficient $O_\lambda(\sqrt{N})$ communication and it is unverifiable.

The efficiency fix is simple: make $\mathbf{D}$ a non-square matrix: specifically, a wide matrix with $\log(N)$ rows and $N/\log(N)$ columns, so that the response vector $\mathbf{Da}$ is of size $O_\lambda(\log N)$ and is efficient to communicate. The query vector $\mathbf{a}$ becomes much larger, so we compress this one-hot vector into an outer product $\mathbf{a} = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \cdots \otimes \begin{bmatrix}0\\1\end{bmatrix}$ of short vectors, where the order of 0s and 1s in each vector depends on the query index.

The verifiability fix is more complicated and, thanks to UC security, we design it in a modular fashion. Starting from the lowest level, we use polynomial commitments $\mathcal{F}_{\text{TPC}}$ to form a digest of the database, and assume that this commitment has been broadcast to all clients. Then, the client retrieves 'hints' from the server in a preprocessing step that uses verifiable linear algebra operations $\mathcal{F}_{\text{VLA}}$; by contrast, we emphasize that our protocol is stateless on the server side. At the highest level, our vPIR protocol makes both *verification queries* $\mathbf{v}$ and *actual queries* $\mathbf{a}$ and uses the preprocessing hints to check if the server's responses are valid. We are able to show (with overwhelming probability) that if the server answers all $\mathbf{v}$ queries correctly then it must answer an $\mathbf{a}$ query correctly too. As a result, either the client retrieves the correct answer, or it aborts—we emphasize that the latter option is not a *selective* failure because the client's action depends only on the server's response to the $\mathbf{v}$ queries and is independent of the actual query index $\mathbf{a}$.

1.3 Related Work

In this section, we compare this work with prior works in PIR, vPIR, and PoR.

PIR against malicious servers. Several early works have sought to strengthen PIR against malicious servers [ZS14, WZ18, ZWH21, BDKP22]. Some of them provide features that this work does not; for instance, Ben-David et al. [BDKP22] allow the server to

succinctly prove complex properties about the database. However, these works only ensure that a query is consistent with *some* valid database, rather than the *specific* database provided by the server during setup. Hence, these schemes are vulnerable to attacks where the server changes the database between queries or for different clients.

Verifiable PIR. In the single server setting, the Authenticated PIR protocol by Colombo et al. [CNC+23] defends against selective-failure attacks in the PIR setting and ensures consistency across queries, but it assumes that the server's initial digest is well-formed. Recent works achieve a more robust form of PIR (which we call vPIR) by removing this assumption. These works build on top of different existing PIR protocols: Wang et al. [WLZ+23] combine pseudorandom sets with PIANO [ZPZS24], Dietz and Tessaro [DT24] patch one of the two aPIR constructions, and de Castro and Lee [dCL24] and Rathee et al. [RLP25] both add verifiability to SimplePIR [HHC+23]. We compare the asymptotic complexity of these constructions in Table 1.

All prior works in the single server setting have a communication complexity that scales with the square root of the database size, whereas the vPIR scheme in this work has asymptotically and concretely lower communication. Conversely, our construction has more computation than prior works, with [WLZ+23] offering sublinear computation.

In the multi-server setting (with non-colluding servers), there are recent works that provide logarithmic communication [ZKN+25] and that offer simulation-based security for PIR [AB25]. This work provides both benefits in the single server setting.

Simulation-based PIR. Whereas PIR protocols have property- and game-based security definitions, Alon and Beimel [AB25] show how to define multi-server PIR with simulation-based security. The challenge here is in defining what the 'correct' database is. Alon and Beimel [AB25] provide full security in the honest majority setting (where the 'correct' database is defined by majority vote) and security up to selective failures in the dishonest majority setting. By contrast, this work provides the first simulation-based (in fact, UC) security analysis of PIR in the single server setting, using both verifiable PIR and a succinct commitment of the database that is assumed to be broadcast to all clients in preprocessing.

Proof of Retrievability. PoR allows a client to verify that the server is maintaining a copy of the database (e.g., [ABC+07, JK07, SSP13, ADdJ+21]). The central security guarantee of a PoR protocol is retrievability: informally, it means that if a server can consistently convince the verifier that it is storing the client's data (by passing the protocol's audits), then there must exist an extractor algorithm that can recover the client's entire file from that server's responses, even if the server is malicious. The vPIR construction in this work, and indeed all vPIR constructions, satisfy the PoR requirements as well.

2 Technical Overview of Our CVPIR Construction

In this section, we describe our novel construction π_{CVPIR} for verifiable PIR with logarithmic online communication and constant client "hint" storage. For what follows, we assume the database is structured as an $m \times n$ matrix $\mathbf{D}$ with entries in a ring R. For minimizing the communication volume, we will generally have the row dimension m be logarithmic in the total size, so $m \ll n$. The ring R of matrix entries can be a finite field with a modulus of slightly more than λ bits, where λ is a statistical security parameter, say 40.

For honest clients and servers, our protocol contains three steps: first computing a *digest* non-interactively and setting up the matrix $\mathbf{D}$; then a *precomputation* stage between a client and server based on a single digest, and finally a *query* protocol between client and server to privately fetch a single column of $\mathbf{D}$. Note that the same digest can be shared by many servers, and precomputation only needs to be performed once by each client, after which an arbitrary number of queries can be performed with respect to that digest.

For ease of exposition, we begin this section by describing the online query protocol, assuming that the client can validate responses from the server. Then, we explain the required digest and precomputation steps that are needed for query validation.

2.1 Query Protocol

The client first chooses a private index sel for the column of $\mathbf{D}$ that they wish to fetch using PIR. The client then prepares λ *query vectors* $\mathbf{q}_i$. With probability $1/2$ and chosen randomly by the client, each query vector is either:

- An *actual* query vector $\mathbf{a} \in \mathsf{R}^n$, which is a "one-hot" vector encoding of sel with $a_{\mathsf{sel}} = 1$ all other $a_{i \neq \mathsf{sel}} = 0$, or
- A *verification* query vector $\mathbf{v}_j \in \mathsf{R}^n$ with random entries under some structure constraints.

Note that there will be multiple *actual* query vectors in the list, but they are all identical and based on the private index being fetched. By contrast, the *verification* vectors are all distinct, and they crucially do *not* depend on the private index.

Before sending these vectors to the server, the client *compresses* each one as a list of smaller vectors, and *encrypts* them using FHE. In fact, the client never stores the full query vectors, which are large, but only their compressed outer product form (as described below).

Upon receiving the compressed, encrypted query vectors: the server decompresses each one homomorphically to get an encrypted vector $\mathbf{q}_i \in \mathsf{R}^n$, then performs a plaintext-ciphertext matrix-vector multiplication to obtain each (encrypted) response vector $\mathbf{r}_i = \mathbf{D}\mathbf{q}_i$. These (small) encrypted response vectors are sent back to the client.

Upon receiving the λ response vectors, the client first decrypts each one, and then, based on information saved from the precomputation stage, attempts to *validate* each response vector to see if it is the correct output of the matrix-vector product. Finally, the client prepares its output as follows, using a strategy reminiscent of cut-and-choose:

1. If *any* of the verification vector responses does not successfully validate, the query fails and the client outputs $\perp$.
2. If all verifications vectors are valid and at least one actual query response validates, the client outputs that validated query response $\mathbf{r}_i \in \mathsf{R}^m$, which must equal the requested column of $\mathbf{D}$.
3. Otherwise, if all validation responses are valid but all actual responses are invalid, the client outputs the zero vector $\mathbf{0} \in \mathsf{R}^m$.

Notice that the failure output (condition 1) depends only on the verification vectors and responses, which in turn are independent of the client's private index. Informally, this prevents selective failure based on the client's index.

Condition 3 can occur for two reasons: (a) the negligible probability $2^{-\lambda}$ that a malicious server correctly guesses the type of each query vector, or (b) the sel'th column of $\mathbf{D}$ was mangled by the server during digest computation, and no query on this index can successfully reach condition 2. By outputting the zero vector $\mathbf{0}$ in case (b), the client achieves a *consistent* view of the database, which is all that is required for verified PIR.

Query vector compression and decompression. The compression scheme is parameterized by a list of small sizes $t_1, t_2, \ldots, t_\ell$ such that their product is at least as large as the vector dimension n. Each query vector $\mathbf{q}_i \in \mathsf{R}^n$ (either actual or verification) is compressed in an identical way into a list of ℓ small vectors $\mathbf{s}_{i,1} \in \mathsf{R}^{t_1}, \ldots, \mathbf{s}_{i,\ell} \in \mathsf{R}^{t_\ell}$. Decompression means taking the (flattened) *tensor product* (i.e., a repeated outer product) of these ℓ small vectors, and truncating the result to length n.

For example, with $\ell = 3$ and $t_1 = 2$, $t_2 = 2$, $t_3 = 3$, the total compressed size is 7 entries between 3 small vectors, the decompressed size is $n = 12$, and the decompression can be written as:

$$\underset{\mathbf{s}_{i,1}}{\begin{bmatrix} a \\ b \end{bmatrix}} \otimes \underset{\mathbf{s}_{i,2}}{\begin{bmatrix} c \\ d \end{bmatrix}} \otimes \underset{\mathbf{s}_{i,3}}{\begin{bmatrix} e \\ f \\ g \end{bmatrix}} = \underset{\mathbf{q}_i}{\begin{bmatrix} ace \\ acf \\ acg \\ ade \\ \vdots \\ bdg \end{bmatrix}}$$

To compress a one-hot actual query vector $\mathbf{r}_i = \mathbf{a}$ encoding a zero-based index sel with $0 \le \mathsf{sel} < n$, first write sel in mixed radix representation with basis $(t_1, \ldots, t_\ell)$, and then produce the one-hot encoding of each digit in this mixed-radix representation. For instance, with the parameters above and $\mathsf{sel} = 7$, the compressed representation is:

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The verification vectors are actually determined from the compression; each verification vector $\mathbf{v}_j \in \mathsf{R}^n$ is created by randomly sampling $t_1 + t_2 + \cdots + t_\ell$ elements of R to fill the ℓ compressed vectors.

2.2 Digest and Preprocessing

In order to talk about validating the different query vector responses, we first need to define the structure of the database *digest* which they will ultimately be validated against.

Digest structure with 2-level commitments. The digest is built in two stages. First, each column of $\mathbf{D}$ is treated as the coefficients of a polynomial P_i, and a polynomial commitment ϕ_i is generated using a $\mathcal{F}_{\text{TPC}}$ subroutine instance. Second, a vector commitment μ is generated from the list of n polynomial commitments $(\phi_1, \ldots, \phi_n)$, and this vector commitment of polynomial commitments is treated as the digest of the entire database $\mathbf{D}$.

Looking ahead, the reason that we need the polynomial commitments to be a UC ideal sub-functionality $\mathcal{F}_{\text{TPC}}$—whereas the vector commitment just uses the actual protocol—has to do with extractability. In our proof that our construction π_{CVPIR} UC-realizes functionality $\mathcal{F}_{\text{PIR}}$, our Simulator needs to extract the entire database $\mathbf{D}$ during preprocessing. In the actual protocol, the honest client learns all n polynomial commitments ϕ_i during preprocessing, as well as a single evaluation opening of each one. So the simulator does not need to extract from the vector commitment (it already has the complete opening), but the simulator does need to extract the complete polynomial from each polynomial commitment after a single validated evaluation. This column extraction is modeled through $\mathcal{F}_{\text{TPC}}$.

Validating actual query vector responses. Recall that each actual query vector $\mathbf{q}_i = \mathbf{a}$ is a one-hot vector with a 1 at the location corresponding to the private column index sel the client is trying to fetch. The response vector $\mathbf{r}_i$ associated with each actual query vector should equal the selected column of $\mathbf{D}$ as committed to by the digest (vector commitment) μ.

First, imagine performing this verification check if the column index sel did not need to be private. The client could simply ask the server for the polynomial commitment ϕ_{sel} and the vector commitment opening Λ_{sel}, and then check up both levels of commitments:

- The response vector $\mathbf{r}_i$ is valid with respect to the polynomial commitment ϕ_{sel}.
- ϕ_{sel} (for the column at index sel) is consistent with the opening Λ_{sel} to the digest μ.

These checks work even against a malicious server, but of course the client cannot ask for ϕ_{sel} and Λ_{sel} without revealing her private PIR query index sel.

So instead, during preprocessing, the client and server embed the tuple (ϕ_i, Λ_i) — which we call the *tag* — at the end of each column i of $\mathbf{D}$ by adding some extra rows $\mathbf{T}$ and producing a new, slightly larger matrix $\mathbf{D}'$:

$$\begin{bmatrix} \\ \mathbf{D}' \\ \\ \end{bmatrix} = \left[\begin{array}{c} \mathbf{D} \\ \hdashline \mathbf{T} \end{array}\right]$$

This augmented $\mathbf{D}'$ will then be used by the client and server for the remainder of the protocol. We write m' for the (augmented) row dimension of $\mathbf{D}'$, and we note that (in our eventual choices of commitment protocols) each tag consists of exactly two elliptic curve points, and so $m' = m + O(1)$.

In the online query phase, the actual query vectors will yield a column of $\mathbf{D}$ along with its tag, which can then be checked by the client at query-time for consistency with the digest μ. Crucially, these checks do *not* require the client to store anything after the preprocessing phase except for the digest μ itself. The binding properties of the polynomial commitment and vector commitment schemes ensure that, for any digest μ and any column index, there is at most one vector which could pass the checks.

Validating verification query vector responses. The commitments and tags from above only guarantee client success when at least one of the actual query vectors is validly computed by the server. The validity of the verification query vectors is what guarantees that with high probability. Specifically, in the online query phase, the client needs to verify the matrix-vector product $\mathbf{r}_i = \mathbf{D}'\mathbf{v}_j$, where $\mathbf{r}_i$ is the server's response vector, and $\mathbf{v}_j$ is a (structured) random verification query vector.

This validation is enabled at preprocessing time by selecting a random small vector $\mathbf{u} \in \mathsf{R}^{m'}$ and computing $\mathbf{s}^\intercal = \mathbf{u}^\intercal\mathbf{D}'$. This computation would be a straightforward matrix-vector product, except that the result must be validated against the digest μ to the entire database, and the vectors $\mathbf{u}$ and $\mathbf{s}$ must be kept hidden from the server.

For notational convenience, split $\mathbf{u} \in \mathsf{R}^{m'}$ into an initial segment $\mathbf{u}_0 \in \mathsf{R}^m$ and a tag segment $\mathbf{u}_1 \in \mathsf{R}^{m'-m}$. Recalling that $\mathbf{D}'$ is the original data matrix $\mathbf{D}$ stacked on top of the tags $\mathbf{T}$, we see that $\mathbf{s}$ can be computed as:

$$\mathbf{s}^\intercal = \mathbf{u}_0^\intercal\mathbf{D} + \mathbf{u}_1^\intercal\mathbf{T}.$$

Because the client already knows the tag rows $\mathbf{T}$ during precomputation, the problem reduces to computing $\mathbf{u}_0^\intercal\mathbf{D}$. To do this without revealing $\mathbf{u}$, we construct a new, two-round protocol that achieves this preprocessing by leveraging the vector and polynomial commitments that were used to construct μ, and based on the missing exponent assumption. This two-round protocol forms the heart of our construction for the $\mathcal{F}_{\text{PIR}}$ ideal functionality, but because it is also rather technical, we defer the details to Section 6.1.

At query time, the vectors $\mathbf{u}, \mathbf{s}$ from precomputation can be used to validate the query response $\mathbf{r}_i$ by checking the equality of two dot products: $\mathbf{s}^\intercal\mathbf{v} = \mathbf{u}^\intercal\mathbf{r}_i$. The problem here is that both $\mathbf{s}$ and $\mathbf{v}$ are too large; we do not want the client to have to store or compute with them in the online query phase. Instead, as both these large vectors are independent of the query index or response, the client precomputes a list of $k\lambda$ random validation vectors $\mathbf{v}_j$ and, for each one, stores the dot product $c_j = \langle \mathbf{s}, \mathbf{v_j} \rangle$. At the end of precomputation, the client deletes $\mathbf{s}$ and saves only the $k\lambda$ precomputed dot products c_j, as well as the random seeds needed to regenerate $\mathbf{u}$ and each $\mathbf{v}_j$.

For our security to hold, the client may never re-use the same validation vector $\mathbf{v}_j$ with the same FHE key. So after every k lookup queries, the client generates new FHE keys

and sends the public evaluation keys to the server. The parameter $k \in O(1)$ is chosen as a trade-off between the space to store $k\lambda$ precomputed dot products, and the communication volume to send FHE keys every k queries.

3 Preliminaries

We assume the reader is familiar with standard notions in modern cryptography like collision-resistant hash functions, computational indistinguishability, semantic security of encryption schemes, and the simulation-based framework for defining secure computation [Lin16]. In this section, we define some cryptographic building blocks and assumptions used in this work.

3.1 Cryptographic Tools

Polynomial commitment. A *polynomial commitment scheme* [KZG10] enables a prover to commit succinctly to a polynomial $P(X) \in \mathbb{F}_q[X]$, and later reveal evaluations $P(a)$ at arbitrary points $a \in \mathbb{F}_q$. The verifier can efficiently check that the evaluation is correct using a short proof. This scheme consists of the following algorithms:

- $\mathsf{PC.KeyGen}(1^k, d) \to \mathsf{pp}$: On input the security parameter k and the maximum allowable polynomial degree d, output public parameters pp (which implicitly define the algebraic structure $\mathcal{G}$).
- $\mathsf{PC.Commit}_{\mathsf{pp}}(P(x)) \to \phi$: Given the public parameter pp and a polynomial $P(x)$, it produces a succinct commitment ϕ to that polynomial.
- $\mathsf{PC.VerifyPoly}_{\mathsf{pp}}(\phi, P(x)) \to \top/\bot$: Verifies that ϕ is the commitment to $P(x)$.
- $\mathsf{PC.CreateWitness}_{\mathsf{pp}}(P(x), a) \to w$: Given the polynomial $P(x)$, and an evaluation point $a \in \mathbb{F}_q$, this algorithm computes a witness w to the evaluation $P(a)$.
- $\mathsf{PC.VerifyEval}_{\mathsf{pp}}(\phi, a, b, w) \to \top/\bot$: Verifies that $P(a) = b$ based on polynomial commitment ϕ to $P(x)$ and witness w to the evaluation.

We require the scheme to satisfy:

- *Correctness:* Honest commitments and evaluations must verify successfully.
- *Polynomial binding:* It is computationally infeasible to open the same commitment for two different polynomials.
- *Evaluation binding:* It is computationally infeasible to open the same commitment at a point a for two different evaluation values.
- *Succinctness:* The sizes of the commitment d and of each witness w output by $\mathsf{PC.CreateWitness}$ are independent of k.

Our protocol can use schemes such as KZG commitments [KZG10], which rely on a structured reference string and pairings over elliptic curves, which is described in Appendix B.

Vector commitment. A *vector commitment scheme* [CF13] allows to commit to an ordered list of values in such a way that the committer can later open a specific element in the list. A vector commitment scheme provides the following algorithms[1]:

- $\mathsf{VC.KeyGen}(1^k, t) \to \mathsf{pp}$: On input the security parameter k and vector length t, output public parameters pp (which implicitly define the message space $\mathcal{M}$).
- $\mathsf{VC.Com}_{\mathsf{pp}}(m_1, \ldots, m_t) \to (C, \mathsf{aux})$: On input $(m_1, \ldots, m_t) \in \mathcal{M}^t$ and pp, output a commitment C and auxiliary information aux.

[1]The original vector commitment scheme considers updates. However, we do not consider updates in this work, so we omitted that feature in the description.

- $\mathsf{VC.Open}_{\mathsf{pp}}(m, i, \mathsf{aux}) \to \Lambda_i$: Run by the committer to produce a proof Λ_i that m is the ith committed message.
- $\mathsf{VC.Ver}_{\mathsf{pp}}(C, m, i, \Lambda_i) \to \top/\bot$: Output $\top$ if Λ_i is a valid proof that C commits to a sequence $(m_1, \ldots, m_t)$ with $m_i = m$. Otherwise output $\bot$.

We require the scheme to satisfy:

- *Correctness:* Honest commitments and evaluations must verify successfully.
- *Binding:* It is computationally infeasible to open the same commitment for the same index i to two different values.
- *Succinctness:* The sizes of the commitment C and of each opening (proof) output by $\mathsf{VC.Open}$ are independent of t.

Our protocol can use a VC scheme provided in [CF13], which rely on a structured reference string and pairings over elliptic curves.

Linearly homomorphic encryption. Linearly Homomorphic Encryption (LHE) schemes allow arithmetic on encrypted data, specifically supporting addition and scalar multiplication over a finite field. In particular, an LHE scheme supports the following algorithms:

- $\mathsf{LHKeyGen}(1^\kappa) \to (\mathrm{PK}, \mathrm{SK})$: Generates private and public keys with respect to a security parameter(and implicitly define the plaintext space $\mathbb{F}_q$ and ciphertext space).
- $\mathsf{LHEnc}(\mathrm{PK}, a) \to c$: Encrypts a plaintext $a \in \mathbb{F}_q$ using PK.
- $\mathsf{LHDec}(\mathrm{SK}, c) \to a$: Decrypts a ciphertext c using SK.

If it's obvious from the context, we sometimes omit PK, SK from LHEnc and LHDec. The scheme allows arithmetic operations on ciphertexts, namely addition $+_L$ and scalar multiplication $\ltimes_L$. Implicitly, homomorphic operations require the knowledge of the public key PK used for encryption. The following correctness is expected for all $a, b \in \mathbb{F}_q$.

- $\mathsf{LHDec}(\mathsf{LHEnc}(a)) = a$
- $\mathsf{LHDec}(\mathsf{LHEnc}(a) +_L \mathsf{LHEnc}(b)) = a + b$
- $\mathsf{LHDec}(a \ltimes_L \mathsf{LHEnc}(b)) = ab$

Fully homomorphic encryption. A fully Homomorphic Encryption (FHE) scheme is an LHE with an additional operator, namely the ciphertext-ciphertext product $\times_F$. Similarly to the LHE scheme, it is built with algorithms $\mathsf{FHKeyGen}, \mathsf{FHEnc}, \mathsf{FHDec}$ and operators $+_F, \ltimes_F, \times_F$. The latter satisfies the following correctness, for all plaintexts $a, b \in \mathbb{F}_q$.

- $\mathsf{FHDec}(\mathsf{FHEnc}(a) \times_F \mathsf{FHEnc}(b)) = ab$

Note that FHDec requires the secret key, and the $+_F, \ltimes_F, \times_F$ operators can use the public key.

3.2 Assumptions

Hardness of missing exponent. In this paper, we introduce a new assumption of hardness of missing exponent. It is similar to the ℓ-wBDHI* assumption in [BBG05, Section 2.3], except that the input contains powers both above and below the target power.

Assumption 1 (Hardness of missing exponent)**.** *Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear pairing between groups of order q, and let $\ell \in \mathbb{N}$. For any PPT algorithm $\mathcal{A}$, we have*

$$\Pr\left[s \leftarrow \mathbb{Z}_q : \mathcal{A}(1^\kappa, g_1, g_2, g_1^s, g_1^{s^2}, \ldots, g_1^{s^{\ell-1}}, g_1^{s^{\ell+1}}, g_1^{s^{\ell+2}}, \ldots, g_1^{s^{2\ell}}) = e(g_1, g_2)^{s^\ell}\right] \leq \mathsf{negl}(\kappa)$$

F-limited leveled fully homomorphic encryption. Using an FHE with unbounded multiplicative depth is unpractical due to the expensive cost of *bootstrapping*. Usually, and in that paper, we only use leveled FHE, which allows evaluation of circuits of a pre-determined bounded multiplicative depth L. For a high-depth circuit F, Chen et al. [CHLR18] introduced a notion of F-limited leveled FHE which says that evaluating F inside ciphertexts is infeasible.

Definition 1 ([CHLR18])**.** We say a leveled fully homomorphic encryption scheme Σ is F-limited if for all F^*, the probability of the following game outputting 1 is negligible:

- Uniformly sample z.
- If $\Sigma.\mathsf{Dec}_{sk}(F^*(\Sigma.\mathsf{Enc}_{pk}(z))) = F(z)$, output 1. Otherwise output 0.

Assumption 2 (F-limited scheme existence)**.** *There is a leveled FHE scheme that is F-limited when F is a high-degree polynomial.*

As with [CHLR18], we use the BFV scheme as an F-limited leveled FHE.

Hardness of RLWE. For our paper, we use the BFV scheme [FV12b] instantiation as FHE, which is secure based on the RLWE hardness assumption.

Assumption 3 (RLWE-hardness assumption [FV12b])**.** *For security parameter λ, let $f(x) = xd + 1$ where $d = d(\lambda)$ is a power of 2. Let $q = q(\lambda) \geq 2$ be an integer. Let $R = \frac{\mathbb{Z}[x]}{(f(x))}$ and let $R_q = \frac{R}{(qR)}$. Let $\chi = \chi(\lambda)$ be a distribution over R. The $RLWE_{d,q,\chi}$ problem is to distinguish the following two distributions: In the first distribution, one samples (a_i, b_i) uniformly from R_q^2. In the second distribution, one first draws $s \leftarrow R_q$ uniformly and then samples $(a_i, b_i) \in R_q^2$ by sampling $a_i \leftarrow R_q$ uniformly, $e_i \leftarrow \chi$ and setting $b_i = a_i s + e_i$. The $RLWE_{d,q,\chi}$ assumption is that the $RLWE_{d,q,\chi}$ problem is infeasible.*

3.3 Global (Restricted) Observable Generic Group

Our protocol operates in the generic group model. Specifically, we adopt the ideal functionality $\mathcal{G}$-oGG from [BFKT24], which formalizes access to a restricted, observable, global generic (bilinear) group resource. This functionality behaves like a globally accessible generic group, with the following additional features:

- An *observability interface* that, via domain separation, allows simulators to observe certain group operations.
- Support for *group reuse* across multiple protocols, without restricting the environment from reusing group elements produced by one protocol as inputs to another.
- *Oblivious sampling* of group elements with unknown discrete logarithms.

For completeness, we provide the full description of this functionality in Appendix C.

4 Ideal functionalities

In this section, we formally specify three ideal functionalities $\mathcal{F}_{\text{PIR}}$, $\mathcal{F}_{\text{VLA}}$, and $\mathcal{F}_{\text{TPC}}$. Looking ahead, we will instantiate these functionalities in Sections 5 to 7.

4.1 Ideal Functionality $\mathcal{F}_{\text{PIR}}$

To achieve composable security, we model the syntax and security properties of our verifiable PIR scheme as an ideal functionality, denoted by $\mathcal{F}_{\text{PIR}}$, within the Universal Composability (UC) framework. The formal UC definition of this functionality is provided in Fig. 1. Below, we present an informal and intuitive explanation of each command supported by the functionality.

The ideal functionality $\mathcal{F}_{\text{PIR}}$ has no parameters. It uses a structured session identifier $\mathsf{sid} = (L, n, \mathit{nonce})$, where L is the bitlength of each database entry and n is the number of entries, and query identifiers $\mathsf{qid} = (\mathsf{sid}, \mathsf{server}, \mathsf{client}, \mathit{nonce})$. We say that db is *valid* if $\mathsf{db} \in (\{0,1\}^L)^n$.

$\mathcal{F}_{\text{PIR}}$ responds to the following messages:

- **Query** (GetDigest, sid, db) from server:
  - Assert db is valid.
  - Send backdoor query (GenDigest, sid, db) to the simulator $\mathcal{S}$ and wait for response (digest, aux).
  - Assert aux $\neq$ `"corrupt"` and no record (digested, sid, digest, db′, aux′) exists with db $\neq$ db′ or aux $\neq$ aux′.
  - Record (digested, sid, digest, db, aux).
  - Return (digest, aux) to server.
- **Message** (Register, qid, digest) from client:
  - Parse qid as (sid, server, client, *nonce*) and assert it has never been used before.
  - Send (GetDB, qid, digest) to server and wait for corresponding response (DBResponse, qid, db, aux).
  - If aux = `"corrupt"` but server is not corrupt, Abort.
  - If (digested, sid, digest, db, aux) is not already recorded, then do:
    - Send backdoor query (CheckDigest, qid, digest, db, aux) to $\mathcal{S}$ and wait for response.
    - If the response was ($\perp$), or another tuple (digested, sid, digest, db′, aux′) is already recorded, then send (RegisterResult, qid, $\perp$) to client and Abort.
    - Else record (digested, sid, digest, db, aux).
  - Record (registered, sid, digest, client).
  - Send (RegisterResult, qid, $\top$) to client.
- **Message** (Lookup, qid, digest, sel) from client:
  - Parse qid as (sid, server, client, *nonce*) and assert it has never been used before, (registered, sid, digest, client) is recorded, and sel $\in [n]$.
  - Send (GetDB, qid, digest) to server and wait for corresponding response (DBResponse, qid, db, aux).
  - If aux = `"corrupt"` but server is not corrupt, Abort.
  - If (digested, sid, digest, db, aux) is not recorded, send (LookupResult, qid, $\perp$) to client.
  - Else send (LookupResult, qid, db[sel]) to client.

Figure 1: Ideal functionality $\mathcal{F}_{\text{PIR}}$ for (verified) private information retrieval

GetDigest is run by an honest prover, who provides a database db and receives decommitment info aux and a digest string μ which is globally bound to the pair $(\mathsf{db}, \mathsf{aux})$.

This digest not necessarily hiding, as the ideal functionality explicitly leaks db to the simulator and asks it to choose the digest string. This situation is common in UC modeling, as the ideal functionality does not "care" what the digest strings look like, only that they are bound to some valid database.

A special value `"corrupt"` for aux represents that the corresponding commitment μ can never be used by honest servers. Note that such corrupted (i.e., maliciously computed) digests μ are still binding to a unique database db, but that database/digest pairing is essentially unable to be transferred to any honest server. This restriction is enforced in the ideal functionality by refusing to return or accept any `"corrupt"` aux value to/from an honest server.

Observe that the ideal functionality does not handle communication of the digest string between a client and server; this task is left to the Environment in the UC world.

Register is a pre-processing step which honest clients must perform with some server before they can perform lookup queries.

The role of Server in our protocol is *not* bound to a single party. Instead, any server who can produce the correct $(\mathsf{db}, \mathsf{aux})$ pair corresponding to the digest μ requested by the Client, can act as the server for that transaction.

To facilitate this, our ideal functionality asks the environment $\mathcal{Z}$ (or the simulator $\mathcal{S}$ in the case of a corrupted server) to produce db and aux for a client's requested digest μ, every time. If the $\mathsf{db}, \mathsf{aux}$ pair has already been bound to μ, then this is guaranteed to work with no extra leakage, and conversely if μ is bound to any other $(\mathsf{db}, \mathsf{aux})$ pair, the operation is guaranteed to fail.

Otherwise, if μ is as-yet unbound, the ideal functionality asks the Adversary whether to bind μ to $(\mathsf{db}, \mathsf{aux})$ now, or to make the operation fail.

If the Register command succeeds, that client may now perform an unbounded number of Lookup queries with the same digest μ.

Lookup represents an actual client PIR query for a single database entry. As with Register, the client's requested server is challenged by the ideal functionality to produce the binding $(\mathsf{db}, \mathsf{aux})$ pair for the client's requested digest μ.

Notice crucially that in this challenge, the server does *not* learn anything about the client's selected index sel. This is the crux of what it means to be a PIR.

The operation may fail, namely if the environment does not produce the correct $(\mathsf{db}, \mathsf{aux})$ pair bound to the digest μ, but this failure cannot depend in any way on the client's index sel.

Because the functionality requires a client to successfully Register with a digest μ before performing any Lookup operations, there is never any leakage to the Adversary during lookups, other than the implicit leakage of the digest μ itself to the environment.

For both Register and Lookup, the server is asked to provide a copy of the original db, and the operation will fail if this does not match with the sepecified digest μ. While we could have allowed register and lookup to fail only with corrupted servers, we chose this modeling approach for two reasons. First, it allows for protocols which are completely stateless on the server-side; a client which registers to one server, for instance, may perform lookups with a different server, with the same integrity guarantees.

Secondly, the need for the server to send the entire database for every successful register or lookup illustrates clearly that any protocol realizing $\mathcal{F}_{\text{PIR}}$ can also trivially act as a proof of retrievability implementation, by replacing PoR audits with arbitrary-index lookup requests. Crucially, note that a real-world protocol realizing $\mathcal{F}_{\text{PIR}}$ need not actually *communicate* the entire data value on every registration and lookup request, but our ideal functionality guarantees that it must have *knowledge* of the entire db in order to successfully complete.

4.2 Ideal Functionality $\mathcal{F}_{\text{VLA}}$

The formal listing of $\mathcal{F}_{\text{VLA}}$ for verified linear algebra is found in Figs. 2 to 3. It supports these five methods:

MatrixCommit is a non-interactive procedure run by the server. It takes a matrix $\mathbf{D}$, and returns a commitment μ to the overall matrix and individual column tags $t_0, \ldots, t_{n-1}$.

The commitment and tag structure are opaque to the ideal functionality, which is modeled in $\mathcal{F}_{\text{VLA}}$ by asking the simulator $\mathcal{S}$ to actually generate them.

The functionality enforces that the commitment μ is binding to some matrix $\mathbf{D} \in \mathsf{R}^{m \times n}$, and the tuple (μ, i, t_i) is therefore also binding to the i'th column of $\mathbf{D}$.

VerifyAll is a non-interactive procedure run by the server, to check that a given matrix $\mathbf{D}$, commitment μ, and tags $t_0, \ldots, t_{n-1}$ are properly formed.

This is needed in our actual protocol because the server is *stateless* and must be told what $\mathbf{D}$ and tags to use by the environment $\mathcal{Z}$ on every query. This VerifyAll method is a mechanism for an honest server to ensure the $\mathbf{D}$ and tags given by $\mathcal{Z}$ are actually consistent with the commitment μ that the client is requesting.

GetTags is an interactive protocol between a client and server, based on a public matrix commitment μ to a matrix $\mathbf{D}$ that is known to the server. If the server approves the request, the client learns the list of column tags $t_0, \ldots, t_{n-1}$. This protocol must be successfully completed by each client prior to calling VecMatProd or VerifyColumn.

Intuitively, this protocol not only fetches the tags for the client, but also proves that the server (even a corrupted one) has knowledge of a valid matrix $\mathbf{D}$ consistent with the given commitment and the return tags.

VecMatProd is an interactive protocol between a client and server, made possible after the client completes GetTags (with the same commitment μ but not necessarily with the same server). The client submits a vector $\mathbf{u} \in \mathsf{R}^m$ and, if the server approves the request, learns $\mathbf{s} \in \mathsf{R}^n$ s.t. $\mathbf{s}^\intercal = \mathbf{u}^\intercal \mathbf{D}$, while the server learns nothing.

VerifyColumn is a non-interactive procedure run by the client, made possible after the client completes GetTags with any server and the given matrix commitment μ. Using only this μ, the index i, and the i'th column tag, the client checks whether a candidate vector $\mathbf{r} \in \mathsf{R}^m$ is equal to the i'th row of the committed matrix $\mathbf{D}$.

In the context of our main protocol π_{CVPIR} for $\mathcal{F}_{\text{PIR}}$, MatrixCommit is used during PIR digest creation, GetTags and VectMatProd are used during the registration (i.e., preprocessing) phase, and VerifyColumn is used in the online query phase.

The ideal functionality $\mathcal{F}_{\text{VLA}}$ is parameterized by a coefficient ring R and a tag bitlength L. It uses a structured session identifier $\mathsf{sid} = (m, n, \mathsf{nonce})$, where m, n are matrix dimensions, and query identifiers $\mathsf{qid} = (\mathsf{sid}, \mathsf{prover}, \mathsf{verifier}, \mathsf{nonce})$.

We say that $\mathbf{D}$ is *valid* if $\mathbf{D} \in \mathsf{R}^{m \times n}$, and tags $(t_0, \ldots, t_{n-1})$ are *valid* if $(t_0, \ldots, t_{n-1}) \in (\{0,1\}^L)^n$, and $\mathbf{u}$ is *valid* if $\mathbf{u} \in \mathsf{R}^m$.

$\mathcal{F}_{\text{VLA}}$ responds to the following messages:

- **Query** (MatrixCommit, $\mathsf{sid}, \mathbf{D}$) from prover:
  - Assert $\mathbf{D}$ is valid
  - Send backdoor query (GenMC, $\mathsf{sid}, \mathbf{D}, \mathsf{prover}$) to the simulator $\mathcal{S}$ and wait for response $(\mu, t_0, \ldots, t_{n-1})$
  - Assert that tags are valid and that no record (mcomm, $\mathsf{sid}, \mu, \mathbf{D}', t'_0, \ldots, t'_{n-1}$) exists with $\mathbf{D}' \neq \mathbf{D}$ or $(t'_0, \ldots, t'_{n-1}) \neq (t_0, \ldots, t_{n-1})$
  - Record (mcomm, $\mathsf{sid}, \mu, \mathbf{D}, t_0, \ldots, t_{n-1}$) and return $(\mu, t_0, \ldots, t_{n-1})$ to prover

- **Message** (VerifyAll, $\mathsf{qid}, \mu, \mathbf{D}, t_0, \ldots, t_{n-1}$) from prover:
  - Assert qid has never been used before, and that $\mathbf{D}$ and $t_0, \ldots, t_{n-1}$ are valid
  - If no tuple (mcomm, $\mathsf{sid}, \mu, \mathbf{D}, t_0, \ldots, t_{n-1}$) is recorded:
    - Send backdoor query (CheckMC, $\mathsf{qid}, \mu, \mathbf{D}, t_0, \ldots, t_{n-1}$) to $\mathcal{S}$ and wait for response
    - If the response is $(\perp)$, or another tuple (mcomm, $\mathsf{sid}, \mu, \mathbf{D}', t'_0, \ldots, t'_{n-1}$) is already recorded, then send (VerifyAllResult, $\mathsf{qid}, \perp$) to prover and Abort
  - Send (VerifyAllResult, $\mathsf{qid}, \top$) to prover

- **Message** (GetTags, qid, μ) from $\mathsf{verifier}$:
  - Assert qid has never been used before
  - Send (GetMatrixTags, qid, μ) to prover via the router and wait for a corresponding response (MatrixTagsResult, $\mathsf{qid}, \mathbf{D}, t_0, \ldots, t_{n-1}$)
  - If no tuple (mcomm, $\mathsf{sid}, \mu, \mathbf{D}, t_0, \ldots, t_{n-1}$) is recorded:
    - Send backdoor query (CheckMC, $\mathsf{qid}, \mu, \mathbf{D}, t_0, \ldots, t_{n-1}$) to $\mathcal{S}$ and wait for response
    - If the response is $(\perp)$, or $\mathbf{D}$ or the tags are not valid, or another tuple (mcomm, $\mathsf{sid}, \mu, \mathbf{D}', t'_0, \ldots, t'_{n-1}$) is already recorded, then send (TagsResult, $\mathsf{qid}, \perp$) to $\mathsf{verifier}$ and Abort
    - Else record (mcomm, $\mathsf{sid}, \mu, \mathbf{D}, t_0, \ldots, t_{n-1}$)
  - Record (gotTags, $\mu, \mathsf{verifier}$)
  - Send (TagsResult, $\mathsf{qid}, t_0, \ldots, t_{n-1}$) to $\mathsf{verifier}$ via the router

Figure 2: Ideal functionality $\mathcal{F}_{\text{VLA}}$ for verified linear algebra, part 1

The ideal functionality $\mathcal{F}_{\text{VLA}}$, continued from Fig. 2

- **Query** $(\textsc{VecMatProd}, \mathsf{qid}, \mu, \mathbf{u})$ from $\mathsf{verifier}$:
  - Assert qid has never been used before, $\mathbf{u}$ is valid, and $(\textsc{gotTags}, \mu, \mathsf{verifier})$ is recorded
  - Send $(\textsc{GetMatrixTags}, \mathsf{qid}, \mu)$ to prover via the router and wait for a corresponding response $(\textsc{MatrixTagsResult}, \mathsf{qid}, \mathbf{D}, t_0, \ldots, t_{n-1})$
  - If $(\textsc{mcomm}, \mathsf{sid}, \mu, \mathbf{D}, t_0, \ldots, t_{n-1})$ is not recorded, send $(\text{VMP}\textsc{Result}, \mathsf{qid}, \perp)$ to $\mathsf{verifier}$ and Abort
  - Compute $\mathbf{s}^\intercal \leftarrow \mathbf{u}^\intercal \mathbf{D}$
  - Send $(\text{VMP}\textsc{Result}, \mathsf{qid}, \mathbf{s})$ to $\mathsf{verifier}$

- **Query** $(\textsc{VerifyColumn}, \mathsf{sid}, \mu, \mathbf{r}, i, \mathsf{tag})$ from $\mathsf{verifier}$:
  - Assert $(\textsc{gotTags}, \mu, \mathsf{verifier})$ is recorded
  - Lookup recorded tuple $(\textsc{mcomm}, \mathsf{sid}, \mu, \mathbf{D}, t_0, \ldots, t_{n-1})$
  - If $\mathsf{tag} \neq t_i$ or $\mathbf{r}$ is not the ith column of $\mathbf{D}$, return $(\perp)$ to $\mathsf{verifier}$
  - Else return $(\top)$ to $\mathsf{verifier}$

Figure 3: Ideal functionality $\mathcal{F}_{\text{VLA}}$ for verified linear algebra, part 2

4.3 Functionality $\mathcal{F}_{\text{TPC}}$ for Polynomial Commitment

Our PIR protocol makes use of polynomial commitments, and the UC security proof for our PIR scheme further requires that the simulator be able to extract the committed polynomial after a successful opening. To support this requirement, we define in Fig. 4 a simple ideal functionality $\mathcal{F}_{\text{TPC}}$ for transferable polynomial commitments. We emphasize that $\mathcal{F}_{\text{TPC}}$ provides the binding property but intentionally does *not* provide hiding.

5 Protocol π_{TPC} Realizing $\mathcal{F}_{\text{TPC}}$

Having listed the three main ideal functionalities we will implement, we now proceed to build up our three protocols to implement each one, over the next three sections of the paper.

In this section, we provide a protocol π_{TPC} and prove that it securely realizes $\mathcal{F}_{\text{TPC}}$, by adapting classical KZG polynomial commitments to the syntax of our ideal functionality over the network.

Let KZG denote the KZG polynomial commitment scheme [KZG10]; for completeness, we describe this protocol in Appendix B. Recall that the functionality $\mathcal{F}_{\text{TPC}}$ does not provide a hiding property. In particular, when a polynomial P is committed, functionality immediately leaks P to the adversary $\mathcal{S}$. Given this weaker security guarantee, we construct a protocol π_{TPC} based on the KZG commitment scheme that securely realizes $\mathcal{F}_{\text{TPC}}$ in $\mathcal{F}_{\text{CRS}}$-hybrid in the presence of $\mathcal{G}$-oGG. The ideal functionality $\mathcal{F}_{\text{CRS}}$ generates the common reference string pp by running the KZG key generation algorithm $\mathsf{pp} \leftarrow \mathsf{KZG.KeyGen}(1^k, d)$ where the arithmetic operations are performed through $\mathcal{G}$-oGG.

Lemma 1. *Protocol π_{TPC} (Fig. 5) securely realizes $\mathcal{F}_{\text{TPC}}$ (Fig. 4) in the $\mathcal{F}_{\text{CRS}}$-hybrid in the presence of $\mathcal{G}$-oGG.*

The ideal functionality $\mathcal{F}_{\text{TPC}}$

Parameters: coefficient ring R, degree upper bound d, and the commitment length L.

A polynomial P is said to be *valid* if $P \in R[x]$ and $deg(P) < d$. Also, a commitment ϕ is said to be *valid* if $\phi \in \{0,1\}^L$.

$\mathcal{F}_{\text{TPC}}$ responds to the following messages:

- **Query** (PolyCommit, sid, P) from prover:
  - Assert P is valid.
  - Send (GenPC, sid, P) to the simulator $\mathcal{S}$ and wait for response (ϕ)
  - Assert ϕ is valid.
  - Assert no tuple (pcomm, sid, ϕ, P') has been recorded with $P' \neq P$.
  - Record (pcomm, sid, ϕ, P) and return (ϕ) to prover
- **Query** (OpenEval, sid, verifier, ϕ, P, a) from prover:
  - Assert P and ϕ are valid.
  - If (pcomm, sid, ϕ, P) has not been recorded, then
    - Send (CheckPC, sid, ϕ, P) to $\mathcal{S}$ and wait for response
    - If response is ($\top$) and no tuple (pcomm, sid, ϕ, P') with $P \neq P'$ has been recorded, then record (pcomm, sid, ϕ, P).
  - If (pcomm, sid, ϕ, P) has still not been recorded, return $\perp$ to verifier and abort.
  - Send (EvalOpening, sid, ϕ, a, $P(a)$, prover) to verifier
  - Return OK to prover.
- **Query** (VerifyPoly, sid, ϕ, P) from verifier:
  - Check if P and ϕ are valid. If the check fails, abort.
  - If (pcomm, sid, ϕ, P) has not been recorded, then
    - Send (CheckPC, sid, ϕ, P) to $\mathcal{S}$ and wait for response
    - If response is ($\top$) and no tuple (pcomm, sid, ϕ, P') with $P \neq P'$ has been recorded, then record (pcomm, sid, ϕ, P).
  - If (pcomm, sid, ϕ, P) has been recorded, return ($\top$) to verifier; otherwise return ($\perp$) to verifier

Figure 4: Ideal functionality $\mathcal{F}_{\text{TPC}}$ for transferable polynomial commitments

Functionality $\mathcal{F}_{\text{CRS}}$

Below, $\mathcal{G}$-oGG implicitly defines the algebraic group under which $\mathcal{F}_{\text{CRS}}$ works.

$\mathcal{F}_{\text{CRS}}$, parameterized with the degree upper-bound d, has access to $\mathcal{G}$-oGG and responds to the following message:

- **Query** (GetCRS, sid):
  - If pp has not been recorded, compute $\mathsf{pp} \leftarrow \mathsf{KZG.KeyGen}(d)$ and record pp.
  - return pp

Protocol π_{TPC}

We assume that session identifier sid includes a degree upper-bound d and a prime p for the group coefficient ring size. The protocol has access to $\mathcal{F}_{\text{CRS}}$ and $\mathcal{G}$-oGG appropriately parameterized with d and p. The common reference string pp is from $\mathcal{F}_{\text{CRS}}$. $\mathcal{G}$-oGG implicitly defines the algebraic group under which the protocol works. A polynomial P is said to be *valid* if $P \in R[x]$ and $deg(P) < d$.

When acting as prover, π_{TPC} responds to the following inputs:

- Input query (PolyCommit, sid, P) from environment $\mathcal{Z}$
  - Check if polynomial P is valid. If the check fails, abort.
  - Compute $\phi \leftarrow \mathsf{KZG.Commit}_{\mathsf{pp}}(P)$ and output (ϕ) to $\mathcal{Z}$
- Input query (OpenEval, sid, verifier, ϕ, P, a) from $\mathcal{Z}$:
  - Check if $\mathsf{KZG.VerifyPoly}_{\mathsf{pp}}(\phi, P) = \top$. If the check fails, abort.
  - Compute $b \leftarrow P(a)$ and $w \leftarrow \mathsf{KZG.CreateWitness}_{\mathsf{pp}}(\phi, P, a)$
  - Send (EvalProof, sid, ϕ, a, b, w) to verifier and return (OK) to $\mathcal{Z}$.

When acting as verifier, π_{TPC} responds to the following inputs and messages:

- Input query (VerifyPoly, sid, ϕ, P) from $\mathcal{Z}$:
  - Compute $v \leftarrow \mathsf{KZG.VerifyPoly}_{\mathsf{pp}}(\phi, P)$ and return v to $\mathcal{Z}$.
- Message (EvalProof, sid, ϕ, a, b, w) from prover:
  - Check if $\mathsf{KZG.VerifyEval}_{\mathsf{pp}}(\phi, a, b, w) = \top$. If the check fails, abort.
  - Output (EvalOpening, sid, ϕ, a, b, prover) to $\mathcal{Z}$

Figure 5: Protocol π_{TPC} realizing the ideal functionality $\mathcal{F}_{\text{TPC}}$ in the $\mathcal{F}_{\text{CRS}}$-hybrid in the presence of $\mathcal{G}$-oGG

The remainder of this section contains a proof of Lemma 1.

Proof. We describe a simulator such that the real-world execution is indistinguishable from the ideal-world execution. Fig. 6 illustrates the interaction between the real and ideal worlds, as well as the roles of the involved entities and functionalities. Note that, in the ideal world, the simulator additionally simulates $\mathcal{F}_{\text{CRS}}$, generating both the public parameters for the KZG commitment scheme and the trapdoor s.

We note that $\mathcal{G}$-oGG.Observe$_{\mathsf{sid}}$ returns only illegal operations (all standard group operations derived from the generator are considered legal). Consequently, all group operations performed by honest provers are confined to the domain of the current session and are therefore unobservable by the environment.

To enable extraction of the committed polynomial, we employ the extraction technique of [BFKT24]. Whenever $\mathcal{Z}$ queries $\mathcal{G}$-oGG in a session with identifier sid, the simulator transparently relays these queries to $\mathcal{G}$-oGG while recording them. If $\mathcal{Z}$ issues a query that involves a group element derived from a different session sid′, $\mathcal{G}$-oGG detects the illegality, which becomes observable to the simulator. Based on the recorded queries and the observables provided by $\mathcal{G}$-oGG, the simulator can extract the polynomial.

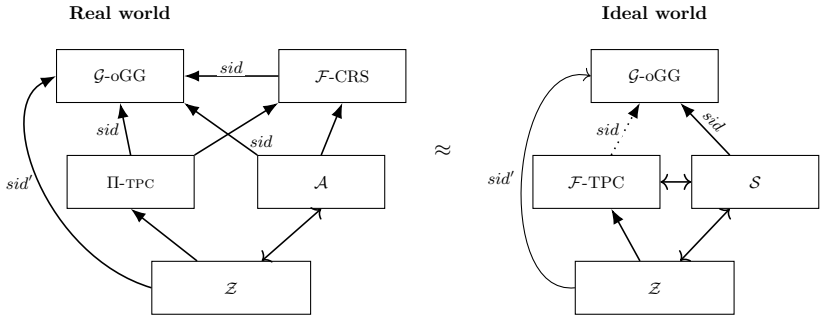


Figure 6: An illustration of the real and ideal world settings

Throughout the simulation, $\mathcal{S}$ maintains a local table storing tuples of the form $(\textit{committed}, \phi, P)$ that bind commitments to their underlying polynomials.

- **Backdoor query** $(\textbf{GenPC}, \mathsf{sid}, P)$ **from** $\mathcal{F}_{\text{TPC}}$**.**

  The simulator computes

  $$\phi \leftarrow \mathsf{KZGCommit}_{\mathsf{pp}}(P).$$

  It records $(\textit{committed}, \phi, P)$ and returns ϕ to $\mathcal{F}_{\text{TPC}}$.

  Intuition. This mirrors an honest commitment in the real protocol, except that the simulator retains knowledge of the committed polynomial for later extraction and consistency checks.

- **Backdoor query** $(\textbf{CheckPC}, \mathsf{sid}, \phi, P)$ **from** $\mathcal{F}_{\text{TPC}}$**.**

  The simulator runs $\mathsf{KZGVerify}_{\mathsf{pp}}(\phi, P)$. If verification fails, it returns $\perp$ to $\mathcal{F}_{\text{TPC}}$. Otherwise, it records $(\textit{committed}, \phi, P)$ and returns $\top$.

  Intuition. This enforces the same correctness checks as in the real execution: invalid commitments are rejected, while valid ones are consistently recorded.

- **Router message** $(\textbf{HonestMessage}, \mathsf{prover}, \mathcal{F}_{\text{TPC}}, \textbf{OpenEval}, \mathsf{sid})$**.**

  The simulator records $(\textit{opener}, \mathsf{sid}, \mathsf{prover})$ and returns Deliver to the router.

  Intuition. This bookkeeping preserves the causal structure of the protocol and allows later evaluation requests to be matched with the correct prover and session.

- **Router message** (**HonestMessage**, $\mathcal{F}_{\text{TPC}}$, verifier, **EvalOpening**, sid)**.**

  The simulator retrieves the most recent tuple (*opener*, sid, prover) and sends the simulated router backdoor

  $$(\textsc{HonestMessage}, \mathsf{prover}, \mathsf{verifier}, \textsc{EvalProof}, \mathsf{sid})$$

  to the adversary. After receiving Deliver, the simulator returns Deliver to the ideal-world router.

  Intuition. This step preserves the adversary's scheduling power and view of honest message delivery.

- **Router message** (**LeakedMessage**, $\mathcal{F}_{\text{TPC}}$, verifier, **EvalOpening**, sid, ϕ, a, b, prover)**.**

  The simulator looks up (*committed*, ϕ, P) and computes an opening witness for evaluation at point a as

  $$w \leftarrow \mathsf{KZGCreateWitness}_{\mathsf{pp}}(P, a).$$

  It sets $\mathsf{advLeakage} := (\phi, a, b, w)$ and sends the backdoor

  $$(\textsc{LeakedMessage}, \mathsf{prover}, \mathsf{verifier}, \textsc{EvalProof}, \mathsf{sid}, \phi, a, b, w)$$

  to the adversary.

  Intuition. Because the simulator knows the committed polynomial, it can always generate a valid opening witness consistent with the functionality's evaluation value $b = P(a)$, exactly matching the leakage available in the real execution.

- **Backdoor** (**CorruptMessage**, prover, verifier, **EvalProof**, sid, ϕ, a, b, w) **from the adversary.**

  The simulator runs $\mathsf{KZGVerifyEval}_{\mathsf{pp}}(\phi, a, b, w)$. If verification fails, the simulator aborts. Otherwise, the simulator extracts the unique polynomial P consistent with ϕ using the recorded queries and observables provided by $\mathcal{G}$-oGG. Since the verification accepts, by the binding property of the KZG commitment scheme it holds that $P(a) = b$. The simulator then forwards the corruption event to the ideal-world router by sending

  $$(\textsc{CorruptMessage}, \mathsf{prover}, \mathcal{F}_{\text{TPC}}, \textsc{OpenEval}, \mathsf{sid}, \mathsf{verifier}, \phi, a, b, w).$$

  Intuition. The adversary's evaluation proof is forwarded verbatim after being validated, while extraction of P is used only to maintain internal consistency of the simulation.

- If a simulator receives an (HonestMessage, ...) backdoor query that is not explicitly handled, it is assumed to immediately reply with (Deliver).

All messages, leakages, and verification outcomes produced by the simulator are distributed identically to those in a real execution of π_{TPC}. Deviations by corrupted parties are handled consistently in both worlds. Therefore, the real and ideal executions are indistinguishable, completing the security proof. □

6 Verified Linear Algebra Protocol

In this section, we provide a protocol π_{VLA} that instantiates the functionality $\mathcal{F}_{\text{VLA}}$ to support verified and private vector-matrix products and column verifications, as defined previously in Fig. 2 of Section 4. This protocol fits in the middle of our modular construction: it uses $\mathcal{F}_{\text{TPC}}$ from Sections 4 to 5, and looking ahead it will be an essential subroutine of our eventual CVPIR protocol in Section 7.

6.1 Protocol π_{VLA} Realizing $\mathcal{F}_{\text{VLA}}$

We have designed an efficient protocol π_{VLA} that UC-realizes $\mathcal{F}_{\text{VLA}}$, which is listed fully in Figs. 7 to 8. Here we describe briefly the key ideas of the construction.

Commitments and tags. The commitment μ and tags $t_0, \ldots, t_{n-1}$ are computed from the matrix $\mathbf{D} \in \mathsf{R}^{m \times n}$. We treat the elements in column i as the coefficients of a *column polynomial*, denoted $P_i(X) \in \mathsf{R}[X]$, each with degree less than m. Then, using an instance of $\mathcal{F}_{\text{TPC}}$, a polynomial commitment ϕ_i is generated for each $P_i(X)$.

Next, the list of n polynomial commitment strings $\phi_0, \ldots, \phi_{n-1}$ is treated as a vector for a vector commitment scheme VC. The matrix commitment μ is generated as a vector commitment to the list of ϕ_i's, and an opening proof Λ_i is generated for each index $i \in [n]$.

Finally, vector commitment openings Λ_i are computed to prove each ϕ_i is at index i in the vector committed to by μ. The i'th column tag is the pair $t_i = (\phi_i, \Lambda_i)$. All of this is done by the (honest) server when running GenerateTags. We provide a visual description of this procedure in Fig. 21.

Fetching and verifying the tags. To implement the functionality of GetTags, we could simply have the server respond to the client's request by sending the tags corresponding to the requested matrix commitment μ.

However, this is not enough to ensure that the tags actually correspond to a valid $\mathbf{D} \in \mathsf{R}^{m \times n}$. For that, the client should verify that the vector commitment openings Λ_i are consistent with the overall vector commitment μ itself, and then perform some interaction with $\mathcal{F}_{\text{TPC}}$ to verify the polynomial commitments ϕ_i are bound to some actual polynomial $P_i(X)$ with degree less than m.

Because both of these checks must already occur in the implementation of VectMatProd, for ease of presentation and without loss of generality or efficiency we combine both of these into a single subroutine $\mathsf{TagsAndVMP}$.

Verifying individual columns. Implementing VerifyColumn is straightforward given our tag structure: the client checks consistency of the polynomial commitment ϕ_i and VC proof Λ_i with the matrix commitment μ using $\mathsf{VC.Ver}$, then uses the VerifyPoly method of $\mathcal{F}_{\text{TPC}}$ to connect the polynomial commitment ϕ_i to the actual column vector $\mathbf{r}$.

Computing a verified vector-matrix product. The crux of our protocol π_{VLA} is the $\mathsf{TagsAndVMP}$ subroutine, our novel two-round interactive protocol to simultaneously fetch tags, compute a vector-matrix product, and verify both of these are consistent with a commitment μ.

The computation itself is straightforward using linearly homomorphic encryption: the client sends $\mathbf{a} = \mathsf{LHE.Enc}(\mathbf{u})$ to the server, which can then compute homomorphically $\mathbf{w}$, that encrypts $\mathbf{s}^\intercal = \mathbf{u}^\intercal \mathbf{D}$, and return it for the client to decrypt. The challenge is *verifying* this result is consistent with each column polynomial commitment ϕ_i.

Consider a single column vector $\mathbf{D}_{*,i} \in \mathsf{R}^m$ of matrix $\mathbf{D}$ with corresponding column polynomial $P_i(X)$. Inspired by the work of MyOPE on oblivious polynomial evaluation [INdPP22], we want to connect the desired dot product $\langle \mathbf{u}, \mathbf{D}_{*,i} \rangle$ to some polynomial operations with $P_i(X)$.

To that end, first define a polynomial $U(X)$ with $\deg U < m$ whose coefficients are the same as the vector $\mathbf{u}$ in reverse:

$$U(X) = u_{m-1} + u_{m-2}X + \cdots + u_0 X^{m-1}$$

The polynomial product $P_i(X)U(X)$ has, as the coefficient of X^{m-1}, the desired dot product $\langle \mathbf{u}, \mathbf{D}_{*,i} \rangle$. So we define another polynomial which is explicitly missing this middle term and therefore will give a reduction to the missing exponent game of Assumption 1:

$$Q_i^{(U)}(X) = P_i(X)U(X) - \langle \mathbf{u}, \mathbf{D}_{*,i} \rangle X^{m-1} \tag{1}$$

Protocol π_{VLA} is parameterized by bitlengths L_{TPC} for $\mathcal{F}_{\text{TPC}}$ and L_{VC} for VC, has $\mathsf{sid} = (m, n, (q, \mathsf{sid}_{\text{TPC}}, \mathsf{nonce}))$, and uses local primitives: VC and LHE.
The implicit parameters of $\mathcal{F}_{\text{VLA}}$ are $\mathsf{R} = \mathbb{F}_q$ and $L = L_{\text{TPC}} + L_{\mathsf{VC}}$.

π_{VLA} responds to the following inputs and messages when acting as prover:

- Input $(\textsc{MatrixCommit}, \mathsf{sid}, \mathbf{D})$ from environment $\mathcal{Z}$:
  - Assert $\mathbf{D}$ is valid.
  - For each $i \in [n]$, send query $(\textsc{PolyCommit}, \mathsf{sid}_{\text{TPC}}, \mathbf{D}_{*,i})$ to $\mathcal{F}_{\text{TPC}}$ and wait for response (ϕ_i).
  - Compute $(\mu, \mathsf{decomm}) \leftarrow \mathsf{VC.Commit}(\phi_0, \ldots, \phi_{n-1})$.
  - For each $i \in [n]$, compute $\lambda_i \leftarrow \mathsf{VC.Open}(\phi_i, i, \mathsf{decomm})$ and set $t_i \leftarrow (\phi_i, \lambda_i)$.
  - Output $(\mu, (t_0, \ldots, t_{n-1}))$ to $\mathcal{Z}$.

- Input $(\textsc{VerifyAll}, \mathsf{qid}, \mu, \mathbf{D}, (t_0, \ldots, t_{n-1}))$ from $\mathcal{Z}$
  where $\mathsf{qid} = (\mathsf{sid}, \mathsf{prover}, \mathsf{prover}, \mathsf{nonce})$:
  - Parse qid and abort if it has been used before, or if $\mathbf{D}$ or tags are invalid.
  - For each $i \in [n]$:
    - Send $(\textsc{VerifyPoly}, \mathsf{sid}_{\text{TPC}}, \phi_i, \mathbf{D}_{*,i})$ to $\mathcal{F}_{\text{TPC}}$ and wait for response.
    - Check $\mathsf{VC.Verify}(\mu, \phi_i, i, \lambda_i)$.
  - If any $\mathcal{F}_{\text{TPC}}$ response is $\perp$ or any check fails, output $(\textsc{VerifyAllResult}, \mathsf{qid}, \perp)$ to $\mathcal{Z}$; else output $\top$.

- Message $(\textsc{VerifierChallenge}, \mathsf{qid}, \mu, \text{PK}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$ from verifier:
  - Output $(\textsc{GetMatrixTags}, \mathsf{qid}, \mu)$ to $\mathcal{Z}$ and wait for response $(\textsc{MatrixTagsResult}, \mathsf{qid}, \mathbf{D}, \mathbf{t})$.
  - For each $i \in [n]$, send $(\textsc{VerifyPoly}, \mathsf{sid}_{\text{TPC}}, \phi_i, \mathbf{D}_{*,i})$ to $\mathcal{F}_{\text{TPC}}$ and wait for responses.
  - If $\mathbf{D}, \mathbf{t}$ are invalid, any response is $\perp$, or any $\mathsf{VC.Verify}$ fails: send $(\textsc{FirstResponse}, \mathsf{qid}, \perp)$ to verifier and Abort.
  - Compute $\mathbf{w}, \mathbf{x}, \mathbf{y}$ from $\mathbf{D}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$.
  - Send $(\textsc{FirstResponse}, \mathsf{qid}, \mathbf{t}, \mathbf{w}, \mathbf{x}, \mathbf{y})$ to verifier and wait for $(\textsc{SigmaReveal}, \mathsf{qid}, \sigma)$.
  - For each $i \in [n]$, send $(\textsc{OpenEval}, \mathsf{sid}_{\text{TPC}}, \mathsf{verifier}, \phi_i, \mathbf{D}_{*,i}, \sigma)$ to $\mathcal{F}_{\text{TPC}}$ and wait for response (OK).
  - Compute $\mathbf{z}$ from $\sigma, \mathbf{D}, \mathbf{a}$ and send $(\textsc{SecondResponse}, \mathsf{qid}, \mathbf{z})$ to verifier.

Figure 7: Protocol π_{VLA}, part 1 (Prover Specification)

π_{VLA}, continued from Fig. 7, responds to the following inputs and messages when acting as verifier:

- Input $(\text{GetTags}, \mathsf{qid}, \mu)$ from $\mathcal{Z}$:
  - Parse qid and abort if nonce has been used before.
  - Sample random $\mathbf{u} \leftarrow \mathsf{R}^m$.
  - Run $\mathsf{TagsAndVMP}(\mathsf{qid}, \mu, \mathbf{u})$, returning $(\mathsf{tags}, \mathbf{s})$.
  - If $\mathsf{tags} \neq \perp$, save $(\text{GotTags}, \mu)$ to memory.
  - Output $(\text{TagsResult}, \mathsf{qid}, \mathsf{tags})$ to $\mathcal{Z}$.

- Input $(\text{VectMatProd}, \mathsf{qid}, \mu, \mathbf{u})$ from $\mathcal{Z}$:
  - Abort if nonce is reused or $(\text{GotTags}, \mu)$ is not in memory.
  - Run $\mathsf{TagsAndVMP}(\mathsf{qid}, \mu, \mathbf{u})$, returning $(\mathsf{tags}, \mathbf{s})$ and output $(\text{VMPResult}, \mathsf{qid}, \mathbf{s})$ to $\mathcal{Z}$.

- Input $(\text{VerifyColumn}, \mathsf{sid}, \mu, \mathbf{r}, i, t)$ from $\mathcal{Z}$:
  - Abort if $(\text{GotTags}, \mu)$ is not in memory. Parse $t = (\phi, \lambda)$.
  - If $\mathsf{VC.Verify}(\mu, \phi, i, \lambda) \neq \top$, output $\perp$ to $\mathcal{Z}$.
  - Send $(\text{VerifyPoly}, \mathsf{sid}_{\text{TPC}}, \phi, \mathbf{r})$ to $\mathcal{F}_{\text{TPC}}$ and wait for response.
  - Output $\top$ to $\mathcal{Z}$ if $\mathcal{F}_{\text{TPC}}$ returns $\top$, else output $\perp$.

- **Subroutine** $\mathsf{TagsAndVMP}(\mathsf{qid}, \mu, \mathbf{u})$:
  1. If any parsing/decryption fails, return $(\perp, \perp)$ and Abort.
  2. Sample random $g_1, g_2, r, \rho, \sigma$ and generate LHE key pair (PK, SK).
  3. Compute $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$ from sampled values.
  4. Send $(\text{VerifierChallenge}, \mathsf{qid}, \mu, \text{PK}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$ to prover and wait for $(\text{FirstResponse}, \mathsf{qid}, \mathsf{tags}, \mathbf{w}, \mathbf{x}, \mathbf{y})$.
  5. Send $(\text{SigmaReveal}, \mathsf{qid}, \sigma)$ to prover and wait for $(\text{SecondResponse}, \mathsf{qid}, \mathbf{z})$.
  6. For each $i \in [n]$, wait to receive $(\text{EvalOpening}, \mathsf{sid}_{\text{TPC}}, \phi_i, \sigma, P_i(\sigma), \mathsf{prover})$ from $\mathcal{F}_{\text{TPC}}$.
  7. Decrypt $\mathbf{s}$ from $\mathbf{w}$ and $\hat{\mathbf{z}}$ from $\mathbf{z}$ using SK.
  8. For each $i \in [n]$, check:
     (a) $\mathsf{VC.Verify}(\mu, \phi_i, i, \lambda_i) = \top$
     (b) $x_i^{\rho} y_i = e(g_1, g_2)^{\phi \cdot \hat{z}_i}$
     (c) $\hat{z}_i = P_i(\sigma) \cdot U(\sigma) - s_i \sigma^{m-1}$
  9. Return $(\mathsf{tags}, \mathbf{s})$ to caller.

Figure 8: Protocol π_{VLA}, part 2 (Verifier Specification)

If we write $\mathcal{I}_j$ for the *index sets* given by $\mathcal{I}_j = \{k : \max(0, j+1-m) \leq k \leq \min(m-1, j)\}$, then we can write the terms of $Q_i(X)$ explicitly as

$$Q_i^{(U)}(X) = \sum_{j \in [2m-1] \setminus \{m-1\}} \left(\sum_{k \in \mathcal{I}_j} u_{m+k-j-1} D_{k,i} \right) X^j \tag{2}$$

Remember that the server knows $\mathbf{D}$ and therefore each $P_i(X)$ in the clear, but must not learn $\mathbf{u}$ or $U(X)$. In order to hide those, the client will also choose a random *masking polynomial* $R(X) \in \mathsf{R}[X]$ with $\deg R < m$ and element $\rho \in \mathsf{R}$, and will send group elements corresponding to $U(X) - R(X)$ and $\rho \cdot R(X)$ respectively. Crucially, we can see from Eq. (2) that the polynomial $Q_i^{(U)}$ is linear in the u_i's, and therefore we have

$$\rho \cdot Q_i^{(U-R)}(X) + Q_i^{(\rho R)}(X) = \rho \cdot Q_i^{(U)}(X). \tag{3}$$

The general idea in our protocol is that that the client chooses a random point σ and sends the server some group elements which allow it to compute two values, $g_T^{Q_i^{(U-R)}(\sigma)}$ and $g_T^{Q_i^{(\rho R)}(\sigma)}$. The server also computes homomorphically $\langle \mathbf{u}, \mathbf{D}_{*,i} \rangle$ as above. After receiving these, the client reveals σ explicitly to the server, which can then open the evaluation of $P_i(\sigma)$ using $\mathcal{F}_{\text{TPC}}$, as well as compute homomorphically $Q_i^{(U)}(\sigma)$. Finally, the client checks consistency of these values with Eqs. (1) and (3) above.

Notation in π_{VLA} for verifier/prover messages. To aid the reader in following the details of π_{VLA}, we list the messages sent between verifier and prover for the verifier subroutine TagsAndVMP, and their mathematical relationships when both parties are honest.

Verifier inputs:

- μ: Vector commitment
- $\mathbf{u} \in \mathsf{R}^m$: given
- $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$: random generators
- $\mathbf{r} \in \mathsf{R}^m$: random masks
- $\rho, \sigma \in \mathsf{R}$: random values

Sent by client in VerifierChallenge:

- $\mathbf{a} = \mathsf{LHEnc}(\mathbf{u})$
- $\mathbf{b} \in \mathbb{G}_1^{2m-2} = \left({g_1}^{\sigma^i} \right)_{i \in [2m-1] \setminus \{m-1\}}$
- $\mathbf{c} \in \mathbb{G}_2^m = {g_2}^{\mathbf{u}-\mathbf{r}}$
- $\mathbf{d} \in \mathbb{G}_2^m = {g_2}^{\rho \mathbf{r}}$

Sent by the server in FirstResponse:

- Tags $(t_i = (\phi_i, \Lambda_i))_{i \in [n]}$, where each ϕ_i is a polynomial commitment to $P_i(X)$, and each Λ_i is a vector commitment opening of ϕ_i w.r.t. VC commitment μ
- $\mathbf{w}$ = encryption of $\mathbf{u}^\intercal \mathbf{D}$, computed homomorphically using $\mathbf{a}$
- $\mathbf{x} \in \mathbb{G}_T^n = \left({g_T}^{Q_i^{(U-R)}(\sigma)} \right)_{i \in [n]}$
- $\mathbf{y} \in \mathbb{G}_T^n = \left({g_T}^{Q_i^{(\rho R)}(\sigma)} \right)_{i \in [n]}$

Sent by the client in SigmaReveal:

- $\sigma \in \mathsf{R}$

Table 2: Complexity analysis for π_{VLA}, ignoring polylog or security parameter factors

Operation	Verifier runtime	Prover runtime	Communication
MatrixCommit	—	mn	—
VerifyAll	—	mn	—
GetTags	$m+n$	mn	$m+n$
VecMatProd	$m+n$	mn	$m+n$
VerifyColumn	m	—	—
	Verifier	**Prover**	
Storage	—	mn	

Sent by the server in SecondResponse:

- Verified evaluations of each $P_i(\sigma)$ w.r.t. ϕ_i and μ, for $i \in [n]$
- $\mathbf{z}$ = encryption of $\left(Q_i^{(U)}(\sigma)\right)_{i\in[n]}$ computed homomorphically

Final client checks:

- Vector commitment μ opens to each ϕ_i using proof Λ_i at index i
- $\mathbb{G}_T$ elements in $\mathbf{x}$ and $\mathbf{y}$ are consistent with ciphertexts in $\mathbf{z}$ according to Eq. (3)
- Each evaluation $Q_i^{(u)}(\sigma)$ in $\mathbf{z}$ is consistent with Eq. (1) according to verified evaluations of $P_i(\sigma)$ and dot products from $\mathbf{w}$

6.2 Analysis of π_{VLA}

We show that a real-world execution, with an ideal copy of $\mathcal{F}_{\text{TPC}}$ for polynomial commitments, is indistinguishable by any probabilistic polynomial time (p.p.t.) environment and adversary $\mathcal{Z} + \mathcal{A}$, from an ideal-world execution using the simulator of Fig. 9.

Theorem 1. *Protocol π_{VLA} UC-realizes $\mathcal{F}_{\text{VLA}}$ in the $\mathcal{F}_{\text{TPC}}$-hybrid model against static, malicious corruptions, under Assumption 1, when using CDH groups, and CPA-secure LHE.*

6.3 Proof of Theorem 1

The complete simulator description is given in Fig. 9.

Simulating honest parties. Most of the work of the simulator is to simulate the real-world adversarial leakage based on the timing of ideal-world messages. This is crucial and intricate, particularly to match up the same timing of the (simulated) protocol messages, with the (actual) outputs and inputs with honest parties to the environment $\mathcal{Z}$.

This is largely straightforward and we leave it to the reader to verify correct simulation of protocol message flows.

Malicious verifier. When the verifier is corrupt, the simulation is also relatively easy. Note that all valid sets of commitment μ, matrix $\mathbf{D}$, and tags, are leaked to the simulator by the $\mathcal{F}_{\text{VLA}}$ in the ideal world. So $\mathcal{S}_{\text{VLA}}$ can perfectly simulate the honest server's behavior to the corrupted verifier, by simply running our protocol using the same exact inputs.

Malicious prover. The more challenging case is when the prover is corrupt. Here, the simulator in Fig. 9 intercepts the ideal-world message GetMatrixTags from $\mathcal{F}_{\text{VLA}}$ to the (corrupted) prover, and must then interact with the adversary to either extract the full matrix $\mathbf{D}$ and tags, or detect failure and return $\perp$, in a way that is consistent with what the real protocol would do interacting with the same adversary.

There are three challenges for our proof: (1) showing how extraction of $\mathbf{D}$ and tags is performed; (2) proving that nothing about the honest client's $\mathbf{u}$ vector is leaked to the

Simulator $\mathcal{S}_{\text{VLA}}$ for π_{VLA} with parameters L_{TPC} and L_{VC}, maintains an honest internal instance of $\mathcal{F}_{\text{TPC}}$ parameterized by $\mathsf{R} = \mathbb{F}_q$ and responds to the following messages:

Handling leakage from honest parties:

- **Backdoor query** ($\textsc{GenMC}, \mathsf{sid}, \mathbf{D}, \mathsf{prover}$) from $\mathcal{F}_{\text{VLA}}$:
  1. Simulate π_{VLA} locally on input ($\textsc{MatrixCommit}, \mathsf{sid}, \mathbf{D}$) acting as prover with simulated $\mathcal{F}_{\text{TPC}}$ to obtain (μ, tags).
  2. If $(\mu, \mathsf{tags}) \neq \perp$, record $(\text{extracted}, \mathsf{sid}, \mu, \mathbf{D}, \mathsf{tags})$ and return (μ, tags) to $\mathcal{F}_{\text{VLA}}$.
- **Backdoor query** ($\textsc{CheckMC}, \mathsf{qid}, \mu, \mathbf{D}, \mathsf{tags}$) from $\mathcal{F}_{\text{VLA}}$:
  1. If $(\text{corruptProver}, \mathsf{qid}, \mathsf{result})$ is recorded, return result to $\mathcal{F}_{\text{VLA}}$ and Abort.
  2. Assert $\mathbf{D}, \mathsf{tags}$ are valid. For $i \in [n]$: simulate $(\textsc{VerifyPoly}, \mathsf{sid}_{\text{TPC}}, \phi_i, \mathbf{D}_{*,i})$ to $\mathcal{F}_{\text{TPC}}$ from prover and check $\mathsf{VC.Verify}(\mu, \phi_i, i, \lambda_i)$.
  3. If any check fails: record $(\text{checkFailed}, \mathsf{qid})$ and return $\perp$ to $\mathcal{F}_{\text{VLA}}$; else: record $(\text{extracted}, \mathsf{sid}, \mu, \mathbf{D}, \mathsf{tags})$ and $(\text{checkPassed}, \mathsf{qid})$, then return $\top$.
- ($\textsc{HonestMsg}, \mathcal{F}_{\text{VLA}}, \mathsf{prover}, \textsc{GetMatrixTags}, \mathsf{qid}$) from Router:
  1. Send ($\textsc{HonestMsg}, \mathsf{verifier}, \mathsf{prover}, \textsc{VerifierChallenge}, \mathsf{qid}$) to $\mathcal{A}$ via backdoor and wait for Deliver; then return Deliver to Router.
- ($\textsc{HonestMsg}, \mathcal{F}_{\text{VLA}}, \mathsf{prover}, \textsc{VerifyAllResult}, \mathsf{qid}$) from Router:
  1. If $(\text{checkFailed/Passed}, \mathsf{qid})$ not recorded: for $i \in [n]$, send ($\textsc{HonestMsg}, \mathsf{prover}, \mathcal{F}_{\text{TPC}}$, $\textsc{VerifyPoly}, \mathsf{sid}_{\text{TPC}}$) and its response to $\mathcal{A}$ (with Delivers).
  2. Return Deliver to Router.
- ($\textsc{HonestMsg}, \mathcal{F}_{\text{VLA}}, \mathsf{verifier}, \textsc{TagsResult}, \mathsf{qid}$) from Router:
  1. If $(\text{corruptProver}, \dots)$ recorded, return Deliver and Abort.
  2. If $(\text{checkFailed/Passed}, \mathsf{qid})$ not recorded: for $i \in [n]$, send ($\textsc{HonestMsg}, \mathsf{prover}, \mathcal{F}_{\text{TPC}}$, $\textsc{VerifyPoly}, \mathsf{sid}_{\text{TPC}}$) and response to $\mathcal{A}$ (with Delivers), then record $(\text{checkPassed}, \mathsf{qid})$.
  3. Send ($\textsc{HonestMsg}, \mathsf{prover}, \mathsf{verifier}, \textsc{FirstResponse}, \mathsf{qid}$) to $\mathcal{A}$ (wait for Deliver).
  4. If $(\text{checkPassed}, \mathsf{qid})$ is recorded: send ($\textsc{HonestMsg}, \mathsf{verifier}, \mathsf{prover}, \textsc{SigmaReveal}, \mathsf{qid}$) and ($\textsc{HonestMsg}, \mathsf{prover}, \mathsf{verifier}, \textsc{SecondResponse}, \mathsf{qid}$) to $\mathcal{A}$ (with Delivers).
  5. Return Deliver to Router.
- ($\textsc{HonestMsg}, \mathsf{verifier}, \mathcal{F}_{\text{VLA}}, \textsc{VerifyColumn}, \mathsf{sid}$) from Router:
  1. Send ($\textsc{HonestMsg}, \mathsf{verifier}, \mathcal{F}_{\text{TPC}}, \textsc{VerifyPoly}, \mathsf{sid}_{\text{TPC}}$) and its response to $\mathcal{A}$, waiting for Delivers; then return Deliver to Router.

Handling corrupted Prover:

- ($\textsc{LeakedMsg}, \mathcal{F}_{\text{VLA}}, \mathsf{prover}, \textsc{GetMatrixTags}, \mathsf{qid}, \mu$) from Router:
  1. Simulate π_{VLA} acting as $\mathsf{verifier}$ on input ($\textsc{GetTags}, \mathsf{sid}, \mu$) with $\mathcal{F}_{\text{TPC}}$ to produce tags.
  2. If $\mathsf{tags} = \perp$, set $\mathsf{result} \leftarrow \perp$.
  3. Else:
     (a) For each $i \in [n]$, lookup $(\text{pcomm}, \mathsf{sid}_{\text{TPC}}, \phi_i, P_i)$ in simulated $\mathcal{F}_{\text{TPC}}$ memory.
     (b) Form matrix $\mathbf{D}$ where column i is the coefficients of P_i. Set $\mathsf{result} \leftarrow (\mathbf{D}, \mathsf{tags})$.
  4. Record $(\text{corruptProver}, \mathsf{qid}, \mathsf{result})$ and send ($\textsc{CorruptMsg}, \mathsf{prover}, \mathcal{F}_{\text{VLA}}, \textsc{MatrixTagResult}$, $\mathsf{qid}, \mathsf{result}$) to Router via backdoor.

Handling corrupted Verifier:

- ($\textsc{CorruptMsg}, \mathsf{verifier}, \mathsf{prover}, \textsc{VerifierChallenge}, \mathsf{qid}, \mu, \text{PK}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$) from $\mathcal{A}$:
  1. Send ($\textsc{CorruptMsg}, \mathsf{verifier}, \mathcal{F}_{\text{VLA}}, \textsc{GetTags}, \mathsf{qid}, \mu$) to Router; wait for ($\textsc{LeakedMsg}, \mathcal{F}_{\text{VLA}}, \mathsf{verifier}, \textsc{TagsResult}, \mathsf{qid}, \mathsf{tags}$).
  2. If $(\text{checkFailed/Passed}, \mathsf{qid})$ not recorded: for $i \in [n]$, send ($\textsc{HonestMsg}, \mathsf{prover}, \mathcal{F}_{\text{TPC}}, \textsc{VerifyPoly}, \mathsf{sid}_{\text{TPC}}$) and response to $\mathcal{A}$ (with Delivers).
  3. Lookup $(\text{extracted}, \mathsf{sid}, \mu, \mathbf{D}, \mathsf{tags})$. If $\mathsf{tags} = \perp$ or lookup fails: send ($\textsc{LeakedMsg}, \mathsf{prover}, \mathsf{verifier}, \textsc{FirstResponse}, \mathsf{qid}, \perp$) to $\mathcal{A}$ and Abort.
  4. Execute π_{VLA} steps 4–8 for $\textsc{VerifierChallenge}$ as prover with simulated $\mathcal{F}_{\text{TPC}}$ and corrupted $\mathsf{verifier}$.

Figure 9: Simulator $\mathcal{S}_{\text{VLA}}$

adversary; and (3) demonstrating that, upon success, the honest verifier will receive the correct $\mathbf{s}^\intercal = \mathbf{u}^\intercal\mathbf{D}$ value (i.e., *soundness*).

Extracting D and tags. For (1), extraction is straightforward when the protocol succeeds. In the FirstResponse message send by the prover in our protocol, the tags $t_i = (\phi_i, \Lambda_i)$ are directly sent to the verifier. Our simulator receives those tags, and then can use the $\mathcal{F}_{\text{TPC}}$ polynomial commitment sub-functionality to extract the each column polynomial $P_i(X)$ and build the full matrix $\mathbf{D}$.

Indistinguishablity of u. For (2), note that the simulator *always* simulates a GetTags operation in the protocol, using a random $\mathbf{u}$ vector, regardless of whether the honest client in the ideal world is actually performing a GetTags or VectMatProd operation. In the real protocol, the honest verifier performs the same call to TagsAndVMP in both cases, with the only difference being the choice of $\mathbf{u}$ vector.

In the protocol, observe that the only values sent to the verifier are the vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$ and scalar σ. Nothing is revealed by random σ and $\mathbf{b} = g_1^{\sigma^i}$.

This leaves us with $\mathbf{a} = \mathsf{LHEnc}(\mathbf{u})$, $\mathbf{c} = g_2^{\mathbf{u}-\mathbf{r}}$, and $\mathbf{d} = g_2^{\rho\mathbf{r}}$. Because ρ is never revealed and $\mathbf{r}$ is chosen randomly, the group elements are indistinguishable from random (otherwise, the DDH assumption is broken in $\mathbb{G}_2$), and the CPA security of LHEnc guarantees $\mathbf{a}$ is indistinguishable from random encryptions.

Soundness via reduction to missing exponent assumption. Finally, we prove soundness of the simulation by an explicit reduction to the missing exponent problem of Assumption 1.

By way of contradiction, assume there exists a p.p.t. adversary $\mathcal{A}$ that, with non-negligible probability, can break soundness in the honest verifier's TagsAndVMP subroutine. That is, the verifier, on input $\mathbf{u}$, will return a list of tags and $\mathbf{s}$ such that $\mathbf{s}^\intercal \neq \mathbf{u}^\intercal\mathbf{D}$, where $\mathbf{D}$ is the matrix extracted from the column polynomials in the tag commitments, using $\mathcal{F}_{\text{TPC}}$ as above.

We will use such an $\mathcal{A}$, in a modified simulator, to break the missing exponent game with non-negligible probability. Take the following challenge for the missing exponent game, with the missing index being $m-1$:

$$1^\kappa, g_1, g_2, g_1^s, g_1^{s^2}, \ldots, g_1^{s^{m-2}}, g_1^{s^m}, g_1^{s^{m+1}}, \ldots, g_1^{s^{2m-2}}$$

The challenge is to compute $g_T^{s^{m-1}}$. Rather than honestly simulating the TagsAndVMP subroutine with $\mathcal{A}$, the modified simulator will instead use values from the missing exponent game. Namely, rather than choosing random exponents σ, we implicitly set $\sigma = s$, the *unknown* exponent from the missing exponent challenge. In the first VerifierChallenge message to $\mathcal{A}$, choose random vectors $\mathbf{u}, \mathbf{r} \in \mathsf{R}^m$ and random exponent ρ and set

- $\mathbf{a} = \mathsf{LHEnc}(\mathbf{u})$
- $\mathbf{b} = \left(g_1^{s^i}\right)_{i \in [2m-1] \setminus \{m-1\}}$ from missing exponent challenge
- $\mathbf{c} = g_2^{\mathbf{u}-\mathbf{r}}$
- $\mathbf{d} = g_2^{\rho\mathbf{r}}$

We emphasize that because the missing exponent game chooses s randomly, these challenge vectors are drawn from the identical distribution as they would have been in the actual protocol. Adversary $\mathcal{A}$ will send in the FirstResponse message a vector of LHE ciphertexts $\mathbf{w}$, and two vectors in $\mathbb{G}_T^n$, $\mathbf{x}$ and $\mathbf{y}$, as well as all the tags t_i.

At this point, the modified simulator chooses a random index $i \in [n]$, computes $g_T^{Q_i^{(U)}(s)}$ using the values from the missing exponent game, according to (2), and then outputs

$$\alpha = \left(\frac{\mathbf{x}_i^\rho \mathbf{y}_i}{g_T^{\rho \cdot Q_i^{(U)}(s)}} \right)^{1/(\langle \mathbf{u}, \mathbf{D}_{*,i} \rangle - \mathbf{s}_i)} \tag{4}$$

as the proposed solution to the missing exponent game. To see why α has a non-negligible chance to win the missing exponent game, we consider what *would have* happened if the protocol continued, and if $\mathcal{A}$ would have broken soundness by tricking the honest verifier to produce a vector $\mathbf{s}$ which is incorrect at index i. By assumption and because n is obviously polynomial, this *would have* occurred with non-negligible probability.

The next message sent by the honest verifier to $\mathcal{A}$ would be SigmaReveal with $\sigma = s$. Of course, our simulator does not actually know the value of s here and could not actually have continued the protocol, but this is irrelevant; the solution α to the missing exponent game has already been determined, and we are just analyzing the (probable) behavior of $\mathcal{A}$. After receiving s, $\mathcal{A}$ would use $\mathcal{F}_{\text{TPC}}$ to open each polynomial evaluation $P_i(s)$, and would also send a final vector of ciphertexts $\mathbf{z}$.

Write $\zeta = \mathsf{LHDec}(\mathbf{z}_i)$. To break soundness, the values sent by $\mathcal{A}$ must pass these checks:

- $\mathbf{x}_i^{\rho} \cdot \mathbf{y}_i = {g_T}^{\rho \cdot \zeta}$
- $\zeta = P_i(s)U(s) - \mathbf{s}_i s^{m-1}$

By the definition of $Q_i^{(U)}(s)$ in (1), the second equation implies

$$\zeta - Q_i^{(U)}(s) = (\langle \mathbf{u}, \mathbf{D}_{*,i} \rangle - \mathbf{s}_i) \cdot s^{m-1}$$

Raising g_T^{ρ} to both sides, combining with the first check equation and with the definition of α in (4), gives $\alpha = g_T^{s^{m-1}}$, the desired solution to the missing exponent game.

7 The CVPIR Protocol π_{CVPIR}

7.1 Protocol Description

A full description of our protocol *Compact and Verified PIR* (π_{CVPIR}) is provided in Figs. 11 to 13. Note that our protocol is inherently multi-user and multi-server, and so the single protocol contains all commands needed to act in either role. Indeed, the same party could be a client for one database and a server for another, or could change roles at any time.

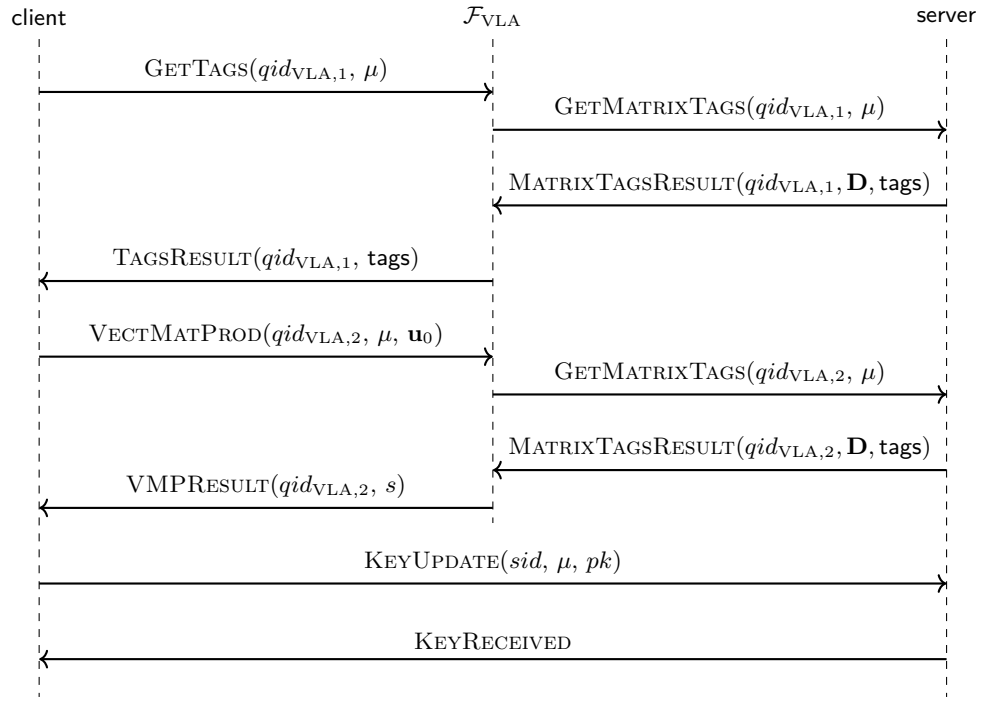
Timing diagrams. Fig. 10 shows two timing diagrams to help readers understand the protocol details.

The Register phase establishes reusable verification tags tied to a fixed database. Fig. 10(a) shows the timing diagram for this phase. The client first queries $\mathcal{F}_{\text{VLA}}$ for tags via GetTags. To answer this, $\mathcal{F}_{\text{VLA}}$ requests matrix tags from the server (via GetMatrixTags) and returns them to the client (TagsResult). The client then invokes $\mathcal{F}_{\text{VLA}}$ on VectMatProd; to answer, $\mathcal{F}_{\text{VLA}}$ requests the same underlying matrix material from the server again (a second GetMatrixTags) and returns the result. This "fetch once, use twice, then delete" pattern avoids long-term server state while ensuring that both steps are consistent with a single database.

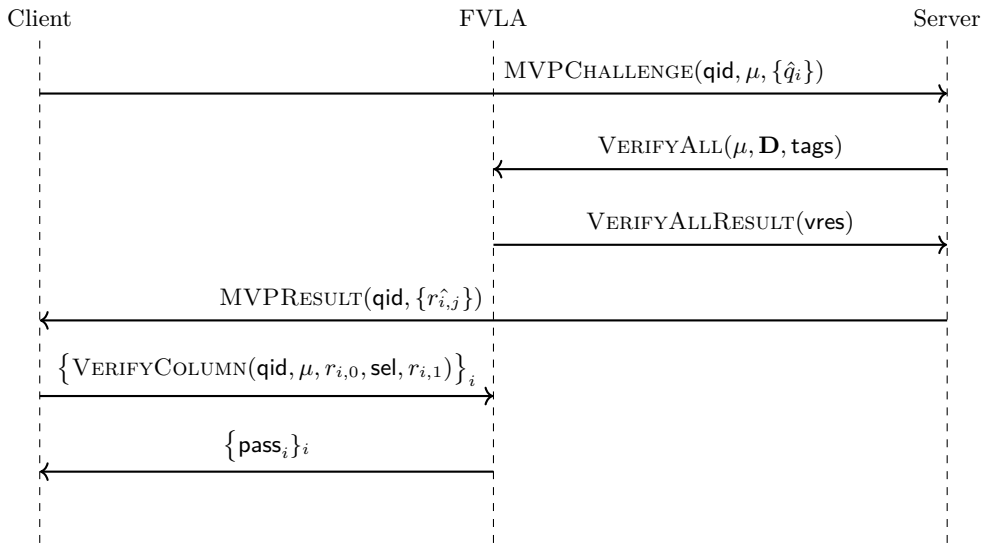
During Lookup, shown in Fig. 10(b), the client sends MVPChallenge to the server and waits for MVPResult. Upon receiving MVPChallenge, the server invokes $\mathcal{F}_{\text{VLA}}$ on VerifyAll and proceeds only if it receives VerifyAllResult indicating success. The server then computes and returns MVPResult to the client. After decrypting the response, the client invokes $\mathcal{F}_{\text{VLA}}$ on VerifyColumn (and waits for the corresponding response) to decide whether to accept the retrieved value; otherwise it aborts or outputs a fixed fallback value.

7.2 Security

Theorem 2 provides a formal statement of the security of our vPIR protocol π_{CVPIR}.



(a) Register



(b) Lookup

Figure 10: Timing diagrams for protocol π_{CVPIR}. (a) Register: $\mathcal{F}_{\text{VLA}}$ fetches tags and reuses the same derived matrix for an immediate consistency check. (b) Lookup: $\mathcal{F}_{\text{VLA}}$ checks consistency with μ and mediates the matrix–vector product challenge/response and verification.

The π_{CVPIR} protocol is parameterized by a prime p, a ring R such that $\{0, \ldots, p-1\} \subseteq R$, and bitlengths L and L_{tag}. It uses the sub-functionality $\mathcal{F}_{\text{VLA}}$ with ring R and tag bitlength L_{tag}, and local primitives including an FHE scheme with plaintext modulus p and a pseudorandom generator (PRG).

The structured session identifier is $\mathsf{sid} = (L, n, (\lambda, k, \alpha, \mathsf{nonce}))$, where L is the bitlength of PIR entries, n is the number of database entries, λ is the statistical security parameter, k is the number of precomputed check vectors stored by the client, and $\alpha = (\alpha_1, \ldots, \alpha_\ell)$ is a list of compressed vector lengths used for query compression, satisfying $\prod_i \alpha_i \geq n$.

We define the following derived parameters:

$$m \leftarrow \lceil L / \mathsf{bitlen}(p) \rceil, \qquad \mathsf{sid}_{\mathsf{VLA}} \leftarrow (m, n, \texttt{"CVPIR\_VLA"} \| \mathsf{nonce}), \qquad \tau \leftarrow \lceil L_{\mathsf{tag}} / \mathsf{bitlen}(p) \rceil.$$

We say db is *valid* if $\mathsf{db} \in (\{0,1\}^L)^n$. We will also use the following notations:

- Throughout the protocol, digest refers to the commitment μ output by $\mathcal{F}_{\text{VLA}}$, and aux refers to the corresponding tags obtained from $\mathcal{F}_{\text{VLA}}$.
- $\mathsf{DeriveMatrix}(\mathsf{db})$ generates a matrix $\mathbf{D} \in R^{m \times n}$ where each entry lies in $[0, p)$, and the i-th column of $\mathbf{D}$ corresponds to $\mathsf{db}[i]$.
- $\mathsf{DeriveMatrix}(\mathsf{tags})$ similarly generates a matrix $\mathbf{T} \in R^{\tau \times n}$.
- $\mathsf{SampleCompVect}(\mathsf{seed}, i) \to \mathsf{cv}$ samples a random compressed vector with entries in $\mathbb{F}_p$, whose shape is determined by α, and which is sampled deterministically from the tuple (seed, i).
- $\mathsf{Decompress}(\mathsf{cv}) \to \mathbf{v}$ decompresses a compressed vector to a vector $\mathbf{v} \in \mathbb{F}_p^n$.
- $\mathsf{OneHot}(\mathsf{sel}) \to \mathsf{cv}$ computes a compressed representation of the one-hot vector encoding of index $\mathsf{sel} \in [n]$.
- $\mathsf{HECompressedMVP}(\text{PK}, \mathbf{M}, \mathsf{cv}) \to \mathbf{r}$ homomorphically multiplies the matrix $\mathbf{M}$ with the homomorphically decompressed vector cv.
- $\mathsf{decode} : \mathbb{F}_p \to R$ maps an element $x \in \mathbb{F}_p$ to the corresponding element in R; that is, $\mathsf{decode}(x)$ is defined to be the natural embedding of x into R.

When acting as server, π_{CVPIR} responds to the following inputs:

- Input $(\text{GetDigest}, \mathsf{sid}, \mathsf{db})$ from environment $\mathcal{Z}$:
  - Assert that db is valid.
  - Compute $\mathbf{D} \leftarrow \mathsf{DeriveMatrix}(\mathsf{db})$.
  - Send query $(\text{MatrixCommit}, \mathsf{sid}_{\mathsf{VLA}}, \mathbf{D})$ to $\mathcal{F}_{\text{VLA}}$ and wait for response (μ, tags).
  - Return output (μ, tags) to $\mathcal{Z}$.
- Message $(\text{GetMatrixTags}, \mathsf{qid}_{\mathsf{VLA}}, \mu)$ from $\mathcal{F}_{\text{VLA}}$ via Router, where $\mathsf{qid}_{\mathsf{VLA}} = (\mathsf{sid}_{\mathsf{VLA}}, \mathsf{server}, \mathsf{client}, \mathsf{nonce} \| \texttt{"1"})$:
  - Set $\mathsf{qid} = (\mathsf{sid}, \mathsf{server}, \mathsf{client}, \mathsf{nonce})$.
  - Output $(\text{GetDB}, \mathsf{qid}, \mathsf{nonce})$ to $\mathcal{Z}$ and wait for corresponding input $(\text{DBResponse}, \mathsf{qid}, \mathsf{db}, \mathsf{tags})$.
  - Assert that db is valid and $\mathsf{tags} \neq$ "corrupt".
  - Set $\mathbf{D} \leftarrow \mathsf{DeriveMatrix}(\mathsf{db})$.
  - Save $(\text{ProvidedMatrix}, \mathsf{qid}, \mathbf{D}, \mathsf{tags})$ to memory.
  - Send $(\text{MatrixTagsResult}, \mathsf{qid}_{\mathsf{VLA}}, \mathbf{D}, \mathsf{tags})$ to $\mathcal{F}_{\text{VLA}}$.
- Message $(\text{GetMatrixTags}, \mathsf{qid}_{\mathsf{VLA}}, \mu)$ from $\mathcal{F}_{\text{VLA}}$ via Router, where $\mathsf{qid}_{\mathsf{VLA}} = (\mathsf{sid}_{\mathsf{VLA}}, \mathsf{server}, \mathsf{client}, \mathsf{nonce} \| \texttt{"2"})$:
  - Set $\mathsf{qid} = (\mathsf{sid}, \mathsf{server}, \mathsf{client}, \mathsf{nonce})$.
  - Lookup and remove $(\text{ProvidedMatrix}, \mathsf{qid}, \mathbf{D}, \mathsf{tags})$ from memory.
  - Send $(\text{MatrixTagsResult}, \mathsf{qid}_{\mathsf{VLA}}, \mathbf{D}, \mathsf{tags})$ to $\mathcal{F}_{\text{VLA}}$.

Figure 11: Protocol π_{CVPIR} for $\mathcal{F}_{\text{PIR}}$, part 1

When acting as server (continued from Fig. 11), π_{CVPIR} responds to the following inputs:

- Message $(\text{MVPCHALLENGE}, \mathsf{qid}, \mu, \{\widehat{q_i}\}_{i\in[\lambda]})$ from client via Router, where $\mathsf{qid} = (\mathsf{sid}, \mathsf{server}, \mathsf{client}, \mathsf{nonce})$:
  - Output $(\text{GETDB}, \mathsf{qid}, \mu)$ to $\mathcal{Z}$ and wait for corresponding input $(\text{DBRESPONSE}, \mathsf{qid}, \mathsf{db}, \mathsf{tags})$.
  - Assert db is valid and $\mathsf{tags} \neq$ "corrupt".
  - Set $\mathbf{D} \leftarrow \mathsf{DeriveMatrix}(\mathsf{db})$ and $\mathbf{T} \leftarrow \mathsf{DeriveMatrix}(\mathsf{tags})$.
  - Let $\mathsf{qid}_{\mathsf{VLA}} = (\mathsf{sid}_{\mathsf{VLA}}, \mathsf{server}, \mathsf{client}, \mathsf{nonce})$ and send $(\text{VERIFYALL}, \mathsf{qid}_{\mathsf{VLA}}, \mu, \mathbf{D}, \mathsf{tags})$ to $\mathcal{F}_{\text{VLA}}$.
  - Wait for corresponding message $(\text{VERIFYALLRESULT}, \mathsf{qid}_{\mathsf{VLA}}, \mathsf{vres})$.
  - Lookup $(\text{CLIENTKEY}, \mathsf{sid}, \mathsf{client}, \mu, \text{PK})$ in memory.
  - If $\mathsf{vres} = \perp$ or lookup failed, send $(\text{MVPRESULT}, \mathsf{qid}, \perp)$ to client and abort.
  - For $i \in [\lambda]$, set
    $$\widehat{r_{i,0}} \leftarrow \mathsf{HECompressedMVP}(\text{PK}, \mathbf{D}, \widehat{q_i})$$
    $$\widehat{r_{i,1}} \leftarrow \mathsf{HECompressedMVP}(\text{PK}, \mathbf{T}, \widehat{q_i})$$
  - Send $(\text{MVPRESULT}, \mathsf{qid}, \{\widehat{r_{i,j}}\}_{i,j})$.
- Message $(\text{KEYUPDATE}, \mathsf{sid}, \mu, \text{PK})$ from client via Router:
  - Store $(\text{CLIENTKEY}, \mathsf{sid}, \mathsf{client}, \mu, \text{PK})$ in memory, replacing any previous entry.
  - Return (KEYRECEIVED) to client via Router.

When acting as client (continued from Fig. 11), π_{CVPIR} responds to the following inputs:

- **Input** $(\text{REGISTER}, \mathsf{qid}, \mu)$ from environment $\mathcal{Z}$:
  - Parse qid as $(\mathsf{sid}, \mathsf{server}, \mathsf{client}, \mathsf{nonce})$ and assert it has never been used before.
  - Choose a random PRG seed seed, then sample $\mathbf{u}_0 \in \mathbb{F}_p^m$, $\mathbf{u}_1 \in \mathbb{F}_p^\tau$, and for each $i \in [k\lambda]$ set $\mathsf{cv}_i \leftarrow \mathsf{SampleCompVect}(p, \alpha, \mathsf{seed}, i)$.
  - $\mathsf{qid}_{\mathsf{VLA},1} = (\mathsf{sid}_{\mathsf{VLA}}, \mathsf{server}, \mathsf{client}, \mathsf{nonce}\|\text{"1"})$, $\mathsf{qid}_{\mathsf{VLA},2} = (\mathsf{sid}_{\mathsf{VLA}}, \mathsf{server}, \mathsf{client}, \mathsf{nonce}\|\text{"2"})$.
  - Send $(\text{GETTAGS}, \mathsf{qid}_{\mathsf{VLA},1}, \mu)$ to $\mathcal{F}_{\text{VLA}}$ and wait for $(\text{TAGSRESULT}, \mathsf{qid}_{\mathsf{VLA},1}, \mathsf{tags})$ from $\mathcal{F}_{\text{VLA}}$.
  - Send $(\text{VECTMATPROD}, \mathsf{qid}_{\mathsf{VLA},2}, \mu, \mathbf{u}_0)$ to $\mathcal{F}_{\text{VLA}}$ and wait for $(\text{VMPRESULT}, \mathsf{qid}_{\mathsf{VLA},2}, \mathbf{s}(\mathbf{D}))$ from $\mathcal{F}_{\text{VLA}}$.
  - If $\mathsf{tags} = \perp$ or $\mathbf{s}(\mathbf{D}) = \perp$, output $(\text{REGISTERRESULT}, \mathsf{qid}, \perp)$ to $\mathcal{Z}$ and Abort.
  - Set $\mathbf{T} \leftarrow \mathsf{DeriveMatrix}(\mathsf{tags})$ and $\mathbf{s} \leftarrow \mathbf{s}(\mathbf{D}) + \mathbf{u}_1 \cdot \mathbf{T}$.
  - For each $i \in [k\lambda]$, compute $c_i \leftarrow \langle \mathbf{s}, \mathsf{Decompress}(\mathsf{cv}_i) \rangle$.
  - Save $(\text{PRECOMP}, \mathsf{sid}, \mu, (c_i)_{i\in[k\lambda]}, \mathsf{seed})$ to memory.
  - Call subroutine $\text{SENDKEY}(\mathsf{sid}, \mu, \mathsf{server})$.
  - Output $(\text{REGISTERRESULT}, \mathsf{qid}, \top)$ to $\mathcal{Z}$.

Figure 12: Protocol π_{CVPIR} for $\mathcal{F}_{\text{PIR}}$, part 2

When acting as client (continued from Fig. 12), π_{CVPIR} responds to the following inputs:

- **Input** (LOOKUP, qid, μ, sel) from environment $\mathcal{Z}$:
  - Parse qid as (sid, server, client, nonce) and assert it has never been used before, (PRECOMP, sid, μ, $(c_i)_{i\in[k\lambda]}$, seed) is in memory, and sel $\in [n]$.
  - Call subroutine GETKEY(sid, μ, server), returning (PK, SK, offset).
  - Regenerate $\mathbf{u}_0, \mathbf{u}_1$ using seed and sample fresh random $b \in \{0,1\}^\lambda$.
  - For each $i \in [\lambda]$ do:
    - If $b_i = 0$, set $q_i \leftarrow \mathsf{OneHot}(\mathsf{sel})$.
    - Else set $q_i \leftarrow \mathsf{SampleCompVect}(p, \alpha, \mathsf{seed}, \mathsf{offset} \cdot \lambda + i)$.
  - Send (MVPCHALLENGE, qid, μ, $(\mathsf{FHEnc}(\text{PK}, q_i))_{i\in[\lambda]}$) to server via Router and wait for corresponding message (MVPRESULT, qid, $(\widehat{r}_{i,j})_{i\in[\lambda],\, j\in\{0,1\}}$) from server via Router.
  - Set entry $\leftarrow 0^L$ and cheating $\leftarrow$ False.
  - For each $i \in [\lambda]$ and $j \in \{0,1\}$, $r_{i,j} \leftarrow \mathsf{FHDec}(\text{SK}, \widehat{r}_{i,j})$.
  - For each $i \in [\lambda]$ do:
    - Send (VERIFYCOLUMN, $\mathsf{sid}_{\mathsf{VLA}}$, μ, $r_{i,0}$, sel, decode$(r_{i,1})$) to $\mathcal{F}_{\text{VLA}}$ and wait for response (pass).
    - If $b_i = 0$ and pass $= \top$, set entry $\leftarrow r_{i,0}$.
    - If $b_i = 1$ and $\langle \mathbf{u}_0, r_{i,0}\rangle + \langle \mathbf{u}_1, r_{i,1}\rangle \neq c_{\mathsf{offset}\cdot\lambda+i}$, set cheating $\leftarrow$ True.
  - Call subroutine USEKEY(sid, μ, server).
  - If cheating $=$ True, output (LOOKUPRESULT, qid, $\perp$) to $\mathcal{Z}$.
  - Else output (LOOKUPRESULT, qid, entry) to $\mathcal{Z}$.
- **Subroutine** SENDKEY(sid, μ, server):
  - (PK, SK) $\leftarrow \mathsf{FHKeyGen}()$.
  - Save (KEYCOUNT, sid, μ, server, PK, SK, 0) to memory.
  - Send query (KEYUPDATE, sid, μ, PK) to server via Router and wait for reply (KEYRECEIVED).
- **Subroutine** GETKEY(sid, μ, server):
  - Lookup (KEYCOUNT, sid, μ, server, PK, SK, count) in memory.
  - If not found, call SENDKEY(sid, μ, server) and return to the previous step.
  - Return (PK, SK, count) to the caller.
- **Subroutine** USEKEY(sid, μ, server):
  - Lookup (KEYCOUNT, sid, μ, server, PK, SK, count) in memory.
  - If count $+ 1 = k$, then call SENDKEY(sid, μ, server).
  - Else update (KEYCOUNT, sid, μ, server, PK, SK, count $+ 1$) in memory.

Figure 13: Protocol π_{CVPIR} for $\mathcal{F}_{\text{PIR}}$, part 3

Theorem 2. *Protocol π_{CVPIR} (Figs. 11 to 13) UC-realizes $\mathcal{F}_{\text{PIR}}$ (Fig. 1) in the $(\mathcal{F}_{\text{TPC}}, \mathcal{F}_{\mathsf{VLA}})$-hybrid model under static corruptions, assuming the hardness of RLWE and the existence of F-limited leveled FHE for any high-degree polynomial F.*

We sketch π_{CVPIR} UC-realizes $\mathcal{F}_{\text{PIR}}$ in the $\mathcal{F}_{\text{VLA}}$-hybrid world by describing a simulator $\mathcal{S}$ and arguing indistinguishability. The simulator $\mathcal{S}$ internally simulates the authenticated router and the $\mathcal{F}_{\text{VLA}}$ functionality, and interacts externally with $\mathcal{F}_{\text{PIR}}$. Its goal is to ensure that all outputs observed by the environment are consistent with a single database and digest, exactly as enforced by $\mathcal{F}_{\text{PIR}}$.

If the server is honest and the client is corrupted, $\mathcal{F}_{\text{PIR}}$ leaks the database to $\mathcal{S}$ when the digest is generated. The simulator then runs the honest-server algorithms of π_{CVPIR} and answers all subsequent verification and lookup interactions using this database. Since $\mathcal{F}_{\text{VLA}}$ replies are computed as specified (e.g., vector-matrix products), the adversary's view is distributed exactly as in the real execution.

If the server is corrupted and the client is honest, $\mathcal{S}$ runs the honest-client protocol against the adversarial server. If registration aborts, both the real and ideal executions abort. Otherwise, successful verification implies that the server committed to a well-defined matrix and tags. Using the extractability guaranteed by the $\mathcal{F}_{\text{VLA}}$ interface, $\mathcal{S}$ recovers the committed matrix and reconstructs a unique database consistent with the digest. This database is recorded by $\mathcal{F}_{\text{PIR}}$ and governs all future lookups.

For lookup, $\mathcal{F}_{\text{PIR}}$ outputs the selected database entry if and only if the recorded digest–database pair is valid. The simulator may use an arbitrary selection index internally, since the verification transcript and abort behavior are independent of the client's index. Thus, the adversary's view during lookup is indistinguishable from the real execution.

7.3 Proof of Theorem 2

We claim the simulation Figs. 14 to 16 is indistinguishable.

Case 1: honest server, corrupt client. In this case, $\mathcal{F}_{\text{PIR}}$ leaks the database to $\mathcal{S}$ via $(\textsc{GenDigest}, \mathsf{sid}, \mathsf{db})$. From that point onward, every message that $\mathcal{A}$ sees is distributed exactly as in the real protocol: (i) tags are answered using the same tags that an honest server would have produced (since $\mathcal{S}$ runs π_{CVPIR}'s digest generation), and (ii) any $\mathcal{F}_{\text{VLA}}$ vector-matrix product reply is computed as $\mathbf{s} = \mathbf{u}^\top \mathbf{D}$ for $\mathbf{D} = \mathsf{DeriveMatrix}(\mathsf{db})$, which is exactly $\mathcal{F}_{\text{VLA}}$'s semantics. Likewise, the simulated honest server's responses to MVPChallenge are generated by running the honest-server algorithm of π_{CVPIR}, so the ciphertext distribution matches the real world. Therefore, $\mathcal{Z}$'s entire view is *perfectly* simulated.

Case 2: corrupt server, honest client. Here, $\mathcal{S}$ runs the honest client code of π_{CVPIR} against the corrupted server. If the client-side protocol outputs $\perp$, then both real and ideal executions abort and $\mathcal{Z}$ receives $\perp$ (matching by construction). If the client-side protocol succeeds, then the corrupted server must have caused a well-formed state in the $\mathcal{F}_{\text{VLA}}$-hybrid (otherwise verification fails and both worlds return $\perp$). $\mathcal{S}$ extracts $(\mathbf{D}, \mathsf{tags})$ from the simulated $\mathcal{F}_{\text{VLA}}$ state and maps columns back to db using pack, setting the output to `corrupt` exactly when a column is outside $\{0, \ldots, p-1\}^m$. This ensures the database handed to $\mathcal{F}_{\text{PIR}}$ is consistent with what the corrupted server effectively committed to (and with the $\mathcal{F}_{\text{VLA}}$ semantics). All subsequent interactions with the corrupted server during lookup are simulated by running the honest client algorithm with an arbitrary selection index, which does not affect the server-visible transcript in π_{CVPIR} (the challenge structure depends only on verification material, not the client index). Thus, the corrupted server's view is distributed as in the real execution, and $\mathcal{Z}$ observes the same accept/reject behavior as in the ideal world. It remains that the only undetectable cheating a corrupted server can perform during the Lookup phase is by answering correctly to all the verification queries

We construct a simulator $\mathcal{S}$ for π_{CVPIR} that interacts with the environment $\mathcal{Z}$ and adversary $\mathcal{A}$ in the real world, and with the ideal functionality $\mathcal{F}_{\text{PIR}}$ in the ideal world, while internally simulating the $\mathcal{F}_{\text{VLA}}$-hybrid and the authenticated router. The simulator is parameterized by $(p, R, L, L_{\mathsf{tag}})$ and assumes $\{0, \ldots, p-1\} \subseteq R$ so that values derived in $[0, p)$ can be interpreted as elements of R consistently, and $\mathcal{F}_{\text{VLA}}$ is also parameterized consistently.

Simulating honest-party leakage (backdoors from $\mathcal{F}_{\text{PIR}}$). $\mathcal{S}$ answers the following backdoor queries from $\mathcal{F}_{\text{PIR}}$ by running the specified hybrid-world computations and/or consulting its records.

- **Backdoor** query $(\textsc{GenDigest}, \mathsf{sid}, \mathsf{db})$ from $\mathcal{F}_{\text{PIR}}$:
  1. Execute exactly the same steps as π_{CVPIR} would upon input $(\textsc{GetDigest}, \mathsf{sid}, \mathsf{db})$ in the $\mathcal{F}_{\text{VLA}}$-hybrid (with $\mathcal{F}_{\text{VLA}}$ simulated internally), obtaining (μ, tags).
  2. Record $(\textsc{validDigest}, \mathsf{sid}, \mu, \mathsf{db}, \mathsf{tags})$.
  3. Return (μ, tags) to $\mathcal{F}_{\text{PIR}}$.
- **Backdoor** query $(\textsc{CheckDigest}, \mathsf{qid}, \mu, \mathsf{db}, \mathsf{tags})$ from $\mathcal{F}_{\text{PIR}}$, where $\mathsf{qid} = (\mathsf{sid}, \mathsf{server}, \mathsf{client}, \mathsf{nonce})$:
  1. If server is corrupt, look for $(\textsc{corruptedResult}, \mathsf{qid}, \mathit{result})$ and return (result) to $\mathcal{F}_{\text{PIR}}$ and Abort
  2. Record $(\textsc{checkPerformed}, \mathsf{qid})$.
  3. Define $(\mathsf{qid}_{\mathsf{VLA},1}, \mathsf{qid}_{\mathsf{VLA},2})$ from qid as in π_{CVPIR}.
  4. Send simulated router message
  $$(\textsc{HonestMessage}, \mathsf{server}, \mathcal{F}_{\text{VLA}}, \textsc{MatrixTagsResult}, \mathsf{qid}_{\mathsf{VLA},1})$$
  to $\mathcal{A}$ and wait for $(\textsc{Deliver})$.
  5. Compute $\mathbf{D} \leftarrow \mathsf{DeriveMatrix}(\mathsf{db})$.
  6. Send backdoor $(\textsc{CheckMC}, \mathsf{qid}_{\mathsf{VLA},1}, \mu, \mathbf{D}, \mathsf{tags})$ to $\mathcal{A}$ and wait for response.
  7. If the response is $\perp$, or $\mathsf{db}/\mathbf{D}/\mathsf{tags}$ are invalid, or $\mathsf{tags} = \texttt{corrupt}$, or some other tuple $(\textsc{validDigest}, \mathsf{sid}, \mu, \mathsf{db}', \mathsf{tags}')$ is already recorded, then return $\perp$ to $\mathcal{F}_{\text{PIR}}$.
  8. Else record $(\textsc{validDigest}, \mathsf{sid}, \mu, \mathsf{db}, \mathsf{tags})$ and return $\top$ to $\mathcal{F}_{\text{PIR}}$.

Simulating router deliveries for honest parties. When the router delivers honest messages involving $\mathcal{F}_{\text{PIR}}$, $\mathcal{S}$ forwards the corresponding $\mathcal{F}_{\text{VLA}}$-side transcript to $\mathcal{A}$ exactly as in the real execution.

- **HonestMessage** $(\mathsf{client}, \mathcal{F}_{\text{PIR}}, \textsc{Register}, \mathsf{qid})$ from Router:
  1. Record $(\textsc{queryType}, \mathsf{qid}, \texttt{Register})$.
  2. If server is corrupt, return $(\textsc{Deliver})$ to Router and abort.
  3. Define $(\mathsf{qid}_{\mathsf{VLA},1}, \mathsf{qid}_{\mathsf{VLA},2})$ from qid as in π_{CVPIR}.
  4. Using the simulated router, sequentially send the following honest-party messages to $\mathcal{A}$, waiting for delivery between them:
     (a) **HonestMessage** $(\mathsf{client}, \mathcal{F}_{\text{VLA}}, \textsc{GetTags}, \mathsf{qid}_{\mathsf{VLA},1})$,
     (b) **HonestMessage** $(\mathcal{F}_{\text{VLA}}, \mathsf{server}, \textsc{GetMatrixTags}, \mathsf{qid}_{\mathsf{VLA},1})$.
  5. Return $(\textsc{Deliver})$ to Router.
- **HonestMessage** $(\mathsf{client}, \mathcal{F}_{\text{PIR}}, \textsc{Lookup}, \mathsf{qid})$ from Router:
  1. Record $(\textsc{queryType}, \mathsf{qid}, \texttt{Lookup})$.
  2. If server is corrupt, return $(\textsc{Deliver})$ to Router and abort.
  3. If no $(\textsc{clientKeyCount}, \mathsf{sid}, \mathsf{server}, \mathsf{client}, \cdot)$ exists, call $\mathsf{SimulateSendKey}(\mathsf{sid}, \mathsf{server}, \mathsf{client})$ defined below.
  4. Send **HonestMessage** $(\mathsf{client}, \mathsf{server}, \text{MVP}\textsc{Challenge}, \mathsf{qid})$ to $\mathcal{A}$ and wait for its delivery.
  5. Return $(\textsc{Deliver})$ to Router.
- **HonestMessage** $(\mathcal{F}_{\text{PIR}}, \mathsf{client}, \textsc{RegisterResult}, \mathsf{qid})$ from Router:
  1. If server is corrupt, return $(\textsc{Deliver})$ to Router and abort.
  2. Define $(\mathsf{qid}_{\mathsf{VLA},1}, \mathsf{qid}_{\mathsf{VLA},2})$ from qid as in π_{CVPIR}.
  3. If $(\textsc{checkPerformed}, \mathsf{qid})$ is not recorded, then send
  $$\textbf{HonestMessage}\ (\mathsf{server}, \mathcal{F}_{\text{VLA}}, \textsc{MatrixTagsResult}, \mathsf{qid}_{\mathsf{VLA},1})$$
  to $\mathcal{A}$ and wait for $(\textsc{Deliver})$.
  4. Send the following simulated router messages to $\mathcal{A}$ sequentially, waiting for delivery between messages:
     (a) **HonestMessage** $(\mathcal{F}_{\text{VLA}}, \mathsf{client}, \textsc{TagsResult}, \mathsf{qid}_{\mathsf{VLA},1})$,
     (b) **HonestMessage** $(\mathsf{client}, \mathcal{F}_{\text{VLA}}, \textsc{VectMatProd}, \mathsf{qid}_{\mathsf{VLA},2})$,
     (c) **HonestMessage** $(\mathcal{F}_{\text{VLA}}, \mathsf{server}, \textsc{GetMatrixTags}, \mathsf{qid}_{\mathsf{VLA},2})$,
     (d) **HonestMessage** $(\mathsf{server}, \mathcal{F}_{\text{VLA}}, \textsc{MatrixTagsResult}, \mathsf{qid}_{\mathsf{VLA},2})$,
     (e) **HonestMessage** $(\mathcal{F}_{\text{VLA}}, \mathsf{client}, \text{VMP}\textsc{Result}, \mathsf{qid}_{\mathsf{VLA},2})$.
  5. Call $\mathsf{SimulateSendKey}(\mathsf{sid}, \mathsf{server}, \mathsf{client})$.
  6. Return $(\textsc{Deliver})$ to Router.

Figure 14: Simulator $\mathcal{S}_{\text{CVPIR}}$ part 1

- **HonestMessage** ($\mathcal{F}_{\text{PIR}}$, client, LookupResult, qid) from Router:
  1. If server is corrupt, return (Deliver) to Router and abort.
  2. Set $\mathsf{qid}_{\mathsf{VLA}} \leftarrow (\mathsf{sid}_{\mathsf{VLA}}, \mathsf{server}, \mathsf{server}, \mathsf{nonce})$.
  3. Send the following simulated router messages sequentially, waiting for delivery between messages:
     (a) **HonestMessage** (server, client, MVPResult, qid),
     (b) **HonestMessage** (server, $\mathcal{F}_{\text{VLA}}$, VerifyAll, $\mathsf{qid}_{\mathsf{VLA}}$),
     (c) **HonestMessage** ($\mathcal{F}_{\text{VLA}}$, server, VerifyAllResult, $\mathsf{qid}_{\mathsf{VLA}}$).
  4. Lookup (clientKeyCount, sid, server, client, c).
  5. If $c + 1 = k$, call SimulateSendKey(sid, server, client);
     else update to (clientKeyCount, sid, server, client, $c + 1$).
  6. Return (Deliver) to Router.
- **Subroutine** SimulateSendKey(sid, server, client):
  1. Send simulated router message **HonestMessage** (client, server, KeyUpdate, sid) to $\mathcal{A}$ and wait for its delivery.
  2. Do the same for the honest server's reply to the client.
  3. Save (clientKeyCount, sid, server, client, 0).

Handling corrupted server. When server is corrupt, $\mathcal{S}$ must provide $\mathcal{F}_{\text{PIR}}$ with a database consistent with whatever the corrupted server committed to in the $\mathcal{F}_{\text{VLA}}$-hybrid during registration. This is achieved by extracting from the simulated $\mathcal{F}_{\text{VLA}}$ state (i.e., the committed matrix and tags), and then mapping matrix columns back into bitstrings using the assumption $\{0, \ldots, p-1\} \subseteq R$.

- **LeakedMessage** ($\mathcal{F}_{\text{PIR}}$, server, GetDB, qid, μ), where $\mathsf{qid} = (\mathsf{sid}, \mathsf{server}, \mathsf{client}, \mathsf{nonce})$:
  1. If (queryType, qid, `Register`) is recorded:
     (a) Execute the client-side code of π_{CVPIR} on input (Register, qid, μ) against the corrupted server, in the simulated $\mathcal{F}_{\text{VLA}}$-hybrid, obtaining (RegisterResult, qid, res) from the code.
     (b) If $\mathsf{res} = \perp$, record (corruptedResult, qid, $\perp$), send
     $$\textbf{CorruptMessage}\ (\mathsf{server}, \mathcal{F}_{\text{PIR}}, \text{DBResponse}, \mathsf{qid}, \perp, \perp)$$
     to Router, and abort.
     (c) Lookup (mcomm, $\mathsf{sid}_{\mathsf{VLA}}, \mu, D, \mathsf{tags}$) in the simulated $\mathcal{F}_{\text{VLA}}$ state.
     (d) For each $i \in [n]$:
        i. If $D[*, i] \in \{0, \ldots, p-1\}^m$, set $\mathsf{db}[i] \leftarrow \mathsf{pack}(D[*, i])$, where pack packs objects into bits.
        ii. Else set $\mathsf{db}[i] \leftarrow 0^L$ and $\mathsf{tags} \leftarrow$ `corrupt`.
     (e) Record (corruptedResult, qid, $\top$) and (validDigest, sid, μ, db, tags).
     (f) Send **CorruptMessage** (server, $\mathcal{F}_{\text{PIR}}$, DBResponse, qid, db, tags) to Router.
  2. Else if (queryType, qid, `Lookup`) is recorded:
     (a) Execute the client-side code of π_{CVPIR} on input (Lookup, qid, μ, 0) against the corrupted server, obtaining (LookupResult, qid, entry) from the code.
     (b) If $\mathsf{entry} = \perp$, send **CorruptMessage** (server, $\mathcal{F}_{\text{PIR}}$, DBResponse, qid, $\perp$, $\perp$) to Router.
     (c) Else lookup (validDigest, sid, μ, db, tags) and send **CorruptMessage** (server, $\mathcal{F}_{\text{PIR}}$, DBResponse, qid, db, tags) to Router.

Figure 15: Simulator $\mathcal{S}_{\text{CVPIR}}$ part 2

Handling corrupted client. When client is corrupt and server is honest, $\mathcal{F}_{\text{PIR}}$ will leak the database and tags to $\mathcal{S}$ at digest-generation time; hence $\mathcal{S}$ can answer all corrupted-client queries consistently by consulting its (validDigest, $\cdot$) record.

- **CorruptMessage** (client, $\mathcal{F}_{\text{VLA}}$, GetTags, $\mathsf{qid}_{\mathsf{VLA},1}$, μ) from $\mathcal{A}$:
  1. If server is corrupt, run $\mathcal{F}_{\text{VLA}}$ honestly with corrupted parties and then abort.
  2. Send **HonestMessage** ($\mathcal{F}_{\text{VLA}}$, server, GetMatrixTags, $\mathsf{qid}_{\mathsf{VLA},1}$) to $\mathcal{A}$ and wait for (Deliver).
  3. Let qid = (sid, server, client, nonce) be the corresponding $\mathcal{F}_{\text{PIR}}$ query id.
  4. Send **CorruptMessage** (client, $\mathcal{F}_{\text{PIR}}$, Register, qid, μ) to Router and wait for **LeakedMessage** ($\mathcal{F}_{\text{PIR}}$, client, RegisterResult, qid, result).
  5. If (checkPerformed, qid) is not recorded, send

     HonestMessage (server, $\mathcal{F}_{\text{VLA}}$, MatrixTagsResult, $\mathsf{qid}_{\mathsf{VLA},1}$)

     to $\mathcal{A}$ and wait.
  6. Record (getTagsResult, qid, result).
  7. If result $= \perp$, send **LeakedMessage** ($\mathcal{F}_{\text{VLA}}$, client, TagsResult, $\mathsf{qid}_{\mathsf{VLA},1}$, $\perp$).
  8. Else lookup (validDigest, sid, μ, db, tags) and send

     LeakedMessage ($\mathcal{F}_{\text{VLA}}$, client, TagsResult, $\mathsf{qid}_{\mathsf{VLA},1}$, tags).

- **CorruptMessage** (client, $\mathcal{F}_{\text{VLA}}$, VectMatProd, $\mathsf{qid}_{\mathsf{VLA},2}$, μ, $\mathbf{u}$) from $\mathcal{A}$:
  1. If server is corrupt, run $\mathcal{F}_{\text{VLA}}$ honestly with corrupted parties and then abort.
  2. Send **HonestMessage** ($\mathcal{F}_{\text{VLA}}$, server, GetMatrixTags, $\mathsf{qid}_{\mathsf{VLA},2}$) to $\mathcal{A}$ and wait.
  3. Let qid = (sid, server, client, nonce) be the corresponding $\mathcal{F}_{\text{PIR}}$ query id.
  4. Lookup (getTagsResult, qid, result); abort if not found.
  5. Send **HonestMessage** (server, $\mathcal{F}_{\text{VLA}}$, MatrixTagsResult, $\mathsf{qid}_{\mathsf{VLA},2}$) to $\mathcal{A}$ and wait.
  6. If result $= \perp$, send **LeakedMessage** ($\mathcal{F}_{\text{VLA}}$, client, VMPResult, $\mathsf{qid}_{\mathsf{VLA},2}$, $\perp$) and abort.
  7. Lookup (validDigest, sid, μ, db, tags).
  8. Compute $D \leftarrow \mathsf{DeriveMatrix}(\mathsf{db})$ and $\mathbf{s}^\top \leftarrow \mathbf{u}^\top \mathbf{D}$.
  9. Send **LeakedMessage** ($\mathcal{F}_{\text{VLA}}$, client, VMPResult, $\mathsf{qid}_{\mathsf{VLA},2}$, $\mathbf{s}$).

- **CorruptMessage** (client, server, MVPChallenge, qid, μ, $\{\widehat{q_i}\}_{i\in[\lambda]}$) from $\mathcal{A}$:
  1. Send **CorruptMessage** (client, $\mathcal{F}_{\text{PIR}}$, Lookup, qid, μ, 0) to Router and wait for **LeakedMessage** ($\mathcal{F}_{\text{PIR}}$, client, LookupResult, qid, entry).
  2. Let $\mathsf{qid}_{\mathsf{VLA}}$ = ($\mathsf{sid}_{\mathsf{VLA}}$, server, server, nonce) and send the following sequentially, waiting for delivery between messages:
     (a) **HonestMessage** (server, $\mathcal{F}_{\text{VLA}}$, VerifyAll, $\mathsf{qid}_{\mathsf{VLA}}$),
     (b) **HonestMessage** ($\mathcal{F}_{\text{VLA}}$, server, VerifyAllResult, $\mathsf{qid}_{\mathsf{VLA}}$).
  3. Lookup (validDigest, sid, μ, db, tags) and (clientKey, sid, server, client, μ, pk).
  4. If entry $= \perp$ or either lookup fails, send

     LeakedMessage (server, client, MVPResult, qid, $\perp$)

     and abort.
  5. Otherwise, perform π_{CVPIR}'s honest-server response to MVPChallenge to obtain ciphertexts $(\widehat{r_{i,j}})$.
  6. Send **LeakedMessage** (server, client, MVPResult, qid, $(\widehat{r_{i,j}})$).

- **CorruptMessage** (client, server, KeyUpdate, sid, μ, pk) from $\mathcal{A}$:
  1. Store (clientKey, sid, server, client, μ, pk).
  2. Return (KeyReceived) to $\mathcal{A}$ via a simulated router **LeakedMessage**.

Figure 16: Simulator $\mathcal{S}_{\text{CVPIR}}$ part 3

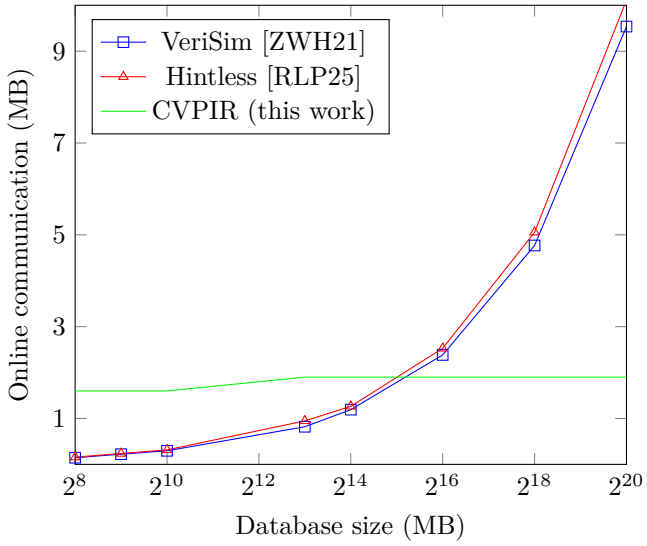


Figure 17: Online communication comparison with verifiable PIRs

and incorrectly to all the actual queries. To do so, for each query, the server evaluates a homomorphic circuit with bounded depth d, or more explicitly evaluates a polynomial of bounded degree d in the verifier's chosen values. The Schwartz-Zippel argument affirms that if any polynomial other than the correct one is chosen, then the probability to obtain the correct evaluation is bounded by $\frac{d}{p}$, with $p > 2^\lambda$ the plaintext field size. Using the F-limited assumption for FHE, we bound the maximum degree d of any polynomial a corrupted server can use to be negligible compared to the field size. It means that with overwhelmed probability, a corrupted server performing a incorrect computation on one verification query will be detected. Then, the server must guess which of the λ queries are the verification queries, and this probability is bounded by $\frac{1}{2^\lambda}$.

8 Query Protocol Implementation with OpenFHE

Experimental setup. Our code is written in C++, using the OpenFHE library that implements the BFV scheme. We ran our benchmarks on an Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz with 56 threads, using parallelism to facilitate the execution of computationally intensive homomorphic operations. The timings given in the following are the median of 3 to 5 runs of the code per input. Our experiments demonstrate that the communication volume of our protocol's online phase scales well with the size of the database as shown in Fig. 17.

BFV parameters. The FHE context is chosen independently of the inputs. We fix the message space to be the field $\mathbb{F}_p$ with p a prime of 48 bits and set the statistical security level of the FHE scheme to 128 bits. In that context, a ciphertext can contain up to 2^{14} field elements (let S be this number of slots, i.e., the packing parameter).

Figure 18 shows the concrete sizes (in MB) for a maximal multiplicative depth of 3 and 4[2]: a fresh ciphertext (which is also the size of the public key), a reduced ciphertext (on which no more homomorphic operation can be performed on), the private key, and the evaluation keys for homomorphic multiplications and rotations[3].

[2]This in enough to evaluate a more than 1TB database with our protocol.

[3]We decided to generate all power-of-2 rotation keys to reduce the computational cost. One could use only keys for rotations by 1 and $\sqrt{S}$, using Horner's style.

Max. depth	fresh ctxt	reduced ctxt	priv. key	mult. key	rot. key
3	1.3	0.3	0.7	6.6	164
4	1.6	0.3	0.8	9.4	236

Figure 18: Sizes (in MB) of the objects depending on the context multiplicative depth

8.1 Protocol Benchmark Implementation

The probabilistic security parameter λ of our implementation is set to 42. For reproducibility, our implementations can be tested using `https://anonymous.4open.science/r/vPIR-openFHE-78D7`.

Benchmark input. The benchmarks take the following input.

- The number l of cuts in the query vector, which also determines the maximum multiplicative depth of the algorithm: $\lceil \log(l+1) \rceil + 1$.
- N_q: the number of query ciphertexts.
- N_a: the number of response ciphertexts.

Maximum database size and communication volume. The choice of those parameters allows the user to have more control on the communication volume as it is then N_q fresh and N_a reduced ciphertexts. From a fixed communication implied by the choice of (N_q, N_a, l), the size of the largest matrix representing the database is computed as follows. We have to pack λl vectors into $N_q S$ slots, so each small vector is of size $B = \frac{N_q S}{\lambda l}$. This implies that the database have up to $n = B^l$ columns. For the answer, we have to pack λ columns plus their tag (which requires 11 slots) into N_a ciphertexts, so the database have up to $m = \frac{N_a S}{\lambda} - 11$ rows. With the choice of the plaintext space, the largest database size if $6mn$ bytes.

Query processing. We are using the SIMD structure of the BFV ciphertexts to perform in parallel the same operations for the λ queries. To do so, the $\frac{S}{\lambda}$ first slots of each ciphertext are only used for the first query, and so on. For clarity, we only describe the algorithm on a single query.

- *Unpacking.* Let $V_1, \cdots, V_l$ be the packed encrypted vectors for a single query ; each of those have B entries. Each coefficient of the first $l-1$ vectors are extracted in separate ciphertexts, using $\ltimes_F$ using an appropriate mask. For each ciphertext created, the coefficient in replicated across $\frac{S}{\lambda}$ slots using rotations and addition. It results in $(l-1)B$ ciphertexts to store. Namely, for all $i \in [1, l-1]$ and $j \in [B]$:

$$\overline{V_i[j]} = \mathsf{FHEnc}(\underbrace{V_i[j], \ldots, V_i[j]}_{\mathsf{S}/\lambda \text{ times}}).$$

  To prepare the *evaluate* algorithm, we extract and store all the rotations on $\frac{S}{\lambda}$ slots of the last vector V_l. Namely, for $k \in [B]$:

$$\sigma_k(V_l) := \mathsf{FHEnc}\big(\mathsf{V_l}[\mathsf{k} \bmod \mathsf{B}], \mathsf{V_l}[(\mathsf{k}+1) \bmod \mathsf{B}], \cdots, \mathsf{V_l}[(\mathsf{k} + \frac{\mathsf{S}}{\lambda} - 1) \bmod \mathsf{B}]\big).$$

  Similarly to the others, the extraction of V_l is done using one $\ltimes_F$ with appropriate masks, some rotations and additions. Altogether, it requires the server to stores lB ciphertexts.

- *Evaluate.* The protocol is built such that the tensor product $V_1 \otimes \cdots \otimes V_l$ is computed before doing the matrix-vector is the database. However, in practice computing

the tensor product homomorphically imply the storage capacity of at least $\frac{\lambda B^l}{S}$ ciphertexts, which is larger than the plaintext database itself. For that reason, we instead perform an equivalent operation making good usage of the slots and never exceeding the lB ciphertext stored from *unpacking*. The database matrix is divided horizontally into N_a sub-matrices of $\frac{S}{\lambda}$ rows. Each sub-matrix evaluation results to one of the N_a answered ciphertexts. Let M be one such sub-matrix, having $n = B^l$ columns. M is divided vertically into B^{l-1} blocks of B columns. Namely, each bloc is denoted $M_{(a_1,\cdots,a_{l-1})}$ for $a_i \in [B]$. As the operator $\ltimes_F$ acts slot-wise, the diagonals of each blocks are packed into the slots of a plaintext to perform the homomorphic matrix-vector product. We denote $D_k(M_{(a_1,\cdots,a_{l-1})})$ the k^{th} diagonal of $M_{(a_1,\cdots,a_{l-1})}$ packed into $\frac{S}{\lambda}$ slots. Finally, the algorithm performs the following operation, combining the tensor product with the matrix-vector product.

$$\sum_{a_i \in [B]} \overline{V_1[a_1]} \times_F \cdots \times_F \overline{V_{l-1}[a_{l-1}]} \times_F \sum_{k \in [B]} D_k(M_{(a_1,\cdots,a_{l-1})}) \ltimes_F \sigma_k(V_l) \qquad (5)$$

This computation is using properly the SIMD structure of the ciphertext, does not require extra storage and is widely parallelizable.

Cost analysis. We ignore the cost of the *unpack* algorithm as it is negligible in practice compared to *evaluate*, however we consider that it requires one layer of depth[4]. The *evaluate* algorithm multiplies together l ciphertexts and 1 plaintext, so has a multiplicative depth $\lceil \log(l+1) \rceil + 1$ through a multiplicative binary tree. The algorithm performs $N_a n = N_a(\frac{S}{\lambda l} N_q)^l$ products $\ltimes_F$ and $N_a n - 1$ additions $+_F$. Note that in (5) we could factorize the products by $\overline{V_i[a_i]}$ to reduce the number of products $\times_F$. However, if we fully factorize, the multiplicative depth is then $l+1$. We instead can partially factorize, such that the multiplicative depth remains minimal. This results into a number of product $\times_F$ between $N_a(\frac{S}{\lambda l} N_q)^{l-1}(l-1)$ and $N_a(\sum_{i=1}^{l-1}(\frac{S}{\lambda l} N_q)^i)$.

8.2 Experiment Results

Runtime results. Figure 19 reports the experimental results for various input parameters[5]. Our main observations are as follows:

- Each entry of the database has to be touched so the number of products $\ltimes_F$ scales linearly with the maximum database size and dominates the runtime. In seconds per gigabyte, we observe relatively small variations for various database sizes, ranging from 15 to 18 seconds/GB.
- The number of product $\times_F$ increases with the number of query vectors l. For instance, with $(l, N_q, N_a) = (3, 1, 1)$, we observe $\times_F$ operations account for 12 seconds/GB, whereas with $(5, 1, 1)$ they account for 24 seconds/GB. However, this quantity also decreases as the number of query ciphertexts N_q increases: in configuration $(3, 5, 2)$ it drops to 2.5 seconds/GB.
- A communication/computation tradeoff can be performed. As an example, various parameters (l, N_q, N_a) allow to evaluate a 1TB database: (5,1,1), (4,1,5), (3,3,8), (3,4,3), (3,5,2), (2,12,78). The lowest communication volume is 1.9 MB using parameters $(5, 1, 1)$, but it comes at the cost of additional products $\times_F$ and the need for a context with 4 layers of multiplicative depth. On the other hand, one could use

[4] A plaintext-ciphertext product can be considered as adding 0.5 layer of depth.
[5] The timings for $(5, 1, 1)$ are estimations.

parameters $(2, 12, 78)$ to minimize the runtime, but then the communication volume skyrockets to 39 MB.

Comparison with the state of the art. The Fig. 17 compares the online communication volume of our protocol with [ZWH21, RLP25]. In their schemes, the communication volume scales as $O(\sqrt{|DB|})$; we (optimistically) estimate their communication volume for databases wider than 16 GB following this asymptotic. For that reason, our scheme is far more scalable than previous works for huge databases as our communication volume is mainly governed by the parameters (l, N_q, N_a) rather than directly by the database size. However, even if our computation asymptotic are not worse than the state of the art's, growing linearly with the database size, some structural aspects of our construction make our runtime efficiency not directly comparable with other papers. Indeed, our solution to have a communication volume growing logarithmically in the database size requires a FHE context with several layers of multiplicative depth, so some computational overheads. Also, our verifiability property requires to perform λ times the database evaluation.

comm. (MB)	inputs (l, N_q, N_a)	maximum $\|DB\|$ (GB)	effi. (s/GB)	$\ltimes_F / \times_F / +_F$ (%)
1.6	$(3, 1, 1)$	5	30.4	46/37/17
1.9	$(4, 1, 1)$	201	43.9	39/42/19
	$(5, 1, 1)$	6565	50.5*	36/47/17
2.8	$(3, 1, 5)$	26	25.4	46/37/17
3.1	$(4, 1, 5)$	1030	39.4	39/42/19
3.2	$(3, 2, 2)$	81	23.6	57/22/21
3.8	$(4, 2, 2)$	6536	33.8	50/26/24
6.1	$(3, 4, 3)$	978	23.7	64/13/24
6.3	$(2, 3, 8)$	6	24	69/6/25
	$(3, 3, 8)$	1107	23.1	61/16/23
7.1	$(3, 5, 2)$	1267	22.4	65/10/24

Figure 19: Experimental results under openFHE

References

[AB25] Bar Alon and Amos Beimel. On the definition of malicious private information retrieval. In Niv Gilboa, editor, *ITC 2025*, volume 343 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl, August 2025. doi:10.4230/LIPIcs.ITC.2025.8.

[ABC+07] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Song. Provable data possession at untrusted stores. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 598–609. ACM Press, October 2007. doi:10.1145/1315245.1315318.

[ADdJ+21] Gaspard Anthoine, Jean-Guillaume Dumas, Mélanie de Jonghe, Aude Maignan, Clément Pernet, Michael Hanling, and Daniel S. Roche. Dynamic proofs of retrievability with low server storage. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 537–554. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/anthoine.

[AIVG22] Kinan Dak Albab, Rawane Issa, Mayank Varia, and Kalman Graffi. Batched differentially private information retrieval. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 3327–3344. USENIX Association, August 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/albab.

[ALP+21] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1811–1828. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/ali.

[BBG05] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical identity based encryption with constant size ciphertext. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 440–456. Springer, Berlin, Heidelberg, May 2005. doi:10.1007/11426639_26.

[BDKP22] Shany Ben-David, Yael Tauman Kalai, and Omer Paneth. Verifiable private information retrieval. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part III*, volume 13749 of *LNCS*, pages 3–32. Springer, Cham, November 2022. doi:10.1007/978-3-031-22368-6_1.

[BFKT24] Jan Bobolz, Pooya Farshim, Markulf Kohlweiss, and Akira Takahashi. The brave new world of global generic groups and UC-secure zero-overhead SNARKs. In Elette Boyle and Mohammad Mahmoody, editors, *TCC 2024, Part I*, volume 15364 of *LNCS*, pages 90–124. Springer, Cham, December 2024. doi:10.1007/978-3-031-78011-0_4.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Berlin, Heidelberg, August 2012. doi:10.1007/978-3-642-32009-5_50.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. doi:10.1109/SFCS.2001.959888.

[Cap13] Justin Cappos. Avoiding theoretical optimality to efficiently and privately retrieve security updates. In Ahmad-Reza Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 386–394. Springer, Berlin, Heidelberg, April 2013. doi:10.1007/978-3-642-39884-1_33.

[CCL15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 3–22. Springer, Berlin, Heidelberg, August 2015. doi:10.1007/978-3-662-48000-7_1.

[CF13] Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 55–72. Springer, Berlin, Heidelberg, February / March 2013. doi:10.1007/978-3-642-36362-7_5.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th FOCS*, pages 41–50. IEEE Computer Society Press, October 1995. doi:10.1109/SFCS.1995.492461.

[CHLR18] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1223–1237. ACM Press, October 2018. doi:10.1145/3243734.3243836.

[CNC+23] Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J. Wu, and Bryan Ford. Authenticated private information retrieval. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3835–3851. USENIX Association, August 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/colombo.

[dCL24] Leo de Castro and Keewoo Lee. VeriSimplePIR: Verifiability in SimplePIR at no online cost for honest servers. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/de-castro.

[DT24] Marian Dietz and Stefano Tessaro. Fully malicious authenticated PIR. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part IX*, volume 14928 of *LNCS*, pages 113–147. Springer, Cham, August 2024. doi:10.1007/978-3-031-68400-5_4.

[FV12a] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. URL: https://eprint.iacr.org/2012/144.

[FV12b] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. In *Cryptology ePrint Archive, Report 2012/144*, 2012.

[HHC+23] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3889–3905. USENIX Association, August 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/henzinger.

[HSW23] Laura Hetz, Thomas Schneider, and Christian Weinert. Scaling mobile private contact discovery to billions of users. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *ESORICS 2023, Part I*, volume 14344 of *LNCS*, pages 455–476. Springer, Cham, September 2023. doi:10.1007/978-3-031-50594-2_23.

[INdPP22] Malika Izabachène, Anca Nitulescu, Paola de Perthuis, and David Pointcheval. MyOPE: Malicious SecuritY for oblivious polynomial evaluation. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22*, volume 13409 of *LNCS*, pages 663–686. Springer, Cham, September 2022. doi:10.1007/978-3-031-14791-3_29.

[JK07] Ari Juels and Burton S. Kaliski, Jr. Pors: proofs of retrievability for large files. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 584–597. ACM Press, October 2007. doi:10.1145/1315245.1315317.

[KO97] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th FOCS*, pages 364–373. IEEE Computer Society Press, October 1997. doi:10.1109/SFCS.1997.646125.

[KRS+19] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1447–1464. USENIX Association, August 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/kales.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Berlin, Heidelberg, December 2010. doi:10.1007/978-3-642-17373-8_11.

[LG15] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 168–186. Springer, Berlin, Heidelberg, January 2015. doi:10.1007/978-3-662-47854-7_10.

[Lin16] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. URL: https://eprint.iacr.org/2016/046.

[RLP25] Mayank Rathee, Keewoo Lee, and Raluca Ada Popa. Verifiable PIR with small client storage. Cryptology ePrint Archive, Paper 2025/1714, 2025. URL: https://eprint.iacr.org/2025/1714.

[Rya14] Mark D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS 2014*. The Internet Society, February 2014. doi:10.14722/ndss.2014.23379.

[SSP13] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 325–336. ACM Press, November 2013. doi:10.1145/2508859.2516669.

[TPY+19] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1556–1571. USENIX Association, August 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/thomas.

[WLZ+23] Yinghao Wang, Xuanming Liu, Jiawen Zhang, Jian Liu, and Xiaohu Yang. Crust: Verifiable and efficient private information retrieval with sublinear online time. Cryptology ePrint Archive, Report 2023/1607, 2023. URL: https://eprint.iacr.org/2023/1607.

[WZ18] Xingfeng Wang and Liang Zhao. Verifiable single-server private information retrieval. In David Naccache, Shouhuai Xu, Sihan Qing, Pierangela Samarati, Gregory Blanc, Rongxing Lu, Zonghua Zhang, and Ahmed Meddahi, editors, *ICICS 18*, volume 11149 of *LNCS*, pages 478–493. Springer, Cham, October 2018. doi:10.1007/978-3-030-01950-1_28.

[ZKN+25] Lin Zhu, Lingwei Kong, Xin Ning, Xiaoyang Qu, and Jianzong Wang. Publicly verifiable private information retrieval protocols based on function secret sharing. *CoRR*, abs/2509.13684, 2025.

[ZPZS24] Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. Piano: Extremely simple, single-server PIR with sublinear server computation. In *2024 IEEE Symposium on Security and Privacy*, pages 4296–4314. IEEE Computer Society Press, May 2024. doi:10.1109/SP54263.2024.00055.

[ZS14] Liang Feng Zhang and Reihaneh Safavi-Naini. Verifiable multi-server private information retrieval. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS 2014*, volume 8479 of *LNCS*, pages 62–79. Springer, Cham, June 2014. doi:10.1007/978-3-319-07536-5_5.

[ZWH21] Liang Zhao, Xingfeng Wang, and Xinyi Huang. Verifiable single-server private information retrieval from LWE with binary errors. *Inf. Sci.*, 546:897–923, 2021. URL: https://doi.org/10.1016/j.ins.2020.08.071, doi:10.1016/J.INS.2020.08.071.

A Overview of Simplified Universal Composability (SUC) Model and Message Conventions

We work in the *SUC model* [CCL15]. All communication between protocol parties and ideal functionalities, as well as between protocol parties themselves, is routed through an authenticated, adversarially controlled *router*. The adversary is notified whenever a message is sent and controls when (if ever) the message is delivered, but cannot modify message contents or observe private message components.

Each message is divided into a *public part* and a *private part*. For our purposes, the public part consists of the sender, recipient, message type, and session identifier (sid); all remaining fields are private and hidden from the adversary unless explicitly leaked.

Message Sending. We use the notation

send message (type, sid, payload) *to* recipient *via router*

to denote that a message with public header (sender, recipient, type, sid) and private payload payload is sent to the router for eventual delivery. Upon accepting the message, the router responds with (OK), at which point the sender's execution resumes and proceeds with the following statements, if any.

Router Behavior. When the router receives a message (type, sid, payload) from sender to recipient, it behaves as follows:

- If the recipient is corrupted, the router sends a backdoor message

  $$(\textsc{LeakedMessage}, \mathsf{sender}, \mathsf{recipient}, \mathsf{type}, \mathsf{sid}, \mathsf{payload})$$

  to the adversary.

- If the recipient is honest, the router sends a backdoor query

  $$(\textsc{HonestMessage}, \mathsf{sender}, \mathsf{recipient}, \mathsf{type}, \mathsf{sid})$$

  to the adversary and waits for a response (Deliver) before delivering the message to the recipient.

The router also accepts backdoor messages of the form

$$(\textsc{CorruptMessage}, \mathsf{sender}, \mathsf{recipient}, \mathsf{type}, \mathsf{sid}, \mathsf{payload})$$

from the adversary when the sender is corrupted. Such messages are immediately delivered to the honest recipient.

For simulation purposes, we conceptually distinguish between two routers: the real router in the ideal world, and a simulated router used to generate the adversary's view. If a simulator receives an (HonestMessage, ...) backdoor query that is not explicitly handled, it is assumed to immediately reply with (Deliver).

Queries and Returns. We use a query/return abstraction to model request–response interactions.

- *Send query* (msg) *to Entity and wait for response* (resp) implicitly generates a fresh message identifier msgid nonce, stores the local execution state, sends the message via the router, and yields execution. When a response carrying the same msgid is later received, the stored state is restored and execution resumes.

- *Return* (resp) *to Entity* sends the response to the original sender via the router, implicitly attaching the same msgid. Unless otherwise stated, the current activation terminates and is not resumed.

This abstraction applies uniformly to routed messages between parties, interactions with ideal functionalities, backdoor communication with the adversary, and input/output with the environment $\mathcal{Z}$.

Abort and Assertions.

- **Abort** terminates the current activation and returns control to the environment.

- **Assert** [stmt] checks that stmt holds; otherwise, the activation aborts.

Session Identifiers. If a session identifier sid is structured (e.g., a tuple), it is implicitly unpacked and validated upon receipt of every message. If the structure or contents of sid are invalid, the protocol aborts.

B The KZG Polynomial Commitment Scheme

Here, we describe a polynomial commitment scheme in [KZG10]. Let $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ be cyclic groups of prime order p with generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$ and a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$.

- $\mathsf{KZG.KeyGen}(1^k, d) \to \mathsf{pp}$: Sample $s \xleftarrow{\$} \mathbb{F}_p$ and set

$$\mathsf{pp} \;=\; \Big(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e,\; g_1, g_2,\; \{g_1^{s^i}\}_{i=1}^{d-1},\; g_2^s\Big).$$

  Here, the value s is the trapdoor and is not included in pp.

- $\mathsf{KZG.Commit}_{\mathsf{pp}}(P(x)) \to \phi$:

  Given $P(X) = \sum_{i=0}^{d-1} u_i X^i \in \mathbb{F}_p[X]$, output $\phi \;=\; g_1^{P(s)} \;=\; \prod_{i=0}^{d-1} \left(g_1^{s^i}\right)^{u_i}$.

- $\mathsf{KZG.VerifyPoly}_{\mathsf{pp}}(\phi, P(x)) \to \top/\bot$:

  Given $P(X) = \sum_{i=0}^{d-1} u_i X^i \in \mathbb{F}_p[X]$, accept iff $\phi \;=\; \prod_{i=0}^{d-1} \left(g_1^{s^i}\right)^{u_i}$.

- $\mathsf{PC.CreateWitness}_{\mathsf{pp}}(P(x), a) \to w$:

  Let $Q(X) \;=\; \frac{P(X)-P(a)}{X-a} \in \mathbb{F}_p[X]$ and compute $w \;=\; g_1^{Q(s)}$ using pp.

- $\mathsf{KZG.VerifyEval}_{\mathsf{pp}}(\phi, a, b, w) \to \top/\bot$:

  Accept iff $e(\phi, g_2) \;=\; e(w, g_2^s \cdot g_2^{-a}) \cdot e(g_1, g_2)^b$.

C Additional Figures

Fig. 20 shows the global generic group model functionality that we use in this work. It is reproduced from Bobolz et al. [BFKT24].

Fig. 21 provides a visual depiction of the commitment and tag computation, which was described in text in Section 6.1.

Functionality: $\mathcal{G}$-oGG

$\mathcal{G}$-oGG is (implicitly) parameterized with

- A prime number p
- Sets $S_1, S_2, S_t \subseteq \{0,1\}^*$ with $|S_i| = p$ for $i \in \{1,2,t\}$.

$\mathcal{G}$-oGG maintains the following state:

- $\tau_i : \mathbb{Z}_p \to S_i$ a random encoding function, mapping discrete logs $x \in \mathbb{Z}_p$ to their randomly encoded group elements $h \in S_i$.
- $\mathsf{Var}_{i,sid}$ keeps track of formal variables.
- $R_i[h]$ for $i \in \{1,2,t\}, h \in \mathcal{S}$, keeps track of polynomial representations corresponding to h.
- Ob keeps track of observable actions.

Furthermore, we use the following terms derived from the current state

- We write $\mathsf{Var}_{sid} = \mathsf{Var}_{1,sid} \| \mathsf{Var}_{2,sid} \| \mathsf{Var}_{t,sid}$.
- We write Var to refer to concatenation of all Var_{sid} over all sid.
- We write $\mathsf{Legal}_{sid} = \langle \mathsf{Var}_{sid} \rangle_{\mathbb{Z}_p[\mathsf{Var}_{sid}]} = \sum_{\mathsf{X} \in \mathsf{Var}_{sid}} \mathsf{X} \cdot \mathbb{Z}_p[\mathsf{Var}_{sid}]$, which refers to the set of polynomials that contains only this session's variables $\mathsf{X} \in \mathsf{Var}_{sid}$.

Init()

1. for $i \in \{1,2,t\}$ do:
   (a) Sample τ_i randomly from all possible mappings.
   (b) $R_i[\tau_i(1)] \leftarrow \{1\}$

CanonicalGen$_{\mathsf{sid}}(i)$:

1. return $\tau_i(1)$.

Op$_{\mathsf{sid}}(i, g_1, g_2, a_1, a_2)$:

1. assert $(g_1, g_2, a_1, a_2) \in S_i^2 \times \mathbb{Z}_p^2$.
2. for $j \in \{1,2\}$: Touch$_{sid}(i, g_j)$
3. $h \leftarrow \tau_i(a_1 \tau_i^{-1}(g_1) + a_2 \tau_i^{-1}(g_2))$,
4. Add $(a_1 R_i[g_1] + a_2 R_i[g_2])$ to $R_i[h]$.
5. if $\exists f \in R[h] : f \notin \mathsf{Legal}_{sid}$ then append $(\mathrm{Op}, g_1, g_2, a_1, a_2, h)$ to Ob,
6. return h.

Observe$_{\mathsf{sid}}()$:

1. return Ob.

Touch$_{\mathsf{sid}}(i, g)$:

1. If $R_i[g] = \emptyset$ then
   (a) Initialize fresh variable X
   (b) Add X to $\mathsf{Var}_{i,sid}$
   (c) $R_i[g] \leftarrow \{\mathsf{X}\}$

Pair$_{\mathsf{sid}}(g_1, g_2)$:

1. assert $(g_1, g_2) \in S_1 \times S_2$
2. for $i \in \{1,2\}$: Touch$_{sid}(i, g_i)$.
3. $h \leftarrow \tau_t(\tau_1^{-1}(g_1) \cdot \tau_2^{-1}(g_2))$,
4. Add $(R_1[g_1] \cdot R_2[g_2])$ to $R_t[h]$
5. if $\exists f \in R_t[h] : f \notin \mathsf{Legal}_{sid}$ then append $(\mathrm{Pair}, t, g_1, g_2, h)$ to Ob.
6. return h.

Figure 20: Ideal functionality $\mathcal{G}$-oGG for the global generic group model, reproduced from [BFKT24].

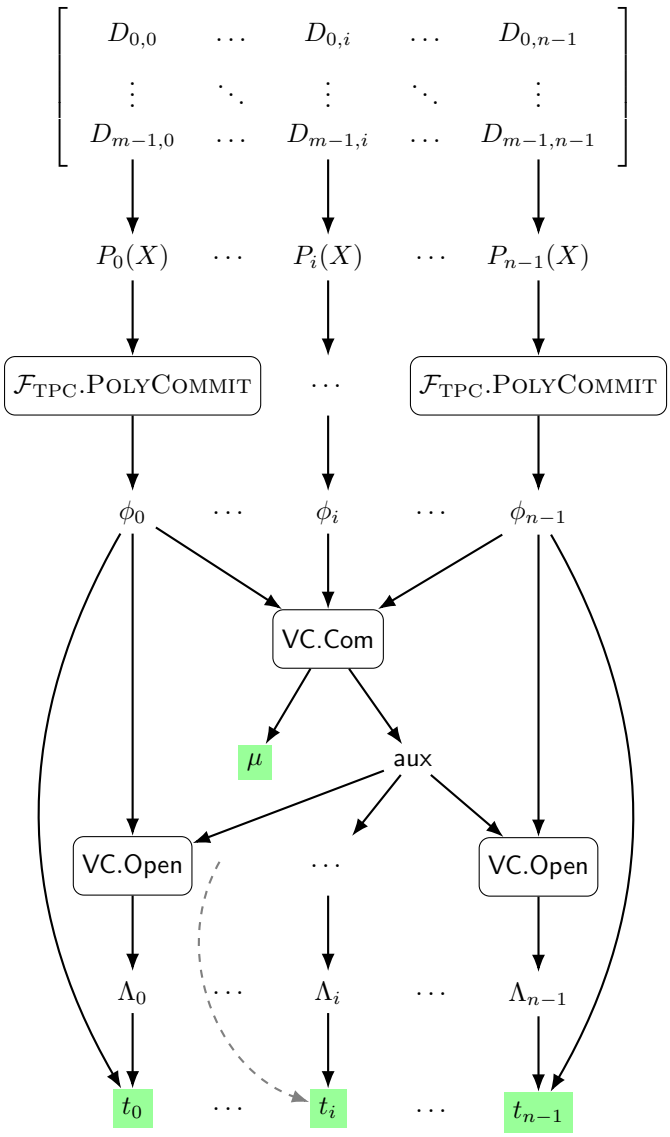


Figure 21: Commitment and tag computation