TAPIR: A Two-Server Authenticated PIR Scheme with Preprocessing

Francesca Falzon
ETH Zurich
Zurich, Switzerland
ffalzon@ethz.ch

Laura Hetz
ETH Zurich
Zurich, Switzerland
lahetz@ethz.ch

Annamira O'Toole
ETH Zurich
Zurich, Switzerland
aotoole@ethz.ch

Abstract

Authenticated Private Information Retrieval (APIR) enables a client to retrieve a record from a public database and verify that the record is "authentic" without revealing any information about which record was requested. In this work, we propose Tapir: the first two-server APIR scheme to achieve both sublinear communication and computation complexity for queries, while also supporting updates. Our scheme builds upon the unauthenticated two-server PIR scheme SinglePass (Lazzaretti and Papamanthou, USENIX'24). Due to its modular design, Tapir provides different trade-offs depending on the underlying vector commitment scheme used.

Moreover, Tapir is the first APIR scheme with preprocessing to support appends and edits in time linear in the database partition size. This makes it an ideal candidate for transparency applications that require support for integrity, database appends, and private lookups. We provide a formal security analysis and a prototype implementation that demonstrates our scheme's efficiency. Tapir incurs as little as 0.11 % online bandwidth overhead for databases of size 2^{22}, compared to the unauthenticated SinglePass. For databases of size $\geq 2^{20}$, our scheme, when instantiated with Merkle trees, outperforms all prior multi-server APIR schemes with respect to online runtime.

Keywords

Private Information Retrieval, Authenticated PIR

1 Introduction

Private Information Retrieval (PIR) enables a client to retrieve a record from a public database without revealing any information about which record was requested to the server(s) holding the database. With growing concerns around digital privacy, PIR has gained a lot of attention in both academia and industry, and has been the focus of extensive research, e.g., [KC21, CHK22, HHC+23, LP23, LP24]. Notably, PIR has many real-world applications such as transparency systems [HHC+23, CNC+23], private web search [HDCZ23, ABG+24], private messaging [CSM+20, AS16, LTW24], and private lookup [KC21, Res24]. PIR has even been deployed at large scale by Apple to enable private caller ID [Res24].

While PIR has been touted as a solution for various use cases, there is a gap between the functionality provided by existing PIR schemes and that required by real-world systems. Practical systems demand more than just query privacy: they require low-latency, sublinear query performance, robust integrity guarantees against adaptive adversaries, and provably secure support for dynamic databases. For example, data structures used in transparency applications must support appends *and* provide integrity (see e.g., [MBB+15, MKKS+23, CDGM19, TBP+19, HHK+21]), and may require additional guarantees such as append-only properties. Furthermore, these systems must operate securely even in the presence of malicious servers, and where PIR is only one part of a larger system. However, existing PIR protocols do not satisfy all of these requirements simultaneously.

A particularly subtle threat in the presence of malicious adversaries are *selective failure attacks*. If a malicious server answers a client's query arbitrarily, the client may detect the error and choose to abort. If the server can observe whether or not the client aborts, the client's behavior may leak information about the queried index, thereby violating PIR's core goal of query privacy. This attack becomes especially problematic when PIR is used as a building block in a larger system—a reasonable assumption for real-world PIR use cases, such as Key Transparency (KT) or secure messaging. Simply verifying the correctness of the returned record may be insufficient to ensure that a PIR scheme achieves both *integrity* and *privacy with abort* [KO97].

To address the problem of selective failure when providing integrity in PIR, Colombo et al. [CNC+23] recently introduced *Authenticated PIR (APIR)*. APIR enables a client to succinctly retrieve information from a public database while guaranteeing both privacy and integrity of the retrieved record. A server's attempt to mount a selective failure attack should result in the client aborting with equal probability, regardless of the record queried. Recent works have proposed APIR schemes for both the single- [CNC+23, DT24, FMS24, dCL24] and multi- [CNC+23, WLZ+23] server settings. Although these schemes ensure both privacy and integrity, they leave open the question of whether such strong guarantees can be achieved alongside the sub-linear query performance and update support required by real-world applications. In particular, prior APIR work has computational server complexity linear in the database size for answering a query. In the single-server setting, this overhead is required to ensure that the server does not learn any information about the requested record. In the multi-server setting, this overhead stems from a lack of database preprocessing, a technique commonly used in state-of-the-art PIR schemes [KC21, LP23, LP24] to reduce query-time costs.

Given the gaps in the literature, we ask the following question:

"*Can we design a multi-server PIR scheme that (1) achieves query computation and communication costs that are sublinear in the database size, (2) provides integrity while mitigating selective failure attacks, and (3) supports updates?*"

1.1 Contributions

In this work, we answer the above question affirmatively by proposing Tapir: the first two-server APIR protocol with preprocessing that achieves sublinear bandwidth and computational query complexity, *and* supports efficient updates, i.e., additions and edits.

New Definitions (§2). The combination of functionalities we aim to support requires a new interface compared to prior work. Prior APIR schemes were either multi-server without preprocessing [AB25, CNC$^+$23], single-server [CNC$^+$23, AR25, DT24], or did not include updates in the client-preprocessing model [FMS25]. To this end, we introduce a new syntax to formalize *updateable APIR with preprocessing*, along with formal definitions of correctness, integrity, and privacy with abort. Our formalization captures both the single-server ($k = 1$) and multi-server ($k \geq 2$) settings. Our new game-based definitions for security capture a powerful adversary who can choose which server to corrupt (for $k \geq 2$), select the database, and adaptively issue lookup and update queries. For $k \geq 2$ servers, our model assumes a malicious-but-non-colluding setting, in which all but one servers may be malicious. Privacy of the retrieved records and integrity of the database are preserved as long as one server behaves honestly.

A New Scheme (§ 5). We present the first two-server APIR scheme with query complexities sublinear in the database size. Our construction, Tapir, builds on SinglePass [LP24], a state-of-the-art two-server PIR protocol with efficient client preprocessing but no authentication guarantees. To provide integrity, we combine SinglePass with a vector commitment (VC) scheme, carefully ensuring protection against selective failure attacks. This protection is not trivial, as a client should abort with equal probability when receiving incorrect information regardless of the queried record. Our scheme treats the VC as a black-box primitive, only requiring that the commitment be deterministic. This design offers implementation simplicity for practitioners while ensuring direct benefits from future advances in VC constructions. Additionally, our approach offers efficiency trade-offs between runtime and bandwidth depending on the choice of commitment scheme used. We demonstrate these trade-offs experimentally in §6. For example, instantiating Tapir with Merkle trees yields smaller server runtime than all prior multi-server APIR schemes, whereas instantiation with Pointproofs [GRWZ20] provides online bandwidth comparable to the unauthenticated SinglePass. This adaptability allows Tapir to be tuned to the requirements of diverse applications.

Support for Updates (§5.2). We extend our scheme to support updates, making Tapir the first APIR scheme with preprocessing that handles dynamic databases in an efficient and provably secure manner. This extension significantly enhances Tapir's utility for applications such as transparency logs, where records are continually appended or modified. We provide new definitions for updatable APIR with preprocessing under adaptive adversaries in a game-based model to ensures that strong security guarantees back our treatment of updates. While SinglePass [LP24] adopts a simulation-based privacy definition that only establishes security for their static scheme under non-adaptive queries, we prove that correctness, integrity, and privacy hold even against malicious adversaries who can adaptively and arbitrarily issue both lookup and update queries.

Evaluation (§6). We implement and evaluate our scheme to showcase its query efficiency compared to prior work. Tapir outperforms both multi-server APIR schemes of [CNC$^+$23] on larger databases with respect to online runtime. Moreover, our scheme performs similarly to non-authenticated state-of-the-art schemes of the same kind in the online phase, while incurring small overhead due to authentication. Concretely, Tapir requires as little as 0.11 % more bandwidth than SinglePass for databases of size $N = 2^{20}$ when instantiated with a state-of-the-art algebraic VC scheme such as Pointproofs [GRWZ20] and 13.11× greater runtime compared to SinglePass when instantiated with Merkle trees. This additional cost is a small price for provable, significantly stronger security guarantees. To assess Tapir's practicality for KT, we conducted our experiments for 32 B records—a typical key size in KT. The results show that our scheme scales well for larger databases ($N \geq 2^{24}$) while maintaining efficient runtimes and bandwidth for the online phase and for updates. To ensure efficiency for even larger databases, our scheme can be extended in a black-box manner by leveraging database partitioning and future improvements to VCs.

1.2 Prior Work

This section provides an overview of the related works on APIR and multi-server PIR.

Multi-Server PIR. "Traditional" PIR schemes with multiple servers [CGKS95, GI14, BGI15, BGI16] incur server computation linear in the database size. The client-preprocessing model [CK20] shifts much of this cost to an *offline* preprocessing phase, in which the client and server(s) jointly process the database, allowing the client to query the database more efficiently in the *online* phase. This approach has since led to two-server schemes with sublinear online complexities [KC21, LP23, LP24]. SinglePass [LP24] improves over prior client-preprocessing schemes [CK20, KC21, LP23] with an offline phase that only requires a single pass over the database, while also supporting updates.

Authenticated PIR. The aforementioned schemes only guarantee privacy and integrity against honest servers, whereas APIR schemes extend these guarantees to malicious servers and protect against selective failure. Colombo et al. [CNC$^+$23] first formalized APIR and proposed both single- and multi-server schemes. Their single-server schemes require a trusted setup to generate an honest digest. This setting is also used in VeriSimplePIR [dCL24]. The scheme of Dietz and Tessaro [DT24] avoids a trusted setup, but only supports 1-bit records without a concrete cost analysis. Balanced-PIR [AR25], concurrent work to ours, proposed a single-server stateful verifiable PIR scheme that avoids the trusted setup by having the client initially download the database and verify the digest generated by the server. It achieves sublinear amortized computation with small client storage. However, while this scheme also supports updates, its security definitions only capture the privacy of look-up queries.

Multi-server schemes eliminate the need for a trusted digest as long as one server is honest. Colombo et al. [CNC$^+$23] authenticate a linear PIR scheme using Merkle trees. They also propose a second two-server APIR scheme that uses a message authentication code (MAC) [DPSZ12] to verify server responses. We refer to these constructions as APIR-Matrix-MT and APIR-DPF, both of which incur online computation linear in the database size.

Crust [WLZ$^+$23] extends [CK20] to provide verifiability, but its security relies on the strong assumption that the dedicated preprocessing server is honest. This assumption is much stronger than those of related multi-server APIR schemes [CNC+23], which assume at least one of the servers to be honest without specifying which one. Alon and Beimel [AB25] modify the compiler of [EKN22] to generically transform a semi-honest PIR scheme into one with security-with-abort. Falk et al. [FMS25] present a generic compiler that converts a PIR scheme into a malicious-secure PIR scheme in both single- and multi-server settings.

2 Preliminaries

We denote the set $\{1, ..., n\}$ by $[n]$ and vectors in boldface, e.g., $\mathbf{v} = (v_1, ..., v_n)$. For some subset of indices $S \subseteq [n]$, we use $\mathbf{v}[S] = (v_i : i \in S)$ to denote the subvector indexed by S.

We consider a database DB with $N \in \mathbb{N}$ records, each of size ℓ_r, i.e., $\mathsf{DB} \in \{0,1\}^{N \times \ell_r}$. Our scheme divides the database into Q partitions of size M where $Q = \lceil N/M \rceil$. We denote the q-th partition by DB_q. Our scheme also builds a second database $\widehat{\mathsf{DB}}$ that contains opening proofs, each of size ℓ_p. We let $\mathsf{DB}_q[m]$ denote the m-th record in DB_q and $\widehat{\mathsf{DB}}_q[m]$ denote the corresponding proof for $m \in [M]$. The databases are padded to their maximum capacity of $M \cdot Q$ with 0-records if $N < M \cdot Q$.

To allow database updates, initial fixed values are denoted by N_{init} for the database size, Q_{init} for the number of partitions, and $\mathsf{DB}_{\mathsf{init}}$ for the sequence of records at the start of the protocol. During protocol execution, these values may change; hence, we use N, Q, and DB, respectively.

For clarity and convenience, we also use the dot notation from object-oriented programming to refer to a particular element of a tuple. For example, if the server state is $\mathsf{st}_S = (\mathsf{DB}, \widehat{\mathsf{DB}})$, then we write $\mathsf{st}_S.\mathsf{DB}$ to access the database directly. We also use this notation to append values to a tuple, e.g., $\mathsf{st}_C.\mathsf{pp} \leftarrow \mathsf{pp}$ denotes adding public parameters to the client's state. In text, we might refer to common values directly instead of using the above notation, e.g., N instead of $\mathsf{pp}.N$.

The symbol $\perp$ either denotes an empty value or that verification has been unsuccessful and that the client has aborted.

2.1 Vector Commitments

Vector commitments (VCs) enable a party to commit to a sequence of elements and later open the commitment and prove the value at a particular index. Merkle trees are one way to implement such a primitive. However, vector commitments often additionally require that the proof be succinct, i.e., that the size of the proof is independent of the length of the vector. Examples of such succinct constructions include the original work on VCs [CF13] and Pointproofs [GRWZ20].

Definition 1 ([CF13]). An **updateable vector commitment** VC is a tuple of five algorithms with the following syntax:

- $\mathsf{pp}_{\mathsf{VC}} \leftarrow \mathsf{ParamGen}(1^\lambda, n)$ takes a security parameter λ and the size of vectors to be committed $n = \mathsf{poly}(\lambda)$. It outputs public parameters $\mathsf{pp}_{\mathsf{VC}}$. We assume the public parameters to be an implicit input to the remaining algorithms.
- $V \leftarrow \mathsf{Commit}(\mathbf{v})$ takes a vector $\mathbf{v}$ and outputs commitment V.
- $\pi \leftarrow \mathsf{Open}(i, \mathbf{v})$ takes as input index $i \in [n]$ and a vector $\mathbf{v}$ and it outputs a proof π.
- $b \leftarrow \mathsf{Verify}(V, S, \mathbf{v}[S], \pi)$ takes as input a commitment V, index subset $S \subseteq [n]$, values $\mathbf{v}[S]$, and a proof π. It outputs bit $b = 1$ if V is consistent with $\mathbf{v}[S]$ and $b = 0$ otherwise.
- $V' \leftarrow \mathsf{Update}(V, S, \mathbf{v}[S], \mathbf{v}'[S])$ takes as input a commitment V and updates the values at the indices in S from $\mathbf{v}[S]$ to values $\mathbf{v}'[S]$. It outputs a commitment V' to the updated vector.

For security, we require that a commitment cannot be opened to different values at the same index. More formally, a vector commitment scheme is **binding** if for every n and every adversary running in time $\mathsf{poly}(\lambda)$, the probability of finding a commitment V and tuples $(S, \mathbf{v}, \pi)$, $(S', \mathbf{v}', \pi')$ such that $\mathbf{v}[S \cap S'] \neq \mathbf{v}'[S \cap S']$ and

$$\mathsf{Verify}(V, S, \mathbf{v}, \pi) = \mathsf{Verify}(V, S', \mathbf{v}', \pi') = 1$$

is negligible in λ.

In the context of our APIR protocol, the **hiding** property is not required, since we assume that the database is public. For correctness, our protocol requires Commit to be **deterministic**. This is the case for Pointproofs and Merkle trees, the two VC schemes considered in this work.

2.2 Permutations

Generating permutations. Random permutations over "small" domains can be sampled using the Fisher-Yates shuffle, also known as the Fisher-Yates-Durstenfeld-Knuth shuffle [Dur64, Knu97, FY53]. At a high level, the algorithm takes as input a list of elements (from the set we wish to permute) and traverses the list, each time swapping the current element with a random element from the remaining list. This algorithm can sample an unbiased permutation of the set $[N]$ in $O(N)$ time. We restate the following lemma.

Lemma 2.2.1 ([Dur64, Knu97, FY53]). *For any $N \in \mathbb{N}$, there exists an algorithm $\mathsf{Permute}(N)$ that can output a permutation σ_N of the set $[N]$ sampled uniformly from the set of all permutations of $[N]$, in $O(N)$ time.*

Both SinglePass and this work rely on this lemma for efficient and secure permutation generation.

Cycle notation. In our examples, we use *cycle notation* to compactly represent permutations. Cycle notation is used to describe a permutation $\sigma : S \to S$ as a list of disjoint cycles. Each cycle follows an element in S by repeatedly applying σ until the starting element is reached. As a concrete example, consider the permutation $\sigma = (4\ 3\ 1\ 2)$ defined by $4 \mapsto 3, \quad 3 \mapsto 1, \quad 1 \mapsto 2, \quad 2 \mapsto 4$.

If all elements in S are contained in the cycle, then we are done, otherwise we pick an element in S not in the previous cycle(s) and start a new cycle:

$$\sigma : (x \ \ \sigma(x) \ \ \sigma(\sigma(x)) \ \ ...)(y \ \ \sigma(y) \ \ \sigma(\sigma(y)) \ \ ...) \ ...$$

where $x, y \in S$. As a concrete example, consider the permutation σ' defined by $1 \mapsto 3, \quad 2 \mapsto 1, \quad 3 \mapsto 2, \quad 4 \mapsto 4$. which can be expressed in cycle notation as $(1\ \sigma'(1)\ \sigma'(\sigma'(1)))(4) = (1\ 3\ 2)(4)$.

2.3 Private Information Retrieval

Private Information Retrieval (PIR) allows a client to retrieve an element of a database without the server holding that database being able to determine which element was retrieved. While this problem admits a trivial solution that achieves perfect privacy—the client downloads the entire database—the communication complexity is linear in the database size. This overhead is problematic for many real-world use cases where databases are large and client storage is limited. PIR therefore focuses on succinctness to achieve query communication complexity sublinear in the database size.

State-of-the-art two-server PIR schemes achieve both sublinear query communication and computational cost under computational security guarantees [CK20, KC21, LP23, LP24]. In these schemes, the servers and client jointly preprocess the database to compute a hint that allows the client to efficiently query the database.

Standard PIR threat models typically assume non-colluding, semi-honest servers and only guarantee privacy under these assumptions. However, in practice—especially when PIR is used as a subprotocol within a larger system—a malicious server may choose to serve a wrong answer and observe whether a client aborts. This side-channel enables an attack known as a *selective failure attack*, which can leak information about the queried index and violate the query privacy guarantees. To provide privacy in the presence of this side-channel, new security definitions and constructions for APIR have been proposed e.g., [CNC+23, FMS25, AB25, DT24, dCL24, AR25]. In the next section, we extend the prior work and formally introduce two-server APIR with client-preprocessing and support for updates.

3 APIR with Preprocessing

In this section, we formalize APIR with preprocessing (Def. 2) and support for updates (Def. 3). An updatable APIR scheme allows the servers to update the database by editing and deleting existing values and adding new values.

We further state the threat model (§ 3.1) and define the required properties of correctness (§ 3.2.1), integrity (§ 3.2.2), and privacy with abort (§ 3.2.3). Our formalization captures APIR schemes with $k \geq 1$ servers, thus it also captures related work [AR25]. We note that the security definitions for the single- and multi-server settings differ in their threat model. While the multi-server setting assumes at least one honest server, this is not the case in the single-server setting. We address this in § 3.1; the additional steps for $k \geq 2$ servers in the security games and oracles are highlighted in teal.

We distinguish between initial fixed-value protocol parameters and those that might change during protocol execution via, for example, updates (see §2). Additionally, we assume that the database is stored as part of the server state.

Definition 2. A **k-server APIR scheme with preprocessing**, for an input vector $\mathsf{DB}_{\mathsf{init}} \in \{0,1\}^{N_{\mathsf{init}} \times \ell_r}$ where $N_{\mathsf{init}}, \ell_r \in \mathbb{N}$, and setup parameters sp, comprises of the following seven algorithms which all take the security parameter λ as an implicit input:

- $(\mathsf{st}_{S_b}, \mathsf{pp}_b) \leftarrow \mathsf{Setup}(\mathsf{sp}, \mathsf{DB}_{\mathsf{init}})$ is executed by each server $b \in [k]$ and takes as input setup parameters sp and a vector $\mathsf{DB}_{\mathsf{init}}$. It outputs server state st_{S_b} and public parameters pp_b. The server state includes the database DB_b (initialized using $\mathsf{DB}_{\mathsf{init}}$ and possibly modified).
- $(\mathsf{st}_C, \mathbf{hq}) \leftarrow \mathsf{RequestHint}(\mathbf{pp})$ takes the servers' public parameters $\mathbf{pp} = (\mathsf{pp}_b)_{b \in [k]}$ as input and outputs the client state st_C and hint queries $\mathbf{hq} = (\mathsf{hq}_b)_{b \in [k]}$.
- $(\mathsf{st}'_{S_b}, \mathsf{resp}_b) \leftarrow \mathsf{AnsHintReq}(\mathsf{st}_{S_b}, \mathsf{hq}_b)$ is executed by each server $b \in [k]$ and takes as input the server state st_{S_b} and hint query hq_b. It outputs server state st'_{S_b} and response resp_b.
- $(\mathsf{st}'_C, \mathsf{hint}) \leftarrow \mathsf{VerSetup}(\mathsf{st}_C, \mathbf{resp})$ takes as input client state st_C, and server responses $\mathbf{resp} = (\mathsf{resp}_b)_{b \in [k]}$. It outputs updated client state st'_C, and hint hint.
- $(\mathsf{st}'_C, \mathbf{qry}) \leftarrow \mathsf{Query}(\mathsf{st}_C, \mathsf{idx})$ takes as input client state st_C and index $\mathsf{idx} \in [N]$. It returns updated client state st'_C and queries $\mathbf{qry} = (\mathsf{qry}_b)_{b \in [k]}$.
- $(\mathsf{st}'_{S_b}, \mathsf{ans}_b) \leftarrow \mathsf{Answer}(\mathsf{st}_{S_b}, \mathsf{qry}_b)$ is executed by each server $b \in [k]$ and takes as input the server state st_{S_b} and query qry_b. It returns the server state st'_{S_b} and answer ans_b.
- $(\mathsf{st}'_C, x, \mathsf{hint}') \leftarrow \mathsf{Recon}(\mathsf{st}_C, \mathsf{hint}, \mathbf{ans})$ takes as input client state st'_C, hint hint, and answers $\mathbf{ans} = (\mathsf{ans}_b)_{b \in [k]}$. It returns updated client state st'_C, a record $x \in \{0,1\}^{\ell_r}$, and hint hint'.

We include setup parameters sp as part of the input to Setup to account for any values used to parameterize the scheme, including parameters generated during a trusted setup phase (e.g., the public parameters of vector commitments).

Definition 3. An **updateable k-server APIR scheme with preprocessing**, is a two-server APIR scheme with preprocessing, with two additional algorithms:

- $(\mathsf{st}'_{S_b}, \mathsf{pp}'_b, U_b) \leftarrow \mathsf{UpdateDB}(\mathsf{st}_{S_b}, U)$ takes as input server state st_{S_b}, and update information U. It outputs the updated state st'_{S_b}, updated public parameters pp'_b, and update information U_b.
- $(\mathsf{st}'_C, \mathsf{hint}') \leftarrow \mathsf{UpdateHint}(\mathsf{st}_C, \mathsf{hint}, \mathbf{pp}, \mathbf{U})$ takes as input the client state st_C, hint hint, and, from each server $b \in [k]$, the parameters $\mathbf{pp} \leftarrow (\mathsf{pp}_b)_{b \in [k]}$ and update information $\mathbf{U} \leftarrow (U_b)_{b \in [k]}$. It outputs the updated client state st'_C, and hint hint'.

An APIR scheme with preprocessing in both the static (Def. 2) and the dynamic (Def. 3) setting must satisfy *correctness*, *integrity*, and *privacy with abort*. We define these notions in § 3.2 according to the threat model specified below.

3.1 Threat Model

Standard PIR schemes do not provide integrity of the responses and, as such, cannot guarantee correctness if one/the server is malicious. In this work, we consider a stronger threat model in which privacy and integrity are guaranteed even if one or multiple servers act malicious. Malicious servers may deviate from the protocol at any phase of the protocol and may serve incorrect responses or "garbage" to the client.

We consider different threat models for the the single- and multi-server settings: in the single-server setting, privacy, and integrity should still be guaranteed, even if the single server acts malicious. In the multi-server setting, we assume that at least one server is semi-honest and does not collude with the other (possibly malicious) servers. In particular, for our two-server APIR scheme Tapir, this assumption ensures that one of the servers behaves semi-honestly.

We note that if all servers act maliciously, they could (by coincidence or via additional information) modify the database in the same manner, hence fooling the client into accepting incorrect information. Without making strong assumptions about the database, we cannot bound this probability and, therefore, require at least one semi-honest server.

```
Game $\mathsf{G}^{\text{Corr-APIR}}_{\text{APIR}}(\lambda, \mathsf{sp})$

$\textbf{global}$ win, hint, $\mathsf{st}_C$, $(\mathsf{st}_{S_b})_{b\in[k]}$
win $\leftarrow$ false, resp $\leftarrow \perp$, pp $\leftarrow \perp$
// Initialization
$(\mathsf{st}_{\mathcal{A}}, \mathsf{DB}_{\mathsf{init}}) \leftarrow \mathcal{A}_0(\mathsf{sp})$
// Setup
$\textbf{for } b \in [k] \textbf{ do}$
    $(\mathsf{st}_{S_b}, \mathbf{pp}.\mathsf{pp}_b) \leftarrow \mathsf{Setup}(\mathsf{sp}, \mathsf{DB}_{\mathsf{init}})$
// Preprocessing
$(\mathsf{st}_C, \mathbf{hq}) \leftarrow \mathsf{RequestHint}(\mathbf{pp})$
$\textbf{for } b \in [k] \textbf{ do}$
    $(\mathsf{st}'_{S_b}, \mathbf{resp}.\mathsf{resp}_b) \leftarrow \mathsf{AnsHintReq}(\mathsf{st}_{S_b}, \mathbf{hq}[b])$
    $\mathsf{st}_{S_b} \leftarrow \mathsf{st}'_{S_b}$
$(\mathsf{st}'_C, \mathsf{hint}) \leftarrow \mathsf{VerSetup}(\mathsf{st}_C, \mathbf{resp})$
$\mathsf{st}_C \leftarrow \mathsf{st}'_C$
// Oracle phase
$\mathcal{A}_1^{\mathcal{O}_{\mathsf{QueryCorr}}(\cdot), \mathcal{O}_{\mathsf{UpdateCorr}}(\cdot)}(\mathsf{st}_{\mathcal{A}})$
$\textbf{return}$ win
```

```
Games $\boxed{\mathsf{G}^{\text{Int-APIR}}_{\text{APIR}}(\lambda, \mathsf{sp})}$ and $\boxed{\mathsf{G}^{\text{Priv-APIR}}_{\text{APIR}}(\lambda, \mathsf{sp})}$

$\textbf{global}$ $\boxed{\mathsf{win}}$, $\boxed{b_{\mathsf{chall}}}$, good, inpt, hint, $\mathsf{st}_C$, $\mathsf{st}_S$, $t_{\mathsf{q}}$, $t_{\mathsf{u}}$
$\boxed{\mathsf{win} \leftarrow \textbf{false}}$
inpt $\leftarrow \perp$, $t_{\mathsf{q}}, t_{\mathsf{u}} \leftarrow 0$
// Initialization
$(\mathsf{st}_{\mathcal{A}}, \mathsf{good}, (\mathsf{pp}_b)_{b\in[k]\setminus\mathsf{good}}), \mathsf{DB}_{\mathsf{init}}) \leftarrow \mathcal{A}_0(\mathsf{sp})$
// Setup
$(\mathsf{st}_S, \mathsf{pp}_{\mathsf{good}}) \leftarrow \mathsf{Setup}(\mathsf{sp}, \mathsf{DB}_{\mathsf{init}})$
$\boxed{b_{\mathsf{chall}} \leftarrow\$ \{0,1\} \text{ // Pick challenge bit}}$
// Preprocessing
$(\mathsf{st}_C, \mathbf{hq}) \leftarrow \mathsf{RequestHint}((\mathsf{pp}_b)_{b\in[k]})$
$(\mathsf{st}'_S, \mathsf{resp}_{\mathsf{good}}) \leftarrow \mathsf{AnsHintReq}(\mathsf{st}_S, \mathsf{hq}_{\mathsf{good}})$
$(\mathsf{st}'_{\mathcal{A}}, (\mathsf{resp}_b)_{b\in[k]\setminus\mathsf{good}})) \leftarrow \mathcal{A}_1(\mathsf{st}_{\mathcal{A}}, (\mathsf{hq}_b)_{b\in[k]\setminus\mathsf{good}}))$
$(\mathsf{st}'_C, \mathsf{hint}) \leftarrow \mathsf{VerSetup}(\mathsf{st}_C, (\mathsf{resp}_b)_{b\in[k]})$
$\mathsf{st}_C \leftarrow \mathsf{st}'_C$, $\mathsf{st}_S \leftarrow \mathsf{st}'_S$
// Oracle phase
$\boxed{\mathcal{A}_2^{\mathcal{O}_{\mathsf{QueryInt}}(\cdot), \mathcal{O}_{\mathsf{ReconInt}}(\cdot), \mathcal{O}_{\mathsf{UpdateInt}}(\cdot)}(\mathsf{st}'_{\mathcal{A}})}$
$\boxed{b' \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathsf{QueryPriv}}(\cdot), \mathcal{O}_{\mathsf{ReconPriv}}(\cdot), \mathcal{O}_{\mathsf{UpdatePriv}}(\cdot)}(\mathsf{st}'_{\mathcal{A}})}$
$\boxed{\textbf{return } \mathsf{win}}$ $\boxed{\textbf{return } b_{\mathsf{chall}} = b'}$
```

Figure 1: The games for APIR **correctness (left), integrity (right, solid box), and privacy (left, dashed box). The corresponding oracles are defined in Fig. 2. Highlighted code only applies to the multi-server setting ($k \geq 2$).**

3.2 Security Properties

In this section, we define the security properties of an updatable APIR scheme with preprocessing (Def. 3). We note that the definitions for APIR with a static database are the same, except the adversary has no access to any Update oracles. These static definitions are excluded for succinctness.

Before we go into the details of these definitions, we provide some context for our game-based security definitions and the capabilities of the adversary. We define the security games in Fig. 1 and the corresponding oracles in Fig. 2. The code and descriptions that apply only to the multi-server setting—where we assume a "good" server—are highlighted in teal.

Let APIR be a (possibly updateable) two-server APIR scheme with preprocessing. The games take as input security parameter λ and setup parameters sp, and they track global variables throughout oracle calls. These variables include the state of the client and of the honest server.

We assume that all adversary outputs are well-formed and within the correct domain. The adversary has the capability to choose the initial set of records $\mathsf{DB}_{\mathsf{init}}$ for the protocol execution in all games. In the privacy and integrity games, the adversary also chooses an index good $\in [k]$ that defines which server is honest. Additionally, the adversary is adaptive and has access to a set of oracles. These oracles can be executed in any order and depend on the game. We assume an efficient adversary that makes at most T_{q} calls to a query oracle and at most T_{u} calls to an update oracle, such that $T_{\mathsf{q}} + T_{\mathsf{u}} \leq \mathsf{poly}(\lambda)$.

The games ensure that the algorithms of the offline phase are executed in the required order, i.e., Setup, RequestHint, AnsHintReq, VerSetup. The games, oracles, and protocol description could be adapted to move this requirement into the protocols themselves. We exclude this here for simplicity.

3.2.1 *Correctness.*

An APIR scheme is correct if, when a client requests the record at index idx, it receives the correct value stored at that index in the database, i.e., DB[idx]. Correctness is guaranteed to hold only when both the client and the servers behave honestly.

Definition 4. Let APIR be an updatable k-server APIR scheme with preprocessing, let $\lambda \in \mathbb{N}$ be the security parameter, and let sp be a well-formed and honestly generated setup parameter. Let $\mathsf{G}^{\text{Corr-APIR}}_{\text{APIR}}$ be the correctness game for APIR defined in Fig. 1 and Fig. 2. The advantage of an adversary $\mathcal{A}$ playing this game is defined as

$$\mathsf{Adv}^{\mathsf{corr}}_{\mathcal{A},\mathsf{APIR}}(\lambda) := \Pr\left[\mathsf{G}^{\text{Corr-APIR}}_{\text{APIR}}(\lambda, \mathsf{sp})\right].$$

We say that APIR is **correct** if $\mathsf{Adv}^{\mathsf{corr}}_{\mathcal{A},\mathsf{APIR}}(\lambda) = 0$ for all PPT $\mathcal{A}$.

3.2.2 *Integrity.*

We define integrity against adaptive queries (and updates) for APIR schemes. Informally, an APIR scheme satisfies integrity if the client outputs either the correct value or aborts with $\perp$, except with negligible probability $\mathsf{negl}(\lambda)$.

Definition 5. Let APIR be an updatable k-server APIR scheme with preprocessing, let $\lambda \in \mathbb{N}$ be the security parameter, and let sp be a well-formed and honestly generated setup parameter. Let $\mathsf{G}^{\text{Int-APIR}}_{\text{APIR}}$ be the adaptive integrity game for APIR defined in Fig. 1 and Fig. 2.

```
Oracle $O_{\mathsf{QueryInt}}(\mathsf{idx})$
  $(\mathsf{st}'_C, \mathbf{qry}) \leftarrow \mathsf{Query}(\mathsf{st}_C, \mathsf{idx})$
  $(\mathsf{st}'_S, \mathsf{ans}_{\mathsf{good}}) \leftarrow \mathsf{Answer}(\mathsf{st}_S, \mathbf{qry}[\mathsf{good}])$
  $\mathsf{inpt} \leftarrow (\mathsf{ans}_{\mathsf{good}}, \mathsf{idx})$
  $\mathsf{st}_S \leftarrow \mathsf{st}'_S, \mathsf{st}_C \leftarrow \mathsf{st}'_C$
  $\textbf{return } \mathbf{qry}[[k] \setminus \mathsf{good}]$

Oracle $O_{\mathsf{ReconInt}}((\mathsf{ans}_b)_{b \in [k] \setminus \mathsf{good}})$
  $(\mathsf{ans}_{\mathsf{good}}, \mathsf{idx}) \leftarrow \mathsf{inpt}$
  $(\mathsf{st}'_C, x, \mathsf{hint}') \leftarrow \mathsf{Recon}(\mathsf{st}_C,$
      $\mathsf{hint}, (\mathsf{ans}_b)_{b \in [k]})$
  $\textbf{if } x \neq \bot \wedge x \neq \mathsf{DB}[\mathsf{idx}] \textbf{ then}$
    $\mathsf{win} \leftarrow \textbf{true}$
  $\mathsf{st}_C \leftarrow \mathsf{st}'_C,\ \mathsf{hint} \leftarrow \mathsf{hint}',\ \mathsf{inpt} \leftarrow (\bot, \bot)$

Oracle $O_{\mathsf{UpdateInt}}((\mathsf{pp}_b)_{b \in [k] \setminus \mathsf{good}}, (U_b)_{b \in [k] \setminus \mathsf{good}}, U)$
  $(\mathsf{st}'_S, \mathsf{pp}_{\mathsf{good}}, U_{\mathsf{good}}) \leftarrow \mathsf{UpdateDB}(\mathsf{st}_S, U)$
  $(\mathsf{st}'_C, \mathsf{hint}') \leftarrow \mathsf{UpdateHint}(\mathsf{st}_C,$
      $\mathsf{hint}, (\mathsf{pp}_b)_{b \in [k]}, (U_b)_{b \in [k]})$
  $\mathsf{st}_C \leftarrow \mathsf{st}'_C,\ \mathsf{hint} \leftarrow \mathsf{hint}', \mathsf{st}_S \leftarrow \mathsf{st}'_S$

Oracle $O_{\mathsf{QueryPriv}}(\mathsf{idx}_0, \mathsf{idx}_1)$
  $\textbf{if } t_{\mathsf{q}} < T_{\mathsf{q}} \textbf{ then}$
    $(\mathsf{st}'_C, \mathbf{qry}) \leftarrow \mathsf{Query}(\mathsf{st}_C, \mathsf{idx}_{b_{\mathsf{chall}}})$
    $(\mathsf{st}'_S, \mathsf{ans}_{\mathsf{good}}) \leftarrow \mathsf{Answer}(\mathsf{st}_S, \mathbf{qry}[\mathsf{good}])$
    $\mathsf{inpt} \leftarrow \mathsf{ans}_{\mathsf{good}}$
    $\mathsf{st}_S \leftarrow \mathsf{st}'_S, \mathsf{st}_C \leftarrow \mathsf{st}'_C,\ t_{\mathsf{q}} \leftarrow t_{\mathsf{q}} + 1$
    $\textbf{return } \mathbf{qry}[[k] \setminus \mathsf{good}]$

Oracle $O_{\mathsf{ReconPriv}}((\mathsf{ans}_b)_{b \in [k] \setminus \mathsf{good}})$
  $\mathsf{ans}_{\mathsf{good}} \leftarrow \mathsf{inpt}$
  $(\mathsf{st}'_C, x, \mathsf{hint}') \leftarrow \mathsf{Recon}(\mathsf{st}_C, \mathsf{hint}, (\mathsf{ans}_b)_{b \in [k]})$
  $\textbf{if } x = \bot \textbf{ then}$
    $\mathsf{abort\text{-}bit} \leftarrow \textbf{true}$
  $\textbf{else } \mathsf{abort\text{-}bit} \leftarrow \textbf{false}$
  $\mathsf{st}_C \leftarrow \mathsf{st}'_C,\ \mathsf{hint} \leftarrow \mathsf{hint}',\ \mathsf{inpt} \leftarrow \bot$
  $\textbf{return } \mathsf{abort\text{-}bit}$

Oracle $O_{\mathsf{UpdatePriv}}((\mathsf{pp}_b)_{b \in [k] \setminus \mathsf{good}}, (U_b)_{b \in [k] \setminus \mathsf{good}}, U)$
  $\textbf{if } t_{\mathsf{u}} < T_{\mathsf{u}} \textbf{ then}$
    $(\mathsf{st}'_S, \mathsf{pp}_{\mathsf{good}}, U_{\mathsf{good}}) \leftarrow \mathsf{UpdateDB}(\mathsf{st}_S, U)$
    $(\mathsf{st}'_C, \mathsf{hint}') \leftarrow \mathsf{UpdateHint}(\mathsf{st}_C,$
        $\mathsf{hint}, (\mathsf{pp}_b)_{b \in [k]}, (U_b)_{b \in [k]})$
    $t_{\mathsf{u}} \leftarrow t_{\mathsf{u}} + 1$
    $\mathsf{st}_S \leftarrow \mathsf{st}'_S,\ \mathsf{st}_C \leftarrow \mathsf{st}'_C,\ \mathsf{hint} \leftarrow \mathsf{hint}'$

Oracle $O_{\mathsf{QueryCorr}}(\mathsf{idx})$
  $(\mathsf{st}'_C, \mathbf{qry}) \leftarrow \mathsf{Query}(\mathsf{st}_C, \mathsf{idx})$
  $\mathbf{ans} \leftarrow \bot$
  $\textbf{for } b \in [k] \textbf{ do}$
    $(\mathsf{st}'_{S_b}, \mathbf{ans}.\mathsf{ans}_b) \leftarrow \mathsf{Answer}(\mathsf{st}_{S_b}, \mathbf{qry}[b])$
    $\mathsf{st}_{S_b} \leftarrow \mathsf{st}'_{S_b}$
  $(\mathsf{st}''_C, x, \mathsf{hint}') \leftarrow \mathsf{Recon}(\mathsf{st}'_C, \mathsf{hint}, \mathbf{ans})$
  $\textbf{if } x \neq \bot \wedge x \neq \mathsf{DB}[\mathsf{idx}]) \textbf{ then}$
    $\mathsf{win} \leftarrow \textbf{true}$
  $st_C \leftarrow \mathsf{st}''_C,\ \mathsf{hint} \leftarrow \mathsf{hint}'$

Oracle $O_{\mathsf{UpdateCorr}}(U)$
  $(\mathbf{U}, \mathbf{pp}) \leftarrow \bot^2$
  $\textbf{for } b \in [k] \textbf{ do}$
    $(\mathsf{st}'_{S_b}, \mathbf{pp}.\mathsf{pp}_b, \mathbf{U}.U_b) \leftarrow \mathsf{UpdateDB}(\mathsf{st}_{S_b}, U)$
    $\mathsf{st}_{S_b} \leftarrow \mathsf{st}'_{S_b}$
  $(\mathsf{st}'_C, \mathsf{hint}') \leftarrow \mathsf{UpdateHint}(\mathsf{st}_C, \mathsf{hint}, \mathbf{pp}, \mathbf{U})$
  $\mathsf{st}_C \leftarrow \mathsf{st}'_C, \mathsf{hint} \leftarrow \mathsf{hint}'$
```

Figure 2: Oracles for $\mathsf{G}^{\mathsf{Int\text{-}APIR}}_{\mathsf{APIR}}$ (top), $\mathsf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ (middle), and $\mathsf{G}^{\mathsf{Corr\text{-}APIR}}_{\mathsf{APIR}}$ (bottom). Highlighted code only applies if $k \geq 2$.

The advantage of an adversary $\mathcal{A}$ playing this game is defined as

$$\mathsf{Adv}^{\mathsf{int}}_{\mathcal{A},\mathsf{APIR}}(\lambda) := \Pr\left[\mathsf{G}^{\mathsf{Int\text{-}APIR}}_{\mathsf{APIR}}(\lambda, \mathsf{sp})\right].$$

We say that APIR achieves **integrity** against adaptive queries and updates if $\mathsf{Adv}^{\mathsf{int}}_{\mathcal{A},\mathsf{APIR}}(\lambda) \leq \mathsf{negl}(\lambda)$ for all PPT $\mathcal{A}$.

3.2.3 Privacy. An APIR scheme is considered private if the server(s) gain no information about which record the client queries. This guarantee must hold even if the server(s) learn whether the client outputs the error symbol $\bot$ and aborts during reconstruction. This stronger notion of privacy was introduced in [CNC+23] and is designed to protect against selective-failure attacks. In this work, we formalize privacy against adaptive queries and updates via a left-or-right indistinguishability game. At the start of the game, a random bit b_{chall} is chosen. The adversary submits two indices to the $O_{\mathsf{QueryPriv}}$ oracle, but only the b_{chall}-th index is used to execute the query. At the end of the game, the adversary must guess which bit was selected.

Definition 6. Let APIR be an updatable k-server APIR scheme with preprocessing, let $\lambda \in \mathbb{N}$ be the security parameter, and let sp be a well-formed and honestly generated setup parameter. Let $\mathsf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ be the privacy game against adaptive queries (and updates) for APIR defined in Fig. 1 and Fig. 2. The advantage of an adversary $\mathcal{A}$ playing this game is defined as

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathcal{A},\mathsf{APIR}}(\lambda) := \left|\Pr\left[\mathsf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}(\lambda, \mathsf{sp})\right] - \frac{1}{2}\right|.$$

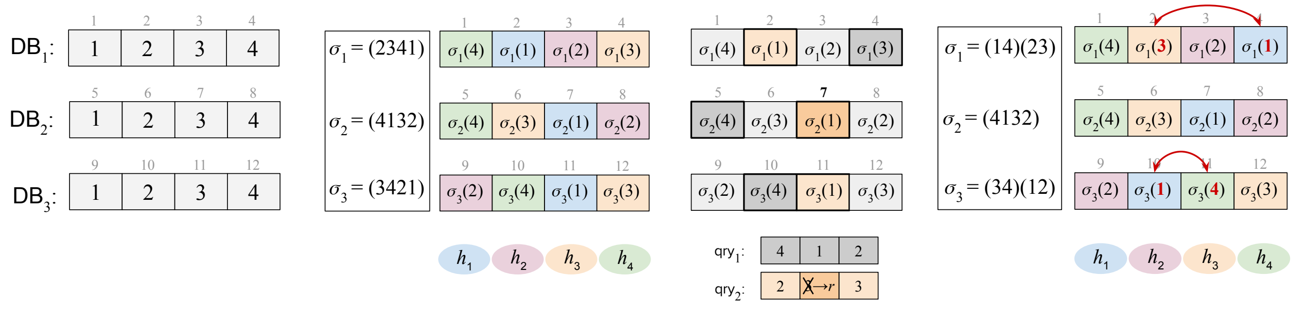
We say that APIR achieves **privacy** against adaptive queries if $\mathsf{Adv}^{\mathsf{priv}}_{\mathcal{A},\mathsf{APIR}}(\lambda) \leq \mathsf{negl}(\lambda)$ for all PPT $\mathcal{A}$.

4 Upgrading SinglePass to Malicious Security

We start by providing a brief overview of the unauthenticated two-server PIR scheme with client-preprocessing, SinglePass [LP24]. We then outline the challenges and solutions to upgrading SinglePass to malicious security.

4.1 An Overview of SinglePass

To achieve privacy in SinglePass, the two servers are assigned distinct roles. Server 1 is responsible for computing the hint during the offline phase and providing updates to it during the online phase. These hint updates ensure that queries remain independent over time, thereby preventing information leakage across multiple queries. Server 2 is responsible for supplying the values required for the client to reconstruct the queried record during the online phase. Crucially, because the hint is used to generate the client's queries, the hint server must not have access to the values



(a) Database DB and its partition. The index of each record within its partition is shown inside the box and the corresponding index in the original database is indicated above. **(b) Original permutation and hints. The i-th hint is computed by XORing the records at indices $\sigma_j(i)$ for $j \in [3]$ (records contributing to the same hint are highlighted in the same color).** **(c) Queries for retrieving $\mathsf{DB}[7] = \mathsf{DB}_2[3]$ are $\mathsf{qry}_2 = [\sigma_1(1), r, \sigma_3(1)]$ where $r \xleftarrow{\$} [4]$ and qry_1 is randomly sampled.** **(d) Updated permutation and hints. The client updates the permutations by swapping the mappings highlighted in (c). The hints are then updated accordingly.**

Figure 3: An example of the SinglePass scheme [LP24] for a database DB with $N = 12$, $Q = 3$, and $M = N/Q = 4$.

used to reconstruct the queried record. Otherwise, it could correlate the hint with the responses and infer the client's queried index.

We begin by assuming the servers share a database DB of size N, which is partitioned into Q disjoint sub-arrays, each of size M. The q-th partition is denoted $\mathsf{DB}_q = \mathsf{DB}[(q-1)\cdot M + 1 : q \cdot M]$, where $Q = \lceil N/M \rceil$. The database is padded to its maximum capacity.

We provide a running example of the scheme in Fig. 3, and show a database partition for $N = 12$ and $Q = 3$ in Fig. 3a.

Offline phase. The offline phase starts with Server 1 sampling a permutation σ_q for each partition DB_q, $q \in [Q]$. For each $m \in [M]$, the server computes the hint $h_m = \bigoplus_{q\in[Q]} \mathsf{DB}_q[\sigma_q(m)]$ and sends $\mathsf{hint} = (h_1, ..., h_M)$ to the client (with the seeds for generating the permutations). In Fig. 3b, we give an example of three permutations and the hint they result in. Concretely, for h_1, we have that

$$h_1 = \bigoplus_{q\in[3]} \mathsf{DB}_1[\sigma_q(1)] = \mathsf{DB}_1[2] \oplus \mathsf{DB}_2[3] \oplus \mathsf{DB}_3[3].$$

Online phase. To issue a query for an index $\mathsf{idx} \in [N]$, the client must first find the partition, q^*, and index within the partition, m^*, that map to the index idx in the original database, i.e., $\mathsf{DB}[\mathsf{idx}] = \mathsf{DB}_{q^*}[m^*]$. The client then computes $\mathsf{ind} \in [M]$ such that $\sigma_{q^*}(\mathsf{ind}) = m^*$ and an array of indices defined as $\mathsf{qry}_2 = [\sigma_q(\mathsf{ind}) : q \in [Q]]$. Next, the client replaces $\mathsf{qry}_2[m^*]$ with a randomly sampled value in $[M]$ (thereby hiding the index of interest and ensuring query privacy). In Fig. 3c, we give an example for querying $\mathsf{idx} = 7$ which corresponds to $(q^*, m^*) = (2, 3)$. The client computes $\mathsf{ind} = 1$ since $\sigma_2(1) = 3$ and sets $\mathsf{qry}_2 = [\sigma_1(1), r, \sigma_3(1)] = [2, r, 3]$ where $r \leftarrow\!\!\$\ [4]$ is random.

In parallel to computing array qry_2, the client also samples $r_1, ..., r_Q \leftarrow\!\!\$\ [M]^Q$, and sets $\mathsf{qry}_1 = [\sigma_q(r_q) : q \in [Q]]$. The arrays qry_1 and qry_2 are then sent to the respective servers. The servers respond by sending an array of the records at the requested indices.

Given the answer from Server 2, $[\mathsf{DB}_q[\mathsf{qry}_2[q]] : q \in [Q]]$, the client can recover the desired record as follows:

$$\mathsf{DB}[\mathsf{idx}] = \mathsf{DB}_{q^*}[m^*] = \left(\bigoplus_{\substack{q\in[Q] \\ q\neq q^*}} \mathsf{DB}_q[\mathsf{qry}_2[q]] \right) \oplus h_{\mathsf{ind}}.$$

Finally, the client takes the answer from Server 1, $[\mathsf{DB}_q[\mathsf{qry}_1[q]] : q \in [Q]]$, and updates the hint, such that $q \neq q^*$ for $q \in [Q]$:

$$h_{\mathsf{ind}} = h_{\mathsf{ind}} \oplus \mathsf{DB}[\mathsf{qry}_1[q]] \oplus \mathsf{DB}[\mathsf{qry}_2[q]]$$
$$h_{r_q} = h_{r_q} \oplus \mathsf{DB}[\mathsf{qry}_1[q]] \oplus \mathsf{DB}[\mathsf{qry}_2[q]].$$

Figure 3d gives an example of the refreshed hint.

The privacy of SinglePass relies on the fact that the client sends a pseudorandom vector of indices—based on the permutations—to each server. If the queried index is in the set, the client replaces it with a random index from the same range. The server receiving this vector does not know the permutations and, thus, cannot identify which, or even if, an index was replaced. Lazzaretti and Papamanthou [LP24] state and prove this with their Show-and-Shuffle theorem which we restate in Theorem A.1.1 in Appendix A.1.

4.2 Upgrading to Malicious Security

We now outline the challenges we encountered in extending SinglePass to guarantee malicious security and privacy with abort.

Challenge 1: Computing the hint privately. In SinglePass, the client and one server jointly preprocess the database to compute a hint stored at the client. Upgrading this step to be secure against a malicious server poses a challenge: a malicious server cannot be trusted to compute the hint correctly. Our key insight is to make the hint generation both private and verifiable. This could be achieved using heavy cryptographic tools (e.g., distributed multipoint functions [KKEPR24, BGH+25] or custom MPC protocols), but we prioritize computational efficiency. Instead, we assume a streaming model, in which both servers stream the database to the client. The client checks that the records streamed from both servers match and then computes the hint locally. Streaming—a standard technique in PIR [ZPZS24, GZS24, GZSP25, AR25]—leaks no information to the servers and significantly reduces their computational cost compared to, e.g., SinglePass. Instead, the client runs the preprocessing itself.

Challenge 2: Protecting against selective failure. Note that the servers in prior multi-server APIR schemes [CNC+23] have symmetric roles, i.e., both servers do the same operations (on different inputs). In existing PIR schemes with client-preprocessing, one server typically generates and updates the hint, while the other only answers queries. In contrast, to ensure privacy in the offline phase against a potentially malicious server, we require both servers to participate in the setup identically. This symmetry prevents either server from gaining an advantage in violating privacy. To achieve this, we adopt a stream-and-compare approach, as mentioned above. Other approaches may also be possible, e.g., by using zero-knowledge proof techniques, which we leave as a direction for future work.

In addition, we leverage vector commitments [CF13] to verify the correctness of queried records. Vector commitments can be viewed as a generalization of Merkle trees and Pedersen vector commitments, both of which have been used in prior work [CNC$^+$23, DT24, FMS25, dCL24].

Challenge 3: Guaranteeing security of updates. Another challenge was proving that integrity and query privacy hold when adaptive lookup and update queries are made to the database. In contrast, the original paper [LP24] only proved security for queries to static databases, and their security definition only considered non-adaptive queries. We address this challenge by providing new game-based security notions for updateable APIR with preprocessing. Our games (see Fig. 1) consider an adversary with access to query, reconstruction, and update oracles. The adversary may adaptively choose to make a polynomial number of queries, and may even interleave lookup and update queries arbitrarily. Care was taken to strengthen our definitions by ensuring that trivial wins/losses were excluded.

5 The TAPIR Scheme

Our scheme proceeds in two phases: an *offline* phase, where the client preprocesses the database, followed by an *online* phase, where the client queries the server based on the preprocessed information. The preprocessing phase allows communication and computational complexities of the online phase to be sublinear in the database size. This section first provides a high-level description of TAPIR for static databases (§5.1) before extending it to updatable databases (§5.2). Lastly, §5.3 states the correctness, integrity, and security of TAPIR.

5.1 Scheme Overview

Our scheme allows the client to abort in most stages of the protocol if it does not accept the input from the servers. This is denoted by the client's internal abort state $\mathsf{st}_C.\mathsf{abort}$ being set to **true**. Once this state is set to **true**, the client stops all computation and only outputs $\perp$. On input $\perp$, the servers also directly output $\perp$. The parties do not proceed with any actual retrievals or computation; thus, the protocol is effectively aborted. SinglePass, and consequently our scheme, requires that the algorithms be executed in a strict order. In particular, the client must run Recon after Query and before invoking either Query again or UpdateHint. This ordering is enforced at the start of the client's online algorithms by checking whether $\mathsf{st}_C.q^*$ is set (for Recon) or unset (otherwise).

5.1.1 Setup parameters. The setup parameters $\mathsf{sp} = (N_{\mathsf{init}}, Q_{\mathsf{init}}, M, \ell_r, \lambda)$ specify the initial database size N_{init}, the number of partitions Q_{init}, the size of each partition M, the size of each record ℓ_r, and the security parameter λ. Our scheme is initialized with these setup parameters sp and a vector of database records $\mathsf{DB}_{\mathsf{init}} \in 0, 1^{N_{\mathsf{init}} \times \ell_r}$. The number of partitions is $Q_{\mathsf{init}} = \lceil N_{\mathsf{init}}/M \rceil$, which is tunable, allowing for a trade-off between communication cost and computation time. The setup parameters also include any values required to parameterize the underlying VC scheme, which we leave implicit.

5.1.2 Offline phase. The Setup algorithm—run by each server—begins by partitioning $\mathsf{DB}_{\mathsf{init}}$ into Q_{init} partitions of size M and padding as necessary to obtain DB. For each $q \in [Q_{\mathsf{init}}]$, the servers compute a vector commitment com_q to each partition DB_q such that the m-th record committed to by com_q corresponds to record $\mathsf{DB}_q[m]$. For every record in DB, the servers also compute an opening proof, storing it in a secondary database $\widehat{\mathsf{DB}}$ of the same size as DB, with each proof stored at the same index as its corresponding record. Using the setup parameters, the servers initialize the public parameters pp, which include the database metadata, security parameters, and the database digest. Both servers maintain copies of DB and $\widehat{\mathsf{DB}}$, along with pp, as part of their state.

When a client joins the system, it first runs RequestHint, during which it requests two things from the servers: (1) the public parameters and associated vector commitments that serve as a digest of the database, and (2) a streamed copy of the database from which it can compute a SinglePass-like hint. It first initializes the abort bit $\mathsf{st}_C.\mathsf{abort} \leftarrow$ **false**. To obtain (1), the client downloads the public parameters pp_1 and pp_2 from Servers 1 and 2, respectively, and checks if $\mathsf{pp}_1 \neq \mathsf{pp}_2$. Since at least one server is honest, equality guarantees that the digest was generated consistently. If the public parameters differ, the client aborts. Otherwise, the client proceeds to obtain (2) by sending a hint request to both servers.

On input of hint query hq_b, Server $b \in \{1, 2\}$ runs AnsHintReq. If $\mathsf{hq}_b \neq \perp$, the server proceeds to stream a copy of the database. Streaming allows the client to verify the equality of corresponding records across both servers, mitigating the risk of a malicious server providing inconsistent data. If the received information is not equal, the client aborts. Otherwise, it samples secret random permutations $\sigma_1, ..., \sigma_Q : [M] \rightarrow [M]$—one for each partition of size M—and computes the hint as $h_m = \bigoplus_{q=1}^{Q} \mathsf{DB}_q[\sigma_q(m)]$ and sets $\mathsf{hint} = (h_1, ..., h_M)$. This ensures that the hint is consistent with the committed database while hiding its value from both servers. The client then stores the verified public parameters pp, the computed hint hint, and the client key ck (i.e., the set of secret permutations) as part of its state, completing the offline phase.

5.1.3 Online phase. To retrieve a record from the database, the client runs Query on this record's index idx corresponding to $(q^*, m^*) \in [\mathsf{pp}.Q] \times [\mathsf{pp}.M]$. If the client aborted earlier, it simply outputs $\perp$ to the servers. Otherwise, the client continues with the protocol and issues a query to each server. These queries are computed as in SinglePass and consist of an array of database indices.

On receiving query qry_b, server b executes Answer. If the client did not abort, the server returns the records at the requested indices, along with the corresponding opening proofs from $\widehat{\mathsf{DB}}$, to the client.

Upon receiving ans_1 and ans_2 from the servers, the client executes Recon, checking whether $\mathsf{st}_C.\mathsf{abort} =$ **true** and exiting early if yes. Otherwise, it verifies *all* the proofs output by the servers

```
$\mathsf{Setup}(\mathsf{sp}, \mathsf{DB}_{\mathsf{init}}) \rightarrow (\mathsf{st}_{S_b}, \mathsf{pp}_b)$
1 :  $(N_{\mathsf{init}}, Q_{\mathsf{init}}, M, \ell_r, \lambda) \leftarrow \mathsf{sp}$
2 :  Initialize $(\mathsf{DB}, \widehat{\mathsf{DB}}) \leftarrow 0^{2\cdot((Q_{\mathsf{init}}\cdot M)\times \ell_r)}$, $\mathsf{d} \leftarrow \perp$
3 :  $\mathsf{DB}[N_{\mathsf{init}}] \leftarrow \mathsf{DB}_{\mathsf{init}}$
4 :  for $q \in [Q_{\mathsf{init}}]$ do
5 :    $\mathsf{d.com}_q \leftarrow \mathsf{VC.Commit}(\mathsf{DB}_q)$
6 :    // Compute and store proofs
7 :    for $m \in [M]$ do
8 :      $\widehat{\mathsf{DB}}_q[m] \leftarrow \mathsf{VC.Open}(m, \mathsf{DB}_q)$
9 :  $\mathsf{pp}_b \leftarrow (M, \ell_r, \lambda, \mathsf{d})$, $\mathsf{pp}_b.N \leftarrow N_{\mathsf{init}}$,
     $\mathsf{pp}_b.Q \leftarrow Q_{\mathsf{init}}$
10 : return $((\mathsf{DB}, \widehat{\mathsf{DB}}, \mathsf{pp}_b), \mathsf{pp}_b)$
```

```
$\mathsf{RequestHint}(\mathbf{pp}) \rightarrow (\mathsf{st}_C, \mathbf{hq})$
1 :  $\mathsf{st}_C \leftarrow \emptyset$, $\mathsf{st}_C.\mathsf{abort} \leftarrow$ false
2 :  // Verify public parameters
3 :  if $\mathbf{pp}[1] \neq \mathbf{pp}[2]$ then
4 :    $\mathsf{st}_C.\mathsf{abort} \leftarrow$ true
5 :    return $(\mathsf{st}_C, \perp)$
6 :  $(\mathsf{hq}_1, \mathsf{hq}_2) \leftarrow (\text{“Stream DB, please!”})^2$
7 :  $\mathsf{st}_C.\mathsf{pp} \leftarrow \mathbf{pp}[1]$
8 :  return $(\mathsf{st}_C, (\mathsf{hq}_1, \mathsf{hq}_2))$
```

```
$\mathsf{AnsHintReq}(\mathsf{st}_{S_b}, \mathsf{hq}_b) \rightarrow (\mathsf{st}'_{S_b}, \mathsf{resp}_b)$
1 :  if $\mathsf{hq}_b = \perp$ then return $(\mathsf{st}_{S_b}, \perp)$
2 :  // Stream database
3 :  return $(\mathsf{st}_{S_b}, \mathsf{st}_{S_b}.\mathsf{DB})$
```

```
$\mathsf{VerSetup}(\mathsf{st}_C, \mathbf{resp}) \rightarrow (\mathsf{st}'_C, \mathsf{hint})$
1 :  // Verify streamed records
2 :  if $\mathsf{st}_C.\mathsf{abort} \vee \mathbf{resp}[1] \neq \mathbf{resp}[2]$ then
3 :    $\mathsf{st}_C.\mathsf{abort} \leftarrow$ true
4 :    return $(\mathsf{st}_C, \perp)$
5 :  $(\mathsf{ck}, \mathsf{hint}) \leftarrow \perp^2$
6 :  // Generate permutations
7 :  for $q \in [\mathsf{st}_C.\mathsf{pp}.Q]$ do
8 :    $\mathsf{ck}.\sigma_q \leftarrow\!\!\$\ \mathsf{Permute}(\mathsf{st}_C.\mathsf{pp}.M)$
9 :  $\mathsf{DB} \leftarrow \mathbf{resp}[1]$
10 : // Compute hint as in Singlepass
11 : for $m \in [\mathsf{st}_C.\mathsf{pp}.M]$ do
12 :   $\mathsf{hint}.h_m = \bigoplus_{q=1}^{\mathsf{st}_C.\mathsf{pp}.Q} \mathsf{DB}_q[\mathsf{ck}.\sigma_q(m)]$
13 : $\mathsf{st}_C.\mathsf{ck} \leftarrow \mathsf{ck}$, $\mathsf{st}_C.q* \leftarrow \perp$
14 : return $(\mathsf{st}_C, \mathsf{hint})$)
```

Figure 4: The offline phase of our two-server APIR scheme.

```
$\mathsf{Query}(\mathsf{st}_C, \mathsf{idx}) \rightarrow (\mathsf{st}'_C, \mathbf{qry})$
1 :  if $\mathsf{st}_C.\mathsf{abort} \vee \mathsf{st}_C.q* \neq \perp$ then return $(\mathsf{st}_C, \perp, \perp)$
2 :  $\mathsf{pp} \leftarrow \mathsf{st}_C.\mathsf{pp}$, $\{\sigma_1, ..., \sigma_{\mathsf{pp}.Q}\} \leftarrow \mathsf{st}_C.\mathsf{ck}$
3 :  $q^* \leftarrow \lceil \mathsf{idx}/\mathsf{pp}.M \rceil$
4 :  $m^* \leftarrow ((\mathsf{idx} - 1) \bmod \mathsf{pp}.M) + 1$
5 :  Find $\mathsf{ind} \in [\mathsf{pp}.M]$, s.t. $\sigma_{q^*}(\mathsf{ind}) = m^*$
6 :  $(r^*, r_1, ..., r_{\mathsf{pp}.Q}) \leftarrow\!\!\$\ [\mathsf{pp}.M]^{\mathsf{pp}.Q+1}$
7 :  $(\mathsf{qry}_1, \mathsf{qry}_2) \leftarrow \perp^2$
8 :  for $q \in [\mathsf{pp}.Q]$ do
9 :    $\mathsf{qry}_1.\mathsf{qry}_{1,q} \leftarrow \sigma_q(r_q)$
10 :   $\mathsf{qry}_2.\mathsf{qry}_{2,q} \leftarrow \sigma_q(\mathsf{ind})$
11 :   if $q \neq q^*$ then
12 :     Swap $\sigma_q(\mathsf{ind})$ and $\sigma_q(r_q)$
13 :   else $\mathsf{qry}_2.\mathsf{qry}_{2,q^*} \leftarrow r^*$
14 : $\mathsf{st}_C.\mathsf{ck} \leftarrow \{\sigma_1, ..., \sigma_{\mathsf{pp}.Q}\}$
15 : $\mathsf{st}_C \leftarrow (q^*, \mathsf{ind}, \mathsf{qry}_1, \mathsf{qry}_2)$
16 : return $(\mathsf{st}_C, (\mathsf{qry}_1, \mathsf{qry}_2))$
```

```
$\mathsf{Answer}(\mathsf{st}_{S_b}, \mathsf{qry}_b) \rightarrow (\mathsf{st}'_{S_b}, \mathsf{ans}_b)$
1 :  if $\mathsf{qry}_b = \perp$ then return $(\mathsf{st}_{S_b}, \perp)$
2 :  $\mathsf{DB} \leftarrow \mathsf{st}_{S_b}.\mathsf{DB}$, $\widehat{\mathsf{DB}} \leftarrow \mathsf{st}_{S_b}.\widehat{\mathsf{DB}}$
3 :  $Q \leftarrow \mathsf{st}_{S_b}.\mathsf{pp}.Q$
4 :  $(\mathsf{qry}_{b,1}, ...\mathsf{qry}_{b,Q}) \leftarrow \mathsf{qry}_b$
5 :  return $[\mathsf{DB}_q[\mathsf{qry}_{b,q}] \| \widehat{\mathsf{DB}}_q[\mathsf{qry}_{b,q}] \mid q \in [Q]]$
```

```
$\mathsf{Recon}(\mathsf{st}_C, \mathsf{hint}, \mathbf{ans}) \rightarrow (\mathsf{st}'_C, x, \mathsf{hint}')$
1 :  if $\mathsf{st}_C.\mathsf{abort} \vee st_C.q* = \perp$ then return $(\mathsf{st}_C, \perp, \perp)$
2 :  $\mathsf{pp} \leftarrow \mathsf{st}_C.\mathsf{pp}$, $\mathsf{ind} \leftarrow \mathsf{st}_C.\mathsf{ind}$, $q^* \leftarrow \mathsf{st}_C.q^*$
3 :  $(\mathsf{qry}_{1,1}, ..., \mathsf{qry}_{1,\mathsf{pp}.Q}) \leftarrow \mathsf{st}_C.\mathsf{qry}_1$
4 :  $(\mathsf{qry}_{2,1}, ..., \mathsf{qry}_{2,\mathsf{pp}.Q}) \leftarrow \mathsf{st}_C.\mathsf{qry}_2$
5 :  $((\mathsf{rec}_{1,1}||\pi_{1,1}), ..., (\mathsf{rec}_{1,\mathsf{pp}.Q}||\pi_{1,\mathsf{pp}.Q})) \leftarrow \mathbf{ans}[1]$
6 :  $((\mathsf{rec}_{2,1}||\pi_{2,1}), ..., (\mathsf{rec}_{2,\mathsf{pp}.Q}||\pi_{2,\mathsf{pp}.Q})) \leftarrow \mathbf{ans}[2]$
7 :  $(\mathsf{com}_1, ..., \mathsf{com}_{\mathsf{pp}.Q}) \leftarrow \mathsf{pp.d}$
8 :  $(h_1, ..., h_{\mathsf{pp}.M}) \leftarrow \mathsf{hint}$
9 :  // Verify commitments
10 : for $q \in [\mathsf{pp}.Q]$ do
11 :   for $b \in \{1, 2\}$ do
12 :     $v \leftarrow \mathsf{VC.Verify}(\mathsf{com}_q, \mathsf{qry}_{b,q}, \mathsf{rec}_{b,q}, \pi_{b,q})$
13 :     if $\neg v$ then
14 :       $\mathsf{st}_C.\mathsf{abort} \leftarrow$ true
15 :       return $(\mathsf{st}_C, \perp, \perp)$
16 : // Recover record
17 : $x \leftarrow \left( \bigoplus_{q \in [\mathsf{pp}.Q] \setminus \{q^*\}} \mathsf{rec}_{2,q} \right) \oplus h_{\mathsf{ind}}$
18 : for $q \in [\mathsf{pp}.Q] \setminus \{q^*\}$ do // Update hint
19 :   $h_{\mathsf{ind}} \leftarrow h_{\mathsf{ind}} \oplus \mathsf{rec}_{1,q} \oplus \mathsf{rec}_{2,q}$
20 :   $h_{r_q} \leftarrow h_{r_q} \oplus \mathsf{rec}_{1,q} \oplus \mathsf{rec}_{2,q}$
21 : $\mathsf{st}_C.\mathsf{ck} \leftarrow \{\sigma_q \mid q \in [\mathsf{pp}.Q]\}$, $\mathsf{st}_C.q* \leftarrow \perp$
22 : return $(\mathsf{st}_C, x, (h_1, ..., h_{\mathsf{pp}.M}))$
```

Figure 5: The online Phase of our two-server APIR scheme.

and, if any verification fails, aborts. Finally, if all checks pass, then the client reconstructs the requested record by combining the returned records with the locally-stored hint, exactly as in SinglePass. It updates the hint using the fresh randomness obtained from ans_1, ensuring consistency for future queries. Finally, the client updates its state with the updated permutations and outputs its state st_C, the record of interest x, and the updated hint.

Intuition for privacy. Observe that since r^* is uniformly sampled from $[M]$, the probability that the client directly queries for $\mathsf{DB}_{q^*}[m^*]$ (the record of interest) is just as likely as it querying for any other record in DB_{q^*} (see Fig. 4 Query line 13). To show that the set of indices in a query reveals no information to the server, we leverage the Show-and-Shuffle theorem introduced by [LP24] and restated in Appendix A.1. The queried record is correctly reconstructed if (1) the hint is correct up until the client runs Query and (2) the answers returned by the two servers are correct. As we show in §5.3, the hint is always correct; otherwise, the client aborts. Thus, it is sufficient to check the proofs returned in the answers. Importantly, since the requested records leak nothing about the queried index, aborting if the opening proofs in the answers are invalid, leaks nothing about the queried index. In contrast, if we were to only check the opening proof of the reconstructed record, then the servers could carry out a selective failure attack.

5.2 Supporting Updates

This section provides an overview of how Tapir can be extended to support appends (add) and edits (edit). We give the pseudocode for our updatable APIR scheme in Fig. 6. Observe that updates require the servers to update their database entries and generate/update the digests and commitments. The client must then update its state and hint accordingly.

5.2.1 Updating the Database. On input of server state $\mathsf{st}_{S_b} = (\mathsf{pp}_b, \mathsf{DB}, \widehat{\mathsf{DB}})$ and a sequence of updates U, Server b executes UpdateDB. Let $\mathsf{pp}' \leftarrow \mathsf{pp}_b$, $\mathsf{st}'_{S_b} \leftarrow \mathsf{st}_{S_b}$, and initialize set $U_b \leftarrow \emptyset$. The procedure iterates through the updates in U and assigns appropriate indices to all new records. This ensures that all operations are applied consistently, since add operations to an existing partition $q \in [\mathsf{pp}.Q]$ are treated equivalently to edit operations.

The procedure then computes the updated database parameters and stores them in pp', where the new number of partitions is $\mathsf{pp}'.Q = \lceil \mathsf{pp}'.N/M \rceil$ and $\mathsf{pp}.Q$ denotes the number of partitions before the update. Let U'_q denote the subset of updates corresponding to partition q. The algorithm iterates over each partition $q \in [\mathsf{pp}'.Q]$ and applies the associated update operations U'_q.

For each update $(\mathsf{op}, \mathsf{idx}, x_{\mathrm{new}}) \in U'_q$, where op denotes the operation, idx is the position m in the partition DB_q, x_{new} is the new value, and x_{old} is the old value, the following steps are performed:

- Update the database record: $\mathsf{DB}_q[m] \leftarrow x_{\mathrm{new}}$.
- Update the partition's vector commitment at index m to x_{new} if the partition is not new, i.e., if $q \le \mathsf{pp}.Q$.
- Append the update information $(\mathsf{op}, \mathsf{idx}, x_{\mathrm{new}} \oplus x_{\mathrm{old}})$ to U_b. Note that U_b stores the delta between the old and new values.

After all updates to a partition have been applied, if the partition is new (i.e., $q > \mathsf{pp}.Q$), then a fresh vector commitment com_q is computed for the partition and added to the digest $\mathsf{pp}'.\mathsf{d}$. Finally, UpdateDB refreshes the opening proofs in $\widehat{\mathsf{DB}}$ and outputs the updated server state st'_{S_b}, the updated public parameters pp', and the update information U_b.

5.2.2 Updating the Hint. Updates on the client side require three steps: (i) verifying the equality of the new public parameters—which contain the updated digests—and the update information returned by the servers, (ii) generating permutations for new partitions, and (iii) updating the hint using the verified update information and secret permutations.

UpdateHint takes as input the client state st_C, the current hint hint, the updated public parameters $\mathsf{pp}_1, \mathsf{pp}_2$, and the update information U_1, U_2 from the two servers. Step (i) verifies consistency of the new parameters and updates. If $\mathsf{pp}_1 \neq \mathsf{pp}_2$, $U_1 \neq U_2$, or $\mathsf{st}_C.\mathsf{abort} = \mathbf{true}$, the client sets $\mathsf{st}_C.\mathsf{abort} \leftarrow \mathbf{true}$ and returns $(\mathsf{st}_C, \perp)$. Since at least one server is semi-honest, equality guarantees correctness. Step (ii) generates permutations for any new

```
UpdateDB($\mathsf{st}_{S_b}, U$) → ($\mathsf{st}'_{S_b}, \mathsf{pp}'_b, U_b$)
$\mathsf{pp}' \leftarrow \mathsf{st}_{S_b}.\mathsf{pp}$, $\mathsf{st}'_{S_b} \leftarrow \mathsf{st}_{S_b}$, $Q_{old} \leftarrow \mathsf{pp}.Q$
$U', U_b \leftarrow \emptyset$
for $(\mathsf{op}, \mathsf{idx}, x) \in U$ do
    if op = add then // Add index to new records
        $\mathsf{pp}'.N = \mathsf{pp}'.N + 1$
        $U' \leftarrow U' \cup \{(\mathsf{op}, \mathsf{pp}'.N, x)\}$
    else $U' \leftarrow U' \cup \{(\mathsf{op}, \mathsf{idx}, x)\}$
$\mathsf{pp}'.Q \leftarrow \lceil \mathsf{pp}'.N/\mathsf{pp}'.M \rceil$ // Updated num. partitions
Extend $\mathsf{st}'_{S_b}.\mathsf{DB}$, $\mathsf{st}'_{S_b}.\widehat{\mathsf{DB}}$, $\mathsf{pp}'.\mathsf{d}$ to $\mathsf{pp}'.Q$ partitions/values
$(\mathsf{com}'_1, \ldots, \mathsf{com}'_{\mathsf{pp}'.Q}) \leftarrow \mathsf{pp}'.\mathsf{d}$
for $q \in [\mathsf{pp}'.Q]$ do
    // Get all update operations for partition $q$
    $U'_q \leftarrow \{(\mathsf{op}, \mathsf{idx}, x) \in U' : \mathsf{pp}'.M \cdot (q-1) < \mathsf{idx} \le \mathsf{pp}'.M \cdot q\}$
    for $(\mathsf{op}, \mathsf{idx}, x_{new}) \in U'_q$ do
        $m \leftarrow ((\mathsf{idx} - 1) \bmod \mathsf{pp}'.M) + 1$
        $x_{old} \leftarrow \mathsf{st}'_{S_b}.\mathsf{DB}_q[m]$
        $\mathsf{st}'_{S_b}.\mathsf{DB}_q[m] \leftarrow x_{new}$
        if $q \le Q_{old}$ then // Update partition commitment
            $\mathsf{com}'_q \leftarrow \mathsf{VC.Update}(\mathsf{com}'_q, m, x_{old}, x_{new})$
        $U'_q \leftarrow U'_q \cup \{(\mathsf{op}, \mathsf{idx}, x_{new} \oplus x_{old})\}$
    if $q > Q_{old}$ then // New partition commitment
        $\mathsf{com}'_q \leftarrow \mathsf{VC.Commit}(\mathsf{st}'_{S_b}.\mathsf{DB}_q)$
    for $m \in [\mathsf{pp}'.M]$ do // Update opening proofs.
        $\mathsf{st}'_{S_b}.\widehat{\mathsf{DB}}_q[m] \leftarrow \mathsf{VC.Open}(m, \mathsf{st}'_{S_b}.\mathsf{DB}_q)$
    $U_b \leftarrow U_b \cup U'_q$
$\mathsf{pp}'.\mathsf{d} \leftarrow (\mathsf{com}'_1, \ldots, \mathsf{com}'_{\mathsf{pp}'.Q})$, $\mathsf{st}'_{S_b}.\mathsf{pp} \leftarrow \mathsf{pp}'$
return $(\mathsf{st}'_{S_b}, \mathsf{pp}', U_b)$

UpdateHint($\mathsf{st}_C, \mathsf{hint}, \mathbf{pp}, \mathbf{U}$) → ($\mathsf{st}'_C, \mathsf{hint}'$)
if $\mathbf{pp}[1] \neq \mathbf{pp}[2] \vee \mathbf{U}[1] \neq \mathbf{U}[2]$ then // Verify update info
    $\mathsf{st}_C.\mathsf{abort} \leftarrow \mathbf{true}$
if $\mathsf{st}_C.\mathsf{abort} \vee \mathsf{st}_C.q* \neq \perp$   return $(\mathsf{st}_C, \perp)$
$Q' \leftarrow \mathsf{st}_C.\mathsf{pp}.Q$, $\mathsf{ck}' \leftarrow \mathsf{st}_C.\mathsf{ck}$
$(\mathsf{hint}_1, \ldots, \mathsf{hint}_{\mathbf{pp}[1].M}) \leftarrow \mathsf{hint}$
for $q \in [\mathbf{pp}[1].Q - \mathsf{st}_C.\mathsf{pp}.Q]$ // Generate new permutation(s)
    $Q' = Q' + 1$
    $\mathsf{ck}'.\sigma_q \leftarrow\$ \mathsf{Permute}(\mathbf{pp}[1].M)$
for $(\mathsf{op}, \mathsf{idx}, x) \in \mathbf{U}[1]$ do // Update hints
    $\hat{q} \leftarrow \lceil \mathsf{idx}/\mathbf{pp}[1].M \rceil$
    $\hat{m} \leftarrow ((\mathsf{idx} - 1) \bmod \mathbf{pp}[1].M + 1$
    $\mathsf{ind} \leftarrow \mathsf{ck}'.\sigma^{-1}_{\hat{q}}(\hat{m})$
    $\mathsf{hint}_{\mathsf{ind}} \leftarrow \mathsf{hint}_{\mathsf{ind}} \oplus x$
$\mathsf{st}_C.\mathsf{pp} \leftarrow \mathbf{pp}[1]$, $\mathsf{st}_C.\mathsf{ck} \leftarrow \mathsf{ck}'$
return $(\mathsf{st}_C, (\mathsf{hint}_1, \ldots, \mathsf{hint}_{\mathbf{pp}[1].M}))$
```

Figure 6: UpdateDB **and** UpdateHint **algorithm descriptions for Tapir.**

database partitions and appends them to the client key $st_C.ck$. Step (iii) updates the hint using the verified update information. For each $(\mathsf{op}, \mathsf{idx}, x) \in U_1$, the client uses its secret permutations $\{\sigma_q\}_{q \in [Q]}$ to locate the position in the hint. Let $(\hat{q}, \hat{m})$ be the partition and offset of idx. The client computes $\mathsf{ind} \leftarrow \sigma_{\hat{q}}^{-1}(\hat{m})$ and sets $\mathsf{hint}_{\mathsf{ind}} \leftarrow \mathsf{hint}_{\mathsf{ind}} \oplus x$. Finally, the client outputs the updated state st_C (including the new public parameters) and the updated hint.

5.3 Scheme Correctness and Security

Tapir requires $k = 2$ servers, one is assumed to be semi-honest. Both servers hold a copy of the same public database. Depending on the VC scheme used to instantiate our APIR scheme, we may also require that the public parameters are honestly generated (e.g., through a trusted setup, as in Pointproofs [GRWZ20]).

Theorem 5.3.1. Let VC be a vector commitment scheme with binding such that VC.Commit is deterministic. The updateable two-server APIR scheme with preprocessing described in Figs. 4, 5, and 6 satisfies correctness (Def. 4).

Theorem 5.3.2. Let $\lambda \in \mathbb{N}$ be the security parameter, $N_{\mathsf{init}}, N = \mathsf{poly}(\lambda)$, and VC be a secure vector commitment scheme with binding such that VC.Commit is deterministic. If both servers are non-colluding, then the updateable two-server APIR scheme with preprocessing in Figs. 4, 5, and 6 satisfies integrity (Def. 5).

Theorem 5.3.3. Let $\lambda \in \mathbb{N}$ be the security parameter, $N_{\mathsf{init}}, N = \mathsf{poly}(\lambda)$, and VC be a secure vector commitment scheme with binding such that VC.Commit is deterministic. If both servers are non-colluding and integrity holds, then the updateable two-server APIR scheme with preprocessing in Figs. 4, 5, and 6 satisfies privacy with abort (Def. 6).

The proofs are in Appendices B.1, B.2, and B.3, respectively.

6 Evaluation

This section evaluates the performance of Tapir and compares it to related work. We implement our scheme with both Pointproofs [GRWZ20] and Merkle trees as the vector commitment (VC) scheme (we refer to these implementations as TAPIR-PP and TAPIR-MT, respectively). We compare our approach against the two-server APIR schemes of [CNC+23], specifically the DPF-based APIR scheme (denoted as APIR-DPF) and the linear APIR scheme implemented with both Pointproofs and Merkle trees (which we denote as APIR-Matrix-PP and APIR-Matrix-MT, respectively). Furthermore, we compare our scheme to SinglePass [LP24] to demonstrate the overhead incurred by making it maliciously secure.

We aim to answer the following questions:

1. How much communication and computation overhead does Tapir have over its base protocol SinglePass? (§6.2)
2. How does Tapir compare to related multi-server APIR schemes, especially in the online phase? (§6.3)
3. What are the amortized communication and computation costs of batch updates? (§6.4)

N	(A)PIR	Offline			Online	
		BW [kiB]	RT [s] (1-Time)	RT [s] (Per-Client)	BW [kiB]	RT [s]
2^{18}	APIR-DPF	-	-	-	1.43	0.04
	APIR-Matrix-MT	0.10	0.58	0.00	49.68	0.01
	TAPIR-MT	18 982.05	0.33	0.08	354.00	0.01
	TAPIR-PP	19 025.02	13 162.05	0.10	34.66	0.99
	SinglePass	18.52	0.00	0.01	34.50	0.00
2^{20}	APIR-DPF	-	-	-	1.56	0.17
	APIR-Matrix-MT	0.10	2.74	0.00	104.36	0.04
	TAPIR-MT	75 852.05	1.45	0.39	776.98	0.02
	TAPIR-PP	75 938.00	105 304.90	0.46	69.65	2.63
	SinglePass	37.02	0.00	0.05	69.50	0.00
2^{22}	APIR-DPF	-	-	-	1.69	0.72
	APIR-Matrix-MT	0.10	11.54	0.00	218.26	0.16
	TAPIR-MT	303 256.05	5.51	1.86	1 690.98	0.05
	TAPIR-PP	303 427.92	844 441.96	1.63	139.66	8.79
	SinglePass	74.03	0.00	0.24	139.51	0.00
2^{24}	APIR-DPF	-	-	-	1.83	2.97
	APIR-Matrix-MT	0.10	47.73	0.00	454.87	0.73
	TAPIR-MT	1 212 720.05	22.14	7.65	3 662.97	0.12
	SinglePass	148.03	0.00	0.96	279.51	0.01

Table 1: Offline and online performance comparison of (A)PIR schemes for record size 32 B. Offline costs are split into one-time server costs and per-client costs. Best APIR performance for each database size N is highlighted. Results for APIR-Matrix-PP and TAPIR-PP ($N > 2^{22}$) are omitted due to high Pointproofs setup costs.

Our protocol is implemented in Go, utilizing and adapting existing libraries for Merkle trees[1], and Pointproofs[2]. For our benchmarks, we extended the (A)PIR implementations of related work, namely, SinglePass[3] and the linear matrix PIR[4], and implemented the APIR-DPF scheme[5].

For TAPIR-PP, we reduce per-query bandwidth and verification cost by aggregating Pointproofs across database partitions, thereby lowering both online bandwidth and runtime. A single Pointproofs proof requires two exponentiations and two pairings, whereas aggregating across ℓ vectors requires only $1 + \ell$ of each—saving one exponentiation and one pairing per vector. Our implementation performs aggregation in Answer, shifting part of the verification cost to the server and allowing it to send a single proof instead of Q. In contrast, APIR-Matrix-PP sends only one proof per query and thus does not benefit from aggregation.

Our source code is available at https://github.com/laurahetz/TAPIR.

6.1 Experimental Setup

All benchmarks are executed on a single machine with two Intel Xeon Gold 6258R CPU 2.7 GHz, each with 28 cores, and 384 GB of DDR4 memory.

For Tapir and related work, we use initial database sizes of $N \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}, 2^{24}\}$ database record 32 B, and where applicable Q database partitions of size $M = \sqrt{N}$.

[1]https://github.com/dedis/apir-code
[2]https://github.com/yacovm/PoL/tree/main/pp
[3]https://github.com/SinglePass712/Submission
[4]https://github.com/dimakogan/checklist
[5]https://github.com/osu-crypto/libOTe

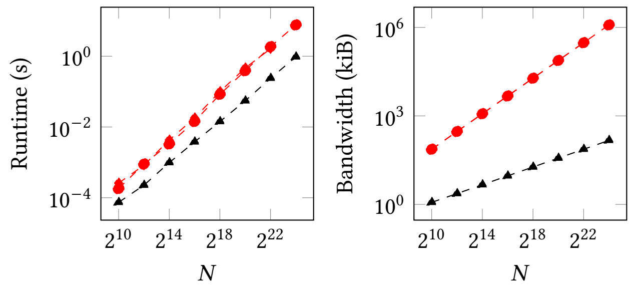


Figure 7: Offline per-client performance comparison between our authenticated scheme Tapir and the unauthenticated base scheme SinglePass. Results shown for databases with N records of size 32 B. All axes use logarithmic scaling. We denote the schemes as follows: TAPIR-MT (- ● -), TAPIR-PP (- ◆ -), and SinglePass (- ▲ -).

Due to the higher one-time runtime cost of Pointproofs in the offline phase, we were unable to run APIR-Matrix-PP and TAPIR-PP for databases larger than 2^{16} and 2^{22}, respectively. All measurements of the online phase are averaged over 25 runs.

6.2 Comparison to SinglePass

Since Tapir modifies and extends SinglePass to achieve malicious security, we expect Tapir to have higher runtime and communication cost in all protocol phases. In the offline phase, the increase in cost stems from the servers' digest generation and the need to stream the database from both servers to ensure the integrity of the preprocessed information. In the online phase, the increase in bandwidth stems from the retrieved records' proofs that are sent together with the records, and the increase in runtime results from the client having to verify these proofs.

Table 1 reports the benchmarking results for the total offline and online costs for all considered (A)PIR schemes. Figure 7 highlights the per-client bandwidth and runtime of Tapir compared to SinglePass. This excludes the digest generation, as it is a one-time setup cost and is reusable for all clients.

As expected, Tapir requires significantly more bandwidth overhead in the offline phase due to the digest and the client's need to authenticate the database records included in the hint. The client-dependent offline runtime of Tapir does not depend on the selected vector commitment scheme and is thus the same for TAPIR-MT and TAPIR-PP. Compared to SinglePass, this offline runtime is up to 8× higher for $N = 2^{24}$ due to the setup verification.

Figure 8 shows the online costs of both SinglePass and Tapir as compared to three related APIR schemes. Once again, the experimental results are consistent with our expectations.

Our scheme demonstrates very modest bandwidth increases in the online phase with respect to SinglePass, with as little as 0.11 % overhead for TAPIR-PP for $N = 2^{22}$ and at most 13.11× overhead for TAPIR-MT for $N = 2^{24}$. We see that TAPIR-PP significantly outperforms TAPIR-MT as Pointproofs allow optimizing bandwidth and verification cost via proof aggregation. This optimization results in communication costs comparable to SinglePass. While query costs are the same across all three schemes, the additional overhead of our scheme comes from the proofs returned as part of the answer. A Pointproofs proof is compact and only comprises a single group element, whereas each Merkle tree proof comprises an authentication path of size $O(\log_2 M)$.

Both Tapir variants show higher online runtimes than SinglePass, with TAPIR-MT outperforming TAPIR-PP. The higher online cost of TAPIR-PP comes from generating and verifying the aggregated proof using group exponentiations and a pairing product. In contrast, proof verification of TAPIR-MT only requires $Q \log_2 M$ hash evaluations by the client, which are cheap.

The online runtime cost of TAPIR-MT (for $N = 2^{24}$) is overall 22.64× higher than SinglePass, with only 1.37× overhead for the server and 54.15× overhead for the client. The online runtime cost of TAPIR-PP (for $N = 2^{22}$) is overall 5.8 magnitudes higher than SinglePass. The majority of the overhead comes from the Pointproofs aggregation in the Answer algorithm and verification in the Recon algorithm, as observed in Fig. 8. We discuss the trade-off between vector commitment schemes in more detail in §7.

6.3 Comparison to Related Work

This section compares the benchmarking results of Tapir (TAPIR-MT and TAPIR-PP) to prior multi-server APIR schemes, namely APIR-DPF, APIR-Matrix-MT [CNC+23], and APIR-Matrix-PP.

Using a linear PIR scheme with a vector commitment scheme, as done in APIR-Matrix-MT and APIR-Matrix-PP, requires each server to generate a digest, which the client verifies as part of the offline phase. Tapir also relies on a vector commitment scheme for integrity in the online phase and thus also incurs the cost of digest generation and communication, as well as proof generation, verification, and communication. Since Tapir executes the vector commitment schemes over each of the Q database partitions instead of the entire database (as done in APIR-Matrix-MT and APIR-Matrix-PP), the digest generation in Tapir is expected to be faster in comparison. In addition, in Tapir the client and servers preprocess the database to ensure integrity of the client's retrieved hint. We expect this hint computation and verification to cause significant overhead in the offline phase of Tapir. As APIR-DPF does not require any preprocessing, we expect all other considered APIR schemes to have a higher offline overhead. APIR-Matrix-MT and APIR-Matrix-PP overall have very low client offline overhead as only the digests of the servers need to be checked for equality. In the online phase, we expect TAPIR-MT to have lower runtimes than all other APIR schemes for larger database sizes, facilitated by the preprocessing and fast verification of proofs.

Table 1 reports the benchmarking results, and Fig. 8 visualizes the online costs of the benchmarked schemes.

With respect to online bandwidth, we see that the schemes instantiated with Merkle trees (TAPIR-MT, APIR-Matrix-MT) are more expensive than their Pointproofs counterparts due to the logarithmically sized proofs of the Merkle trees. We note that communication in the APIR-MATRIX schemes is lower as only secret shares of one record and its proof are communicated, while Q records and proofs are required in Tapir. APIR-DPF has the lowest bandwidth of all considered (A)PIR schemes, at the cost of server runtime linear in the size of the database.

Regarding client runtime, APIR-DPF is even faster than the unauthenticated SinglePass while APIR-Matrix-MT and APIR-Matrix-PP

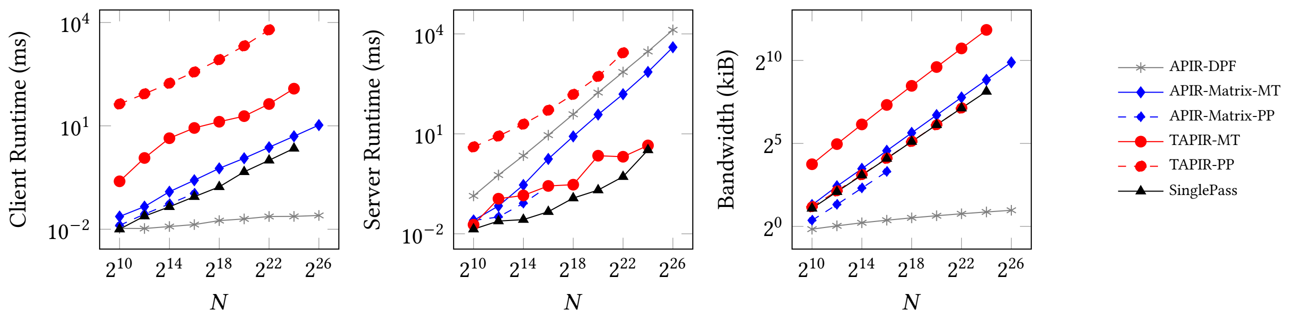


Figure 8: Online performance comparison of our scheme Tapir and prior work. Results shown for databases with N records of size 32 B. All axes use logarithmic scaling.

have similar cost, and Tapir incurs the highest overhead. This is expected, as the Recon algorithm of Tapir is more complex and computationally intensive compared to the cheap XOR operations in the APIR-Matrix schemes. Regarding server runtime, TAPIR-MT outperforms all APIR schemes with only 1.37× overhead compared to its unauthenticated base scheme. As expected, we observe a linear trend for the server runtime in APIR-DPF and the APIR-Matrix schemes, whereas Tapir has sublinear runtime in the database size.

For $N \geq 2^{20}$, TAPIR-MT is the fastest APIR scheme online, outperforming APIR-Matrix-MT and APIR-DPF by a factor of up to 5.83× and 23.82×, respectively.

6.4 Database Updates

This section evaluates the practicality of database updates in Tapir. For this, we measured the bandwidth and runtime cost of applying a batch of 500 update operations to unique database indices for databases with N 32 B records. We differentiate between batches of only additions (ADD), only edits (EDIT), and BOTH additions and edits, and report the amortized costs in Fig. 9.

We expect edits and additions within existing partitions to be equally fast, as they are handled the same way. Client runtime and bandwidth are expected to be independent of the chosen VC scheme. In contrast, we expect the server runtime to be higher for Pointproofs than for Merkle trees, due to the higher cost of generating and updating commitments and proofs. We expect bandwidth costs to be similar over all update types, with the exception that new partitions require communicating new digests. These only occur with ADD and BOTH, while BOTH has, on average, only half the additions of ADD, and thus will incur less bandwidth.

We observe that the client runtime is very fast, as updates only require lightweight operations such as generating new permutations and performing basic arithmetic (e.g., XOR, mod, division) to update the hint. For instance, the largest observed amortized client runtime is just 0.005 ms at $N = 2^{22}$ for TAPIR-MT. TAPIR-PP incurs slightly higher cost than TAPIR-MT, due to the equality check of the public parameters containing bilinear group elements as opposed to hash function outputs.

The server runtime is higher but remains practical, especially since update computations are client-independent. Amortized runtimes range from 0.005 ms ($N = 2^{10}$, TAPIR-MT) to 370.5 ms ($N = 2^{14}$, TAPIR-PP). TAPIR-PP is generally slower due to the costly bilinear pairings required for Pointproofs commitments. EDIT operations also tend to be more expensive than ADD, as each of the 500 edits may require updating commitments for up to 500 existing partitions, whereas additions only require initializing and committing to a few new partitions.

Bandwidth costs are modest, ranging from 83.48 B ($N = 2^{10}$, TAPIR-MT) to 86.01 B ($N = 2^{22}$, TAPIR-MT). The bandwidth of TAPIR-MT and TAPIR-PP is comparable across ADD, EDIT, and BOTH operations, since the client only receives the update tuples (identical for both schemes) and the new public parameters with the updated digest. The digests differ slightly in form—TAPIR-MT uses the Merkle tree root and TAPIR-PP bilinear group element—but both are small and similar in size. Among the three operations, ADD incurs the highest bandwidth due to the transmission of new vector commitments for added partitions. EDIT incurs the least since no new partitions are created, and BOTH falls in between as half of the operations are additions on average. A noticeable bandwidth gap emerges around $N = 2^{16}$ due to partitioning: with $M = 2^8 = 256$, ADD requires approximately two new partitions per batch, EDIT requires none, and BOTH requires about one.

6.5 Application: Key Transparency

Systems for Key Transparency (KT) are widely deployed by companies such as Google [Goo24], WhatsApp [LL23], Proton [GH24], and Keybase [key22] to enable users to verify public keys for end-to-end encrypted communication, and are even undergoing standardization [Gro25]. Typically, KT schemes use an authenticated dictionary hosted by an untrusted server that maps user IDs to public keys. A digest of the dictionary is computed, allowing clients to perform verifiable lookups.

Because KT requires integrity for public key lookups, it presents a promising application for APIR. For example, KT can be instantiated by extending index-based APIR to support keyword search (see [CD24, HLP$^+$25] and § 7). Alternatively, Index-APIR can be used to index public keys directly; clients can then obtain the index of a public key for lookups (e.g., via gossiping, an auditor, or a previous lookup). Standard deployed KT systems often overlook a critical aspect: query privacy for users requesting another user's key. APIR fills this gap by providing query privacy with abort even against adversarial servers.

Both WhatsApp's [Met23] and Proton's [GH24] white papers state that their KT systems use Curve25519 as the Public Identity

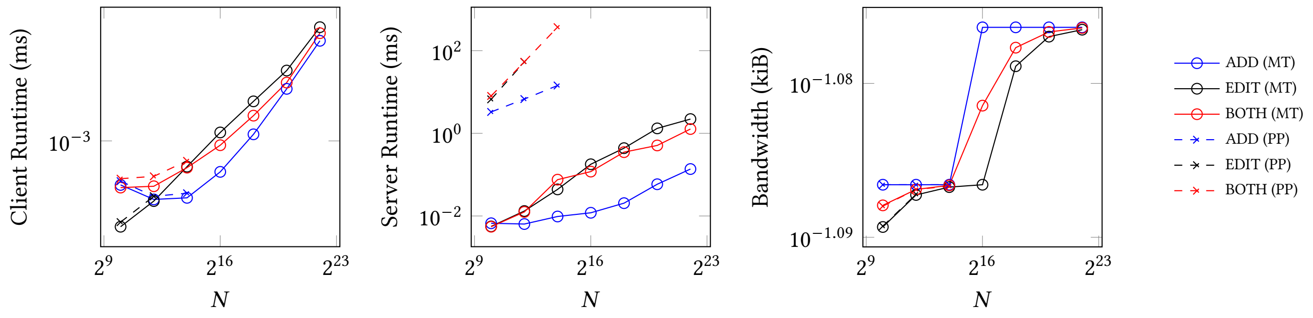


Figure 9: Amortized update costs for Tapir with Merkle trees (MT) and Pointproofs (PP), comparing ADD, EDIT, and BOTH operations. Results are for databases with N records of 32 B sizes and update batches of size 500. Axes use logarithmic scaling.

Key, meaning keys stored in the dictionary are 32 B long. In our experiments (see, e.g., Fig. 8), we demonstrate the performance of our schemes TAPIR-MT and TAPIR-PP on 32 B records to reflect this use case. In particular, TAPIR-MT scaled easily to $N = 2^{24}$ and can be extended to larger databases common in KT by leveraging database partitioning as described in § 7. Moreover, the efficient client and server runtimes, along with the small bandwidth required for updates of a batch size of 500 (as discussed in § 6.4), further bolster the practicality of our scheme for transparency.

7 Conclusion

Optimizations. The complexity of Tapir grows with the database size N. To improve scalability, the database can be partitioned into p sub-databases [HSW23], with Tapir running on each in parallel. This adds minor overhead but reduces per-database cost and enables trivial parallelization of both phases. We leave this for future work.

Support for Keywords. While (A)PIR usually assumes knowledge of a record's index, many applications rely on keywords. Our scheme can be extended in a black-box way to support keyword queries [CD24, HLP+25]. This incurs extra bandwidth and computation, but allows retrieval based on labels rather than indices.

Trade-offs and Use Cases. Tapir supports a flexible runtime and bandwidth trade-off (Fig. 5 and Tab. 1). For example, using Merkle trees yields the smallest runtimes among multi-server APIR schemes, while using Pointproofs [GRWZ20] matches the online bandwidth of SinglePass. In contrast, linear schemes like [CNC+23] show little variation across different VCs. Thus, bandwidth-sensitive applications with frequent queries (e.g., key transparency, oblivious message detection) benefit from TAPIR-PP, while infrequent-query settings (e.g., contact discovery, medical look-up) favor TAPIR-MT.

Acknowledgments

The authors thank Tianxin Tang, Christian Weinert, Matilda Backendal for their helpful feedback and input. The authors would also like to thank Shannon Veitch for her suggestion to adopt a game-based security definition. Francesca Falzon is supported by Armasuisse Science and Technology.

References

[AB25] Bar Alon and Amos Beimel. On the definition of malicious private information retrieval. In *The sixth Information-Theoretic Cryptography (ITC)*, 2025.

[ABG+24] Hilal Asi, Fabian Boemer, Nicholas Genise, Muhammad Haris Mughees, Tabitha Ogilvie, Rehan Rishi, Guy N. Rothblum, Kunal Talwar, Karl Tarbe, Ruiyu Zhu, and Marco Zuliani. Scalable Private Search with Wally. *CoRR*, abs/2406.06761, 2024.

[AR25] Pranav Shriram Arunachalaramanan and Ling Ren. Single-server stateful PIR with verifiability and balanced efficiency. Cryptology ePrint Archive, Paper 2025/1055, 2025.

[AS16] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, pages 551–569. USENIX Association, 2016.

[BGH+25] Elette Boyle, Niv Gilboa, Matan Hamilis, Yuval Ishai, and Yaxin Tu. Improved Constructions for Distributed Multi-Point Functions . In *2025 IEEE Symposium on Security and Privacy*, pages 2414–2432. IEEE Computer Society Press, May 2025.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Berlin, Heidelberg, April 2015.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[CD24] Sofía Celi and Alex Davidson. Call me by my name: Simple, practical private information retrieval for keyword queries. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 4107–4121. ACM Press, October 2024.

[CDGM19] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. SEEMless: Secure end-to-end encrypted messaging with less trust. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1639–1656. ACM Press, November 2019.

[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.

[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.

[CHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 3–33. Springer, Cham, May / June 2022.

[CK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 44–75. Springer, Cham, May 2020.

[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.

[CSM+20] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas E. Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private group messaging with hidden access patterns. In *ACSAC*, pages 84–99. ACM, 2020.

[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.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Berlin, Heidelberg, August 2012.

[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.

[Dur64] Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, July 1964.

[EKN22] Reo Eriguchi, Kaoru Kurosawa, and Koji Nuida. On the optimal communication complexity of error-correcting multi-server PIR. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part III*, volume 13749 of *LNCS*, pages 60–88. Springer, Cham, November 2022.

[FMS24] Brett Falk, Pratyush Mishra, and Matan Shtepel. Malicious security for PIR (almost) for free. Cryptology ePrint Archive, Report 2024/964, 2024.

[FMS25] Brett Hemenway Falk, Pratyush Mishra, and Matan Shtepel. Malicious security for PIR (almost) for free. In *CRYPTO 2025, Part I*, LNCS. Springer, Cham, August 2025.

[FY53] Ronald Aylmer Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research*. Hafner Publishing Company, 1953.

[GH24] Thore Göbel and Daniel Huigens. Proton key transparency whitepaper. https://proton.me/files/proton_keytransparency_whitepaper.pdf, 2024.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Berlin, Heidelberg, May 2014.

[Goo24] Inc. Google. Key transparency. https://github.com/google/keytransparency, 2024. Accessed: 12/08/2025.

[Gro25] KT Internet Engineering Task Force Working Group. Key transparency (keytrans). https://datatracker.ietf.org/wg/keytrans/about/, 2025.

[GRWZ20] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 2007–2023. ACM Press, November 2020.

[GZS24] Ashrujit Ghoshal, Mingxun Zhou, and Elaine Shi. Efficient pre-processing PIR without public-key cryptography. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part VI*, volume 14656 of *LNCS*, pages 210–240. Springer, Cham, May 2024.

[GZSP25] Ashrujit Ghoshal, Mingxun Zhou, Elaine Shi, and Bo Peng. Pseudorandom functions with weak programming privacy and applications to private information retrieval. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part VII*, volume 15607 of *LNCS*, pages 284–313. Springer, Cham, May 2025.

[HDCZ23] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private Web Search with Tiptoe. In *SOSP*, pages 396–416. ACM, 2023.

[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.

[HHK+21] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle^2: A low-latency transparency log system. In *2021 IEEE Symposium on Security and Privacy*, pages 285–303. IEEE Computer Society Press, May 2021.

[HLP+25] Meng Hao, Weiran Liu, Liqiang Peng, Cong Zhang, Pengfei Wu, Lei Zhang, Hongwei Li, and Robert H. Deng. Practical keyword private information retrieval from key-to-index mappings. In *USENIX Security 2025*. USENIX Association, August 2025.

[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.

[KC21] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 875–892. USENIX Association, August 2021.

[key22] keybase.io. Keybase chat. book.keybase.io/docs/chat, 2022. Accessed: 12/08/2025.

[KKEPR24] Erki Külaots, Toomas Krips, Hendrik Eerikson, and Pille Pullonen-Raudvere. SLAMP-FSS: Two-party multi-point function secret sharing from simple linear algebra. Cryptology ePrint Archive, Report 2024/1394, 2024.

[Knu97] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[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.

[LL23] Kevin Lewi and Sean Lawlor. Introducing auditable key transparency for end-to-end encrypted messaging. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/, April 2023. Meta Engineering Blog.

[LP23] Arthur Lazzaretti and Charalampos Papamanthou. TreePIR: Sublinear-time and polylog-bandwidth private information retrieval from DDH. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part II*, volume 14082 of *LNCS*, pages 284–314. Springer, Cham, August 2023.

[LP24] Arthur Lazzaretti and Charalampos Papamanthou. Single pass client-preprocessing private information retrieval. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

[LTW24] Zeyu Liu, Eran Tromer, and Yunhao Wang. PerfOMR: Oblivious message retrieval with reduced communication and computation. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

[MBB+15] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing key transparency to end users. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 383–398. USENIX Association, August 2015.

[Met23] Meta. Whatsapp key transparency overview. https://www.whatsapp.com/security/WhatsApp-Key-Transparency-Whitepaper.pdf, 2023.

[MKKS+23] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean F. Lawlor. Parakeet: Practical key transparency for end-to-end encrypted messaging. In *NDSS 2023*. The Internet Society, February 2023.

[Res24] Apple Machine Learning Research. Combining machine learning and homomorphic encryption in the apple ecosystem. https://machinelearning.apple.com/research/homomorphic-encryption, oct 2024. Accessed: 2025-07-14.

[TBP+19] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1299–1316. ACM Press, November 2019.

[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.

[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.

A Additional Preliminaries

A.1 Show-and-Shuffle

In SinglePass [LP24], the Show-and-Shuffle game $\mathbf{G}^{\mathsf{Ind\text{-}SaS}}$ was introduced to capture the main steps of a single protocol round and to prove its privacy. Since Tapir extends SinglePass to achieve malicious security, we can leverage this game to also prove privacy for our scheme. Hence, we restate the Show-and-Shuffle game in Fig. 10 and the Show-and-Shuffle Indistinguishability lemma in Theorem A.1.1, and refer to [LP24, § 3.1] for the full proof. The game is parameterized over $M, Q \in \mathbb{N}$.

Lemma A.1.1. For the Show-and-Shuffle game $\mathbf{G}^{\mathsf{Ind\text{-}SaS}}$ (Fig. 10), and any $M, Q \in \mathbb{N}$, the advantage of any adversary playing this game is defined as:

$$\mathsf{Adv}^{\mathsf{ind-sas}}_{\mathcal{A}}(M, Q) = \left| \Pr\left[\mathbf{G}^{\mathsf{Ind\text{-}SaS}}(M, Q)\right] - \frac{1}{2} \right| = 0.$$

B Additional Proof Content

We now give the proofs for Theorem 5.3.1 (Correctness), Theorem 5.3.2 (Integrity), and Theorem 5.3.3 (Privacy).

```
Game $\mathbf{G}^{\text{Ind-SaS}}(M, Q)$
1 :  $(\sigma_1, ..., \sigma_Q) \leftarrow\$ \mathsf{Permute}(M)^Q$
2 :  $(\mathsf{st}_{\mathcal{A}}, x) \leftarrow \mathcal{A}_0(M, Q)$ where $x = (q^*, m^*) \in ([Q] \times [M])$
3 :  Find $\mathsf{ind} \in [M]$, s.t. $\sigma_{q^*}(\mathsf{ind}) = m^*$
4 :  $\mathbf{v} \leftarrow \perp$
5 :  for $q \in [Q]$ do
6 :    if $q \neq q^*$ then $\mathbf{v}.v_q \leftarrow \sigma_q(\mathsf{ind})$
7 :    else $\mathbf{v}.v_q \leftarrow\$ [M]$
8 :  $b \leftarrow\$ \{0, 1\}$
9 :  $\mathcal{R}_0 = (F_1, ..., F_Q) \leftarrow\$ \mathsf{Permute}(M)^Q$
10 : for $q \in [Q] \wedge q \neq q^*$ do
11 :   $r_q \leftarrow\$ [M]$
12 :   $\sigma'_q \leftarrow \sigma_q$
13 :   Swap $\sigma'_q(\mathsf{ind})$ and $\sigma'_q(r_q)$
14 : $\mathcal{R}_1 \leftarrow (\sigma'_1, ..., \sigma'_{q^*-1}, \sigma_{q^*}, \sigma'_{q^*+1}, ...\sigma'_Q)$
15 : $b' \leftarrow \mathcal{A}_1(\mathsf{st}_{\mathcal{A}}, q, m, \mathbf{v}, \mathcal{R}_b)$
16 : return $b = b'$
```

Figure 10: The Show-and-Shuffle game of SinglePass.

B.1 Proof of Correctness (Theorem 5.3.1)

Proof. We note that correctness is only defined for honest servers. Let $\mathsf{DB}_{\mathsf{init}}$ be the initial database records generated by $\mathcal{A}_0$. Let $\sigma_q : [\mathsf{pp}.M] \to [\mathsf{pp}.M]$ denote the permutation for partition $q \in [\mathsf{pp}.Q_{\mathsf{init}}]$. By correctness of the vector commitment scheme VC and since VC.Commit is assumed to be deterministic, for each $q \in [\mathsf{pp}.Q_{\mathsf{init}}]$ we have

$$\mathsf{com}_{1,q} = \mathsf{com}_{2,q} = \mathsf{VC.Commit}(\mathsf{DB}_q).$$

In the offline phase, the digests satisfy

$$\mathsf{d}_1 = (\mathsf{com}_{1,1}, ..., \mathsf{com}_{1,Q_{\mathsf{init}}}) = (\mathsf{com}_{2,1}, ..., \mathsf{com}_{2,Q_{\mathsf{init}}}) = \mathsf{d}_2.$$

Since the remaining public parameters are derived directly from the setup parameters provided to Setup, both servers output identical values. We thus have $\mathbf{pp}[1] = \mathbf{pp}[2]$ (Fig. 4, line 3), and RequestHint completes successfully.

Lookup correctness. We now prove that the initial lookup query succeeds, and then proceed by induction over the sequence of T_{q} lookups. Since Server b sets $\mathsf{st}_{S_b}.\mathsf{DB}[[N_{\mathsf{init}}]] \leftarrow \mathsf{DB}_{\mathsf{init}}$ (Fig. 4, line 3) and, by correctness of the streaming procedure (Fig. 4, line 3), we obtain $\mathsf{DB}_1 = \mathsf{DB}_2$. It follows that $\mathsf{resp}_1 = \mathsf{resp}_2$ (Fig. 4, line 15). Thus, the VerSetup algorithm computes a hint (Fig. 4, line 12) of the form $\mathsf{hint} = (h_1, ..., h_M)$ where, for each $m \in [M]$:

$$h_m = \bigoplus_{q \in [Q_{\mathsf{init}}]} \mathsf{DB}[(q-1) \cdot M + \sigma_q(m)] = \bigoplus_{q \in [Q_{\mathsf{init}}]} \mathsf{DB}_q[\sigma_q(m)].$$

Let $\mathsf{idx} \in [N]$ be any index queried for by the client and $(\hat{q}, \hat{m}) \in [Q_{\mathsf{init}}] \times [M]$ be such that $\mathsf{DB}[\mathsf{idx}] = \mathsf{DB}_{\hat{q}}[\hat{m}]$. Let $\mathsf{ind} \in [M]$ be such that $\sigma_{\hat{q}}(\mathsf{ind}) = \hat{m}$.

Consider that Server 2 responds to the first query honestly. Applying the above hint equation to line 17 (Fig. 5) yields

$$\begin{aligned} x &= \bigoplus_{q \in [Q_{\mathsf{init}}] \setminus \{\hat{q}\}} \mathsf{rec}_{2,q} \oplus h_{\mathsf{ind}} \\ &= \left(\bigoplus_{q \in [Q_{\mathsf{init}}] \setminus \{\hat{q}\}} \mathsf{DB}_q[\sigma_q(\mathsf{ind})] \right) \oplus h_{\mathsf{ind}} \\ &= \left(\bigoplus_{\substack{q \in [Q_{\mathsf{init}}] \\ q \neq \hat{q}}} \mathsf{DB}_q[\sigma_q(\mathsf{ind})] \right) \oplus \left(\bigoplus_{q \in [Q_{\mathsf{init}}]} \mathsf{DB}_q[\sigma_q(\mathsf{ind})] \right) \\ &= \mathsf{DB}_{\hat{q}}[\sigma_{\hat{q}}(\mathsf{ind})] = \mathsf{DB}_{\hat{q}}[\hat{m}] = \mathsf{DB}[\mathsf{idx}]. \end{aligned}$$

By correctness of VC, and since the servers are assumed to be honest and return the correct record corresponding to qry_b, it follows that for every $q \in [Q_{\mathsf{init}}]$ and every $b \in \{1, 2\}$,

$$1 = \mathsf{VC.Verify}(\mathsf{com}_q, \mathsf{qry}_{b,q}, \mathsf{rec}_{b,q}, \pi_{b,q})$$

where $\mathsf{ans}_{b,q} = (\mathsf{rec}_{b,q} \| \pi_{b,q})$. Thus, the client reconstructs and outputs $x = \mathsf{DB}[\mathsf{idx}]$ and the win is set to **true** with probability 0.

We now show that subsequent lookup queries correctly recover the requested record and that the invariant, that the hint remains correct after every step, is preserved. In particular, we want to show that for every $m \in [M]$, the following invariant holds:

$$h_m = \bigoplus_{q \in [\mathsf{pp}.Q]} \mathsf{DB}[\sigma_q(m)]. \tag{1}$$

Assume that $\mathsf{hint} = (h_1, ..., h_M)$ is correct after responding to the k-th query and let $\mathsf{pp}.Q$, $\mathsf{pp}.M$, and $\mathsf{pp}.N$ denote the current public parameters at this step.

We appeal to the argument of correctness in [LP24] and reproduce the details for completeness below. Note that for each swap between $\sigma_q(\mathsf{ind})$ and $\sigma_q(r_q)$ the hint is modified such that

$$h_{\mathsf{ind}} \leftarrow h_{\mathsf{ind}} \oplus \mathsf{DB}_q[\sigma_q(\mathsf{ind})] \oplus \mathsf{DB}_q[\sigma_q(r_q)].$$

This XOR removes the old element in the hint and adds the new one. Thus at the end of the $(k+1)$-the query, $h_m = \oplus_{q \in [\mathsf{pp}.Q]} \mathsf{DB}_q[\sigma_q(m)]$ for all $m \in [\mathsf{pp}.M]$. Since the hint is still of the correct form (Eq. 1), then, by an argument similar to the one above, the client must always recover the correct database element using the hint and the win is set to **true** with probability 0.

Update correctness. We have already shown that the hint is computed correctly at setup and after any number of T_{q} lookup queries. Our goal now is to prove that update queries preserve the invariant that the hint remains correct after executing UpdateDB and UpdateHint (thereby implying that lookups executed after an update are correct). Let $\mathsf{pp}'.Q$, $\mathsf{pp}'.M$, and $\mathsf{pp}'.N$ denote the public parameters before the first update is made. Suppose that the hint satisfies Eq. 1 before an update is made and that for all $q \in [\mathsf{pp}'.Q]$ and $m \in [\mathsf{pp}.M]$, $\widehat{\mathsf{DB}}_q[m]$ is a valid opening for $\mathsf{DB}_q[m]$ under $\mathsf{com}_q = \mathsf{VC.Commit}(\mathsf{DB}_q)$.

UpdateDB first initializes an empty set U' and the for loop on line 3 (Fig. 6) cycles through the set of updates $(\mathsf{op}, \mathsf{idx}, x) \in U$ and does the following: if $\mathsf{op} = \mathsf{add}$ then it increments $\mathsf{pp}'.N$ and adds $(\mathsf{add}, \mathsf{pp}'.N, x)$ to U', otherwise $\mathsf{op} = \mathsf{edit}$ and it adds $(\mathsf{op}, \mathsf{idx}, x)$ to U'. This step ensures that the public parameters are correct for the new database and that new values are assigned the correct index.

Using the new $\mathsf{pp}'.N$, the server also extends the size of DB and $\widehat{\mathsf{DB}}$ accordingly and initializes the new entries.

UpdateDB then processes updates by partition (Fig. 6, line 11). For $q \in [\mathsf{pp}'.Q]$, let $U'_q \subseteq U'$ denote the subset of updates targeting partition q. For every $(\mathsf{op}, \mathsf{idx}, x) \in U'_q$ and the corresponding shift $m \leftarrow ((\mathsf{idx} - 1) \bmod \mathsf{pp}'.M) + 1$ the algorithm executes the following:

- If op = edit (Fig. 6, line 18), the server updates $\mathsf{DB}_q[m] \leftarrow x_{new}$ and sets $\mathsf{com}_q \leftarrow \mathsf{VC.Update}(\mathsf{com}_q, m, x_{old}, x_{new})$.
- If DB_q is a new partition (Fig. 6, line 21), the server commits to this new partition, i.e., $\mathsf{com}_q \leftarrow \mathsf{VC.Commit}(\mathsf{DB}_q)$.
- For every q and every $m \in [\mathsf{pp}'.M]$ (Fig. 6, line 23), the server recomputes the opening as $\widehat{\mathsf{DB}}_q[m] \leftarrow \mathsf{VC.Open}(m, \mathsf{DB}_q)$.

Thus, by correctness of VC, for all q, com_q commits to DB_q and the opening proofs are all correct with respect to the new DB.

Now consider UpdateHint. With honest servers, we have $\mathbf{pp}[1] = \mathbf{pp}[2]$ and $\mathbf{U}[1] = \mathbf{U}[2]$, so no abort is triggered (i.e., Fig. 6, line 2 is skipped). UpdateHint then samples fresh permutations $\sigma_q \leftarrow\$ \mathsf{Permute}(M)$ for each new partition. Let U' denote the update information returned by the servers. For each $(\mathsf{op}, \mathsf{idx}, x) \in U'$, let $(\hat{q}, \hat{m})$ be the partition and offset of idx. We consider two cases:

- If op = edit, the servers must have set $x = x_{new} \oplus x_{old}$ where x_{new} and x_{old} are the new and old records at index idx respectively (Fig. 6, line 20). The client computes $\mathsf{ind} = \sigma_{\hat{q}}^{-1}(\hat{m})$ and updates $h_{\mathsf{ind}} \leftarrow h_{\mathsf{ind}} \oplus x = h_{\mathsf{ind}} \oplus x_{new} \oplus x_{old}$ thereby removing the old element in the hint and adding the new one.
- If op = add, then $x = x_{new} \oplus x_{old} = x_{new} \oplus 0^\ell = x_{new}$ since new database entries are initialized to 0. As in the previous case, the client computes $\mathsf{ind} = \sigma_{\hat{q}}^{-1}(\hat{m})$ and updates $h_{\mathsf{ind}} \leftarrow h_{\mathsf{ind}} \oplus x = h_{\mathsf{ind}} \oplus x_{new}$, thereby XOR-ing the new value x_{new} into h_{ind}.

In either case, the invariant for h_{ind} is preserved, and all other h_m are unaffected by the update and remain correct. The hint remains correct with respect to the updated database after the update is made. Thus, any later honest Answer returns the correct value, and thus win is set to **true** with probability 0. □

B.2 Proof of Integrity (Theorem 5.3.2)

Proof. Let good $\in \{1, 2\}$ denote the index of the honest server and, for ease of notation, let bad = 3 − good denote the index of the malicious server. Let $\mathsf{pp}_{\mathsf{bad}}$ and $\mathsf{DB}_{\mathsf{init}}$ be the public parameters and the initial database records generated by $\mathcal{A}_0$. Suppose for a contradiction that the adversary wins the integrity game, i.e., the client queries for an index $\mathsf{idx} \in [N]$ and outputs a tuple $(\mathsf{st}'_C, x, \mathsf{hint}')$ such that $x \neq \bot$ and $x \neq \mathsf{DB}[\mathsf{ind}]$. In other words, the client reconstructs and accepts an incorrect value. We proceed with respect to the possible actions of $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$:

$\mathcal{A}_0$ **generates malicious public parameters.** Let $\mathsf{d}_{\mathsf{bad}}$ denote the digest contained in $\mathsf{pp}_{\mathsf{bad}}$. If $\mathsf{pp}_1 \neq \mathsf{pp}_2$, then the client sets $\mathsf{st}_C.\mathsf{abort} \leftarrow$ **true** (Fig. 4, RequestHint, line 4). Consequently, when the client executes Recon, it always returns $(\mathsf{st}_C, \bot, \bot)$ (Fig. 5, Recon, line 1) and never proceeds with the remainder of the Recon algorithm. Thus, the probability that win = **true** is 0.

$\mathcal{A}_1$ **generates a malicious response.** Let DB and DB′ be the databases streamed by Server 1 and Server 2, respectively. Since at least one server is honest, there exists an index $\mathsf{idx} \in [\mathsf{st}_C.\mathsf{pp}.N]$ such that $\mathsf{DB}[\mathsf{idx}] \neq \mathsf{DB}'[\mathsf{idx}]$. This implies $\mathbf{resp}[1] \neq \mathbf{resp}[2]$. By the definition of the VerSetup algorithm, whenever $\mathbf{resp}[1] \neq \mathbf{resp}[2]$, the client sets $\mathsf{st}_C.\mathsf{abort} \leftarrow$ **true** (Fig. 4, VerSetup, line 3). As in the previous case, executing Recon then always outputs $(\mathsf{st}_C, \bot, \bot)$ (Fig. 5, Recon, line 1), so the probability that win is set to **true** is 0.

$\mathcal{A}_2$ **generates malicious and valid update.** Let $\mathsf{pp}_{\mathsf{bad}}$ denote the updated public parameters and U_{bad} the update information provided by the malicious server. If $\mathsf{pp}_1 \neq \mathsf{pp}_2$ or $U_1 \neq U_2$, then by the definition of the UpdateHint algorithm the client sets $\mathsf{st}_C.\mathsf{abort} \leftarrow$ **true** (Fig. 6, UpdateHint, line 2). Consequently, any subsequent call to Recon will always return $(\mathsf{st}_C, \bot, \bot)$ (Fig. 5, Recon, line 1). Since one of the servers is honest, then the probability that the client accepts a malformed update and win is set to **true** is 0.

$\mathcal{A}_2$ **generates a malicious answer.** Parse the answer as $\mathsf{ans}_{\mathsf{bad}} = (\mathsf{ans}_{\mathsf{bad},1}, \dots, \mathsf{ans}_{\mathsf{bad},\mathsf{pp}.Q})$ and the digest as $(\mathsf{com}_1, \dots, \mathsf{com}_{\mathsf{pp}.Q}) \leftarrow \mathsf{pp.d}$. Suppose that there exists some partition index $q \in [\mathsf{pp}.Q]$ such that

$$\mathsf{VC.Verify}(\mathsf{com}_q, \mathsf{qry}_{\mathsf{bad},q}, \mathsf{rec}_{\mathsf{bad},q}, \pi_{\mathsf{bad},q}) = 1,$$

where $\mathsf{ans}_{\mathsf{bad},q} = (\mathsf{rec}_{\mathsf{bad},q} \| \pi_{\mathsf{bad},q})$ and $\mathsf{rec}_{\mathsf{bad},q} \neq \mathsf{DB}_q[\mathsf{qry}_{\mathsf{bad},q}]$. Because the commitment com_q is correct (due to the determinism of VC.Commit we must have equality of $\mathsf{pp}_1 = \mathsf{pp}_2$ for honestly generated parameters), this would violate the binding property of VC. The probability that this happens is $\mathsf{negl}(\lambda)$. Taking a union bound over $Q = \mathsf{poly}(\lambda)$ database partitions and $\mathsf{poly}(\lambda)$-many queries still yields at most negligible probability.

Hence, we reach a contradiction, completing the proof. □

B.3 Proof of Privacy (Theorem 5.3.3)

In this section, we prove Theorem 5.3.3 with a sequence of game hops. As Tapir extends SinglePass to achieve malicious security, this proof follows the privacy proof in [LP24] with additional consideration for integrity and updates.

For simplicity, we denote $[k] = \{1, 2\}$ with $\{0, 1\}$ in this proof. This allows us to reference the malicious server with ¬good.

Proof Sketch. The hint generation and online phase of Tapir follows that of SinglePass, except for the addition of integrity checks and the client's abort status. In the privacy game, Query is run for one of the two adversarially chosen indices, and it outputs a sequence of database indices ($\mathsf{pp}.Q$ values in $[\mathsf{pp}.M]$) for each server. Each server answers the query by sending the database records and proofs at these indices back to the client. The client verifies the correctness of all the received records based on the corresponding proof and the database commitments stored in the client's state. The client aborts if a single verification fails. Hence, integrity (and thereby privacy with abort) is ensured by the deterministic commitment generation and binding of VC. To show that neither the set of indices in a query nor the abort status reveals any information to the server, we leverage the Show-and-Shuffle theorem introduced by [LP24] and restated in Fig. 10.

Proof. Let $\mathcal{A}$ be the adversary in the privacy game $\mathbf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ in Fig. 1. The adversary has the capabilities to corrupt an honest server defined by bit good and to select any well-formed sequence of records $\mathsf{DB}_{\mathsf{init}}$ and update information U for use in the game. We denote the maximum number of calls to the query and update oracles by T_q and T_u. For all $N, T_\mathrm{q}, T_\mathrm{u} \in \mathrm{poly}(\lambda)$ where $T_\mathrm{q} + T_\mathrm{u} \in \mathrm{poly}(\lambda)$. Let $\Pr[\mathrm{G}_x]$ denote the probability that game G_x outputs 1.

We now describe the sequence of games G_0 to G_6 for proving privacy with abort of Tapir as stated in Theorem 5.3.3. See Figures 11 and 12 for the detailed pseudocode.

- **Game $\mathbf{G}_0$:** This is equal to the privacy game $\mathbf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ (Fig. 1). The call to Recon has been replaced with the according pseudocode of this algorithm. A flag *BAD* is set to true if the adversary breaks binding of the VC scheme (G_0, line 17).
- **Game $\mathbf{G}_1$:** This game is equal to G_0 until *BAD* is set, so $\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] \leq \Pr[BAD]$ where $\Pr[BAD]$ denotes the probability of the flag *BAD* being set to **true**.
- **Bounding $\Pr[\boldsymbol{BAD}]$:** Let G_1' be identical to G_1, except that it returns *BAD* instead of $b = b_{\mathsf{chall}}$ such that $\Pr[BAD] = \Pr[G1']$. Let G_1'' be identical to G_1', except that it additionally keeps track of the adversary's oracle inputs and outputs to Query, Recon, and Update as tuples in the sets S_q, S_r, S_u. Hence, $\Pr[BAD] = \Pr[\mathrm{G}_1'] = \Pr[\mathrm{G}_1'']$.
  Let $\mathcal{B}$ be an adversary in the integrity game $\mathbf{G}^{\mathsf{Int\text{-}APIR}}_{\mathsf{APIR}}$. This adversary $\mathcal{B}$ simulates G_1'' for $\mathcal{A}$, where it samples a challenger bit b_{chall}, honestly executes the steps in the game, and stores the oracle inputs and outputs in the respective variables. Once $\mathcal{A}$ halts, $\mathcal{B}$ executes the same oracle calls in their game as $\mathcal{A}$ does in G_1''. For calls to oracle QueryInt, $\mathcal{B}$ inputs idx_b from $(\mathsf{idx}_0, \mathsf{idx}_1, \mathsf{qry}) \longleftarrow S_q$. For calls to oracle UpdateInt, $\mathcal{B}$ inputs $(\mathsf{pp}, U) \longleftarrow S_u$. For calls to oracle ReconInt, $\mathcal{B}$ inputs ans from $(\mathsf{ans}, \mathsf{abort\text{-}bit}) \longleftarrow S_r$.
  If $\mathcal{A}$ finds a valid proof $\pi'_{\neg\mathsf{good},q}$ for a value $\mathsf{rec}'_{\neg\mathsf{good},q} \neq \mathsf{rec}_{\neg\mathsf{good},q}$ to a commitment com_q, then $\mathcal{B}$ will win the integrity game. Thus, $\Pr[BAD] \leq \mathsf{Adv}^{\mathsf{int}}_{\mathcal{A},\mathsf{APIR}}(\lambda)(\mathcal{B})$ and
  $$\Pr[\mathrm{G}_0] = \mathsf{Adv}^{\mathsf{int}}_{\mathcal{A},\mathsf{APIR}}(\lambda)(\mathcal{B}) + \Pr[\mathrm{G}_1].$$
- **Game $\mathbf{G}_2$:** This is equal to the privacy game $\mathbf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ (Fig. 1). The calls to RequestHint, VerSetup, Query, and UpdateHint have been replaced with the according pseudocode of these algorithms.
- **Game $\mathbf{G}_{(3,i)}$ for $i \in [T_\mathrm{q}]$:** This game is equal to G_2, except for the following change: the first i queries are constructed based on freshly sampled pseudorandom permutations $P_1, ..., P_{\mathsf{pp}.Q}$ over $[M]$. For the last $T_\mathrm{q} - i$ queries, the game runs exactly as G_0 using the permutations stored in $\mathsf{st}_C.\mathsf{ck}$ and swapping out elements. Distinguishing between G_2 and $\mathrm{G}_{(3,1)}$ is equivalent to breaking the Show-and-Shuffle game $\mathbf{G}^{\mathsf{Ind\text{-}SaS}}$ Fig. 10 as shown in [LP24]. For every $i \in [T_\mathrm{q} - 1]$, the same argument as above holds for $\mathrm{G}_{(3,i)}$ and $\mathrm{G}_{(3,i+1)}$. After T_q game hops, all queries are generated using freshly sampled permutations instead of the stored ones used in G_2. Hence, by the indistinguishability of $\mathbf{G}^{\mathsf{Ind\text{-}SaS}}$ (Theorem A.1.1)
  $$\begin{aligned}|\Pr[G_2] - \Pr[G_{(3,1)}]| &= |\Pr[G_{(3,i)}] - \Pr[G_{(3,i+1)}]| \\ &= \mathsf{Adv}^{\mathsf{ind-sas}}_{\mathcal{A}}(\mathsf{pp}.Q, \mathsf{pp}.M) = 0\end{aligned}$$
- **Game $\mathbf{G}_4$:** This game is equal to $\mathrm{G}_{(3,T_\mathrm{q})}$.
- **Game $\mathbf{G}_{(5,i)}$ for $i \in [T_\mathrm{q}]$:** This game is equal to the previous game, except for the following change: the first i queries, qry_b for $b \in \{0, 1\}$, are sampled pseudorandomly and uniformly from $[\mathsf{pp}.M]$. For the last $T_\mathrm{q} - i$ queries, the game runs exactly as G_4. Consider the distribution for qry_b for the t-th call to the query oracle for $b \in \{0, 1\}$ and $t \in [T_\mathrm{q}]$. If $t \leq i$, then by definition qry_0 and qry_1 are uniformly sampled from $[M]$. For all other queries, each $P_q(\mathsf{ind})$ is uniformly distributed over $[M]$, as the permutations are sampled independently and uniformly. For $b = 0$, it follows that all values $(\mathsf{qry}_{0,q})_{q\in[Q]}$ are uniformly distributed over $[M]$. For $b = 1$, the same follows but for $(\mathsf{qry}_{1,q})_{q\in[Q]\setminus q^*}$. By definition qry_{1,q^*} is also uniformly distributed over $[M]$. Hence, the entire query vector is uniformly distributed, and thus the distributions between G_4 and $\mathrm{G}_{(5,1)}$ are equal. An analogous argument applies to the games $\mathrm{G}_{(5,i)}$ and $\mathrm{G}_{(5,i+1)}$ for $i \in [T_\mathrm{q} - 1]$.
  $$|\Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_{(5,1)}]| = 0$$
- **Game $\mathbf{G}_6$:** This game is equal to $\mathrm{G}_{(5,T_\mathrm{q})}$.

In this final game, we note that all queries are now pseudorandomly sampled and fully independent of any server input. We can conclude that through the above series of game hops, the adversary has an advantage negligible in the security parameter λ in winning the privacy game $\mathbf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ over a random guess:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{priv}}_{\mathcal{A},\mathsf{APIR}}(\lambda) :=& \mathsf{Adv}^{\mathsf{ind-sas}}_{\mathcal{A}}(\mathsf{pp}.Q, \mathsf{pp}.M) + \mathsf{Adv}^{\mathsf{int}}_{\mathcal{A},\mathsf{APIR}}(\lambda) \\ \leq& \mathsf{negl}(\lambda)\end{aligned}$$

Both N and T are polynomial in λ, and so we have a polynomial number of game hops.

□

G_0: $\mathcal{O}_{\mathsf{ReconPriv}}(\mathsf{ans}_{\neq\mathsf{good}})$

```
1 :  **if** $\neg\mathsf{st}_C.\mathsf{abort} \wedge \mathsf{st}_C.q \neq \bot$ **then**
2 :    $\mathsf{ans}_{\mathsf{good}} \leftarrow \mathsf{inpt}$
     ~~$(\mathsf{st}'_C, x, \mathsf{hint}',) \leftarrow \mathsf{Recon}(\mathsf{st}_C, \mathsf{hint}, (\mathsf{ans}_b)_{b\in\{0,1\}})$~~
3 :    $\mathsf{pp} \leftarrow \mathsf{st}_C.\mathsf{pp},\ \mathsf{ind} \leftarrow \mathsf{st}_C.\mathsf{ind},\ q^* \leftarrow \mathsf{st}_C.q^*$
4 :    $(\mathsf{qry}_{0,1}, \ldots, \mathsf{qry}_{0,\mathsf{pp}.Q}) \leftarrow \mathsf{st}_C.\mathsf{qry}_0$
5 :    $(\mathsf{qry}_{1,1}, \ldots, \mathsf{qry}_{1,\mathsf{pp}.Q}) \leftarrow \mathsf{st}_C.\mathsf{qry}_1$
6 :    $((\mathsf{rec}_{0,1}||\pi_{0,1}), \ldots, (\mathsf{rec}_{0,\mathsf{pp}.Q}||\pi_{0,\mathsf{pp}.Q})) \leftarrow \mathsf{ans}_0$
7 :    $((\mathsf{rec}_{1,1}||\pi_{1,1}), \ldots, (\mathsf{rec}_{1,\mathsf{pp}.Q}||\pi_{1,\mathsf{pp}.Q})) \leftarrow \mathsf{ans}_1$
8 :    $(\mathsf{com}_1, \ldots, \mathsf{com}_{\mathsf{pp}.Q}) \leftarrow \mathsf{pp.d}$
9 :    $(h_1, \ldots, h_{\mathsf{pp}.M}) \leftarrow \mathsf{hint}$
10 : // Verify commitments
11 :   **for** $q \in [\mathsf{pp}.Q]$ **do**
12 :     **for** $b \in \{0,1\}$ **do**
13 :       $v \leftarrow \mathsf{VC.Verify}(\mathsf{com}_q, \mathsf{qry}_{b,q}, \mathsf{rec}_{b,q}, \pi_{b,q})$
14 :       **if** $\neg v$ **then**
15 :         $\mathsf{st}_C.\mathsf{abort} \leftarrow$ **true**
16 :       **elseif** $\mathsf{rec}_{b,q} \neq \mathsf{st}_S.\mathsf{DB}[\mathsf{qry}_{b,q}]$
17 :         $BAD \leftarrow$ **true**
18 : // Recover record
19 :   **if** $\neg\mathsf{st}_C.\mathsf{abort}$
20 :     $x \leftarrow \left(\bigoplus_{q\in[\mathsf{pp}.Q]\setminus\{q^*\}} \mathsf{rec}_{1,q}\right) \oplus h_{\mathsf{ind}}$
21 :     **for** $q \in [\mathsf{pp}.Q] \setminus \{q^*\}$ **do** // Update hint
22 :       $h_{\mathsf{ind}} \leftarrow h_{\mathsf{ind}} \oplus \mathsf{rec}_{0,q} \oplus \mathsf{rec}_{1,q}$
23 :       $h_{r_q} \leftarrow h_{r_q} \oplus \mathsf{rec}_{0,q} \oplus \mathsf{rec}_{1,q}$
     ~~**if** $x = \bot$ **then**~~
     ~~abort-bit $\leftarrow$ **true**~~
     ~~**else** abort-bit $\leftarrow$ **false**~~
24 :     $\mathsf{st}_C.\mathsf{ck} \leftarrow \{\sigma_q \mid q \in [\mathsf{pp}.Q]\}$
25 :     $\mathsf{st}_C.q{*} \leftarrow \bot$ $\mathsf{hint} \leftarrow (h_1, \ldots, h_{\mathsf{pp}.M})$
26 :   **return** $\mathsf{st}_C.\mathsf{abort}$ ~~abort-bit~~
```

G_1 $\boxed{G'_1}$: $\mathcal{O}_{\mathsf{ReconPriv}}(\mathsf{ans}_{\mathsf{bad}})$

```
     ⋮
13 :       $v \leftarrow \mathsf{VC.Verify}(\mathsf{com}_q, \mathsf{qry}_{b,q},$
              $\mathsf{rec}_{b,q}, \pi_{b,q})$
14 :       **if** $\neg v$ **then**
15 :         $\mathsf{st}_C.\mathsf{abort} \leftarrow$ **true**
16 :       **elseif** $\mathsf{rec}_{b,q} \neq \mathsf{st}_S.\mathsf{DB}[\mathsf{qry}_{b,q}]$
17 :         $BAD \leftarrow$ **true**
18 :         $\mathsf{st}_C.\mathsf{abort} \leftarrow$ **true**
19 :   **if** $\neg\mathsf{st}_C.\mathsf{abort}$
20 :     $x \leftarrow \left(\bigoplus_{q\in[\mathsf{pp}.Q]\setminus\{q^*\}} \mathsf{rec}_{1,q}\right) \oplus h_{\mathsf{ind}}$
21 :     **for** $q \in [\mathsf{pp}.Q] \setminus \{q^*\}$ **do**
22 :       $h_{\mathsf{ind}} \leftarrow h_{\mathsf{ind}} \oplus \mathsf{rec}_{0,q} \oplus \mathsf{rec}_{1,q}$
23 :       $h_{r_q} \leftarrow h_{r_q} \oplus \mathsf{rec}_{0,q} \oplus \mathsf{rec}_{1,q}$
24 :     $\mathsf{st}_C.\mathsf{ck} \leftarrow \{\sigma_q \mid q \in [\mathsf{pp}.Q]\}$
25 :     $\mathsf{st}_C.q{*} \leftarrow \bot,\ \mathsf{hint} \leftarrow (h_1, \ldots, h_{\mathsf{pp}.M})$
   $\boxed{\textbf{return } BAD}$ **return** $\mathsf{st}_C.\mathsf{abort}$
```

G_2: $\mathbf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}(\mathcal{A}, \lambda, \mathsf{sp})$

```
     ⋮
8 :  // Preprocessing
     ~~$(\mathsf{st}_C, \mathbf{hq}) \leftarrow \mathsf{RequestHint}((\mathsf{pp})_{b\in\{0,1\}})$~~
9 :  $\mathsf{st}_C.\mathsf{abort} \leftarrow$ **true**, $(\mathsf{hq}_0, \mathsf{hq}_1) \leftarrow \bot^2$
10 : // Verify public parameters
11 : **if** $\mathsf{pp}_0 = \mathsf{pp}_1$ **then**
12 :   $\mathsf{st}_C.\mathsf{abort} \leftarrow$ **false**
13 :   $(\mathsf{hq}_0, \mathsf{hq}_1) \leftarrow (\texttt{“Stream DB, please!”})^2$
14 :   $\mathsf{st}_C.\mathsf{pp} \leftarrow \mathsf{pp}_0$
15 : $(\mathsf{st}'_S, \mathsf{resp}_{\mathsf{good}}) \leftarrow \mathsf{AnsHintReq}(\mathsf{st}_S, \mathsf{hq}_{\mathsf{good}})$
16 : $(\mathsf{st}'_{\mathcal{A}}, \mathsf{resp}_{\neg\mathsf{good}}) \leftarrow \mathcal{A}_1(\mathsf{st}_{\mathcal{A}}, \mathsf{hq}_{\neg\mathsf{good}})$
     ~~$(\mathsf{st}'_C, \mathsf{hint}) \leftarrow \mathsf{VerSetup}(\mathsf{st}_C, (\mathsf{resp}_b)_{b\in\{0,1\}})$~~
     ~~$\mathsf{st}_C \leftarrow \mathsf{st}'_C$~~
17 : **if** $\mathsf{st}_C.\mathsf{abort} \vee \mathsf{resp}_0 \neq \mathsf{resp}_1$ **then**
18 :   $\mathsf{st}_C.\mathsf{abort} \leftarrow$ **true**
19 : **else**
20 :   $(\mathsf{ck}, \mathsf{hint}) \leftarrow \bot^2$
21 : // Generate permutations
22 :   **for** $q \in [\mathsf{st}_C.\mathsf{pp}.Q]$ **do**
23 :     $\mathsf{ck}.\sigma_q \leftarrow\!\!\$\ \mathsf{Permute}(\mathsf{st}_C.\mathsf{pp}.M)$
24 :   $\mathsf{DB} \leftarrow \mathsf{resp}_0$
25 :   **for** $m \in [\mathsf{st}_C.\mathsf{pp}.M]$ **do**
26 :     $\mathsf{hint}.h_m = \bigoplus_{q=1}^{\mathsf{st}_C.\mathsf{pp}.Q} \mathsf{DB}_q[\mathsf{ck}.\sigma_q(m)]$
27 :   $\mathsf{st}_C.\mathsf{ck} \leftarrow \mathsf{ck}$
     ⋮
```

Figure 11: Games G_0 (left), G_1 and G'_1 (middle), and $G_2\mathbf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ (right) for the game hops for proving $\mathbf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ for Tapir (Theorem 5.3.3).

G_2:$\mathcal{O}_{\mathsf{QueryPriv}}(\mathsf{idx}_0, \mathsf{idx}_1)$

```
if $t_q < T_q \wedge \neg st_C.abort \wedge st_C.q^* = \bot$ then
    ~~$(st'_C, (qry_b)_{b \in \{0,1\}}) \leftarrow Query(st_C, idx_{b_{chall}})$~~
    $pp \leftarrow st_C.pp,\ \{\sigma_q \mid q \in [pp.Q]\} \leftarrow st_C.ck$
    $q^* \leftarrow \lceil idx_{b_{chall}} / pp.M \rceil$
    $m^* \leftarrow ((idx_{b_{chall}} - 1) \bmod pp.M) + 1$
    Find $ind \in [pp.M]$, s.t. $\sigma_{q^*}(ind) = m^*$
    $(r^*, r_1, ..., r_{pp.Q}) \leftarrow\!\$\ [pp.M]^{pp.Q+1}$
    $(qry_0, qry_1) \leftarrow \bot^2$
    for $q \in [pp.Q]$ do
        $qry_0.qry_{0,q} \leftarrow \sigma_q(r_q)$
        $qry_1.qry_{1,q} \leftarrow \sigma_q(ind)$
        if $q \neq q^*$ then
            Swap $\sigma_q(ind)$ and $\sigma_q(r_q)$
        else $qry_1.qry_{1,q^*} \leftarrow r^*$
    $st_C.ck \leftarrow \{\sigma_q \mid q \in [pp.Q]\}$
    $st_C.q^* \leftarrow q^*,\ st_C.ind \leftarrow ind$
    $st_C.qry_0 \leftarrow qry_0,\ st_C.qry_1 \leftarrow qry_1$
    $(st'_S, ans_{good}) \leftarrow Answer(st_S, qry_{good})$
    ⋮
```

$G_{(3,i)}$:$\mathcal{O}_{\mathsf{QueryPriv}}(\mathsf{idx}_0, \mathsf{idx}_1)$

```
⋮
    $(qry_0, qry_1) \leftarrow \bot^2$
    if $t_q < i$ then
        $(P_1, ..., P_{pp.Q}) \leftarrow\!\$\ Permute(pp.M)^{pp.Q}$
        for $q \in [pp.Q]$ do
            if $q \neq q^*$ then
                $qry_0.qry_{0,q} \leftarrow P_q(r_q)$
                $qry_1.qry_{1,q} \leftarrow Pq(ind)$
            else
                $qry_1.qry_{1,q} \leftarrow\!\$\ [pp.M]$
    else
        for $q \in [pp.Q]$ do
            $qry_0.qry_{0,q} \leftarrow \sigma_q(r_q)$
            $qry_1.qry_{1,q} \leftarrow \sigma_q(ind)$
            if $q \neq q^*$ then
                Swap $\sigma_q(ind)$ and $\sigma_q(r_q)$
            else $qry_1.qry_{1,q^*} \leftarrow r^*$
    $st_C.ck \leftarrow \{\sigma_q \mid q \in [pp.Q]\}$
⋮
```

G_4:$\mathcal{O}_{\mathsf{QueryPriv}}(\mathsf{idx}_0, \mathsf{idx}_1)$

```
⋮
~~if $t_q < i$ then~~
    $(P_1, ..., P_{pp.Q}) \leftarrow\!\$\ Permute(pp.M)^{pp.Q}$
    for $q \in [pp.Q]$ do
        if $q \neq q^*$ then
            $qry_0.qry_{0,q} \leftarrow P_q(r_q)$
            $qry_1.qry_{1,q} \leftarrow Pq(ind)$
        else
            $qry_1.qry_{1,q} \leftarrow\!\$\ [pp.M]$
~~else~~
    ~~for $q \in [pp.Q]$ do~~
        ~~$qry_0.qry_{0,q} \leftarrow \sigma_q(r_q)$~~
        ~~$qry_1.qry_{1,q} \leftarrow \sigma_q(ind)$~~
        ~~if $q \neq q^*$ then~~
            ~~Swap $\sigma_q(ind)$ and $\sigma_q(r_q)$~~
        ~~else $qry_{1,q^*} \leftarrow r^*$~~
~~$st_C.ck \leftarrow \{\sigma_q \mid q \in [pp.Q]\}$~~
$st_C.q^* \leftarrow q^*,\ st_C.ind \leftarrow ind$
⋮
```

$G_{(5,i)}$: $\mathcal{O}_{\mathsf{QueryPriv}}(\mathsf{idx}_0, \mathsf{idx}_1)$

```
⋮
    $pp \leftarrow st_C.pp$~~$,\ \{\sigma_q \mid q \in [pp.Q]\} \leftarrow st_C.ck$~~
    if $t_q < i$ then
        $qry_0, qry_1 \leftarrow [pp.M]^{2 \cdot pp.Q}$
    else
        $q^* \leftarrow \lceil idx_{b_{chall}} / pp.M \rceil$
        $m^* \leftarrow ((idx_{b_{chall}} - 1) \bmod pp.M) + 1$
    ⋮
```

G_6 : $\mathcal{O}_{\mathsf{QueryPriv}}(\mathsf{idx}_0, \mathsf{idx}_1)$

```
if $t_q < T_q \wedge \neg st_C.abort \wedge st_C.q^* = \bot$ then
    $pp \leftarrow st_C.pp$
    ~~if $t_q < i$ then~~
        $(qry_0, qry_1) \leftarrow [pp.M]^{2 \cdot pp.Q}$
    ~~else~~
        ~~$q^* \leftarrow \lceil idx_{b_{chall}} / pp.M \rceil$~~
        ~~$m^* \leftarrow ((idx_{b_{chall}} - 1) \bmod pp.M) + 1$~~
        ~~Find $ind \in [pp.M]$, s.t. $\sigma_{q^*}(ind) = m^*$~~
        ~~$(r^*, r_1, ..., r_{pp.Q}) \leftarrow\!\$\ [pp.M]^{pp.Q+1}$~~
        ~~$qry_0, qry_1 \leftarrow \bot$~~
        ~~$(P_1, ..., P_{pp.Q}) \leftarrow\!\$\ Permute(pp.M)^{pp.Q}$~~
        ~~for $q \in [pp.Q]$ do~~
            ~~$qry_0.qry_{0,q} \leftarrow P_q(r_q)$~~
            ~~$qry_1.qry_{1,q} \leftarrow Pq(ind)$~~
        ~~$st_C.q^* \leftarrow q^*,\ st_C.ind \leftarrow ind,$~~
    $st_C.qry_0 \leftarrow qry_0,\ st_C.qry_1 \leftarrow qry_1$
    ⋮
```

Figure 12: Games $G_2 Oracle_{\mathsf{QueryPriv}}$ (top left), $G_{(3,i)}$ (top middle), G_4 (top right), $G_{(5,i)}$ (bottom left) and G_6 (bottom middle) for the game hops for proving $G^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ for Tapir (Theorem 5.3.3).