TAPIR: A Two-Server Authenticated PIR Scheme with Preprocessing

Francesca Falzon[1[0000−0001−8415−6237]], Laura Hetz[1[0009−0000−3673−6421]], and Annamira O'Toole[1[0009−0004−9535−2590]]

ETH Zurich, Zurich, Switzerland
{ffalzon,lahetz,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 is being requested to the server(s) that store 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., [16,33,37,41,42]. Notably, PIR has many real-world applications such as transparency systems [15,33], private web search [5,32], private messaging [2,13,44], and private lookup [3,37]. PIR has even been deployed at large scale by Apple to enable private caller ID [3].

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., [12,34,45,46,48]), and may require additional guarantees such as append-only properties. Furthermore, these systems must operate securely even in the presence of malicious servers and when 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* [40].

To address the problem of selective failure when providing integrity in PIR, Colombo et al. [15] 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 attempting a selective failure attack must cause the client to abort with a probability that is independent of the queried record, Recent works have proposed APIR schemes for both the single- [9,15,19,22] and multi- [15,49] 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 [37,41,42] to reduce query-time costs.

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

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

1.1 Contributions

We answer the above question affirmatively with Tapir, the first two-server APIR with preprocessing that achieves sublinear bandwidth and computational query complexity while supporting efficient updates.

New Definitions (§ 2). The combination of functionalities supported by Tapir requires a new interface. Prior APIR schemes were either multi-server without preprocessing [1,15], single-server [4,15,19], or lacked update support in the client-preprocessing model [22]. 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. The model considers a malicious-but-non-colluding server, capturing adversaries that adaptively issue lookup and update queries. In the full version of this work [23], we further generalize these definitions to both the single-server and multi-server settings for $k \geq 2$ servers.

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 [42], 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 non-trivial, as the client must abort with a probability that is independent 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. At the cost of streaming the database once from each server during the offline phase, Tapir achieves sublinear computation and communication in the online query phase. Additionally, our approach offers efficiency trade-offs between runtime and bandwidth depending on the used commitment scheme. 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 [30] provides online bandwidth comparable to the unauthenticated SinglePass. This adaptability allows Tapir to be tuned to the requirements of diverse use cases.

Update Support (§ 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 [42] 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 [15] 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 band-

width than SinglePass for databases of size $N = 2^{20}$ when instantiated with a state-of-the-art algebraic VC scheme such as Pointproofs [30] 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 (A)PIR.

Multi-Server PIR. In "traditional" multi-server PIR schemes [7,8,14,27], the server computation is linear in the database size for each query. The client-preprocessing model [17] shifts much of this cost to an *offline* phase, where the client and servers jointly preprocess 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 [37,41,42]. SinglePass [42] improves over prior client-preprocessing schemes [17,37,41] with an offline phase that only requires a single pass over the database, while also supporting updates.

Authenticated PIR. PIR 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. [15] 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 [9]. The scheme of Dietz and Tessaro [19] avoids a trusted setup, but only supports 1-bit records without a concrete cost analysis. Balanced-PIR [4], 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 updates are supported, 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. [15] authenticate a linear PIR scheme using Merkle trees. They also propose a second two-server APIR scheme that uses a message authentication code (MAC) [18] to verify server responses. We refer to these constructions as APIR-Matrix-MT and APIR-DPF; both incur online computation linear in the size of the database.

Crust [49] extends [17] 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 [15], which assume at least one of the servers to be honest without specifying which one. Alon and Beimel [1] modify the compiler of [21] to generically transform

a semi-honest PIR scheme into one with security-with-abort. Falk et al. [22] 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 write $[n]$ for the set $\{1, \ldots, n\}$ and denote vectors in bold e.g., $\mathbf{v} = (v_1, \ldots, v_n)$.

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 $m \in [M]$. Databases are padded with 0-records to their maximum capacity $M \cdot Q$.

To formalize database updates, we denote the initial fixed values 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. Throughout this paper, we generally refer to the non-static ones for convenience.

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 $\mathsf{pp}.N$.

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 [10] and Pointproofs [30].

Definition 1 ([10]). *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 a commitment V.*
- $\pi \leftarrow \mathsf{Open}(i, \mathbf{v})$ *takes as input index $i \in [n]$ and vector $\mathbf{v}$ and outputs proof π.*
- $b \leftarrow \mathsf{Verify}(V, i, v, \pi)$ *takes as input a commitment V, index $i \in [n]$, value v, and a proof π. It outputs $b = 1$ if V is consistent with v and $b = 0$ otherwise.*
- $(V') \leftarrow \mathsf{Update}(V, i, v, v')$ *takes as input a commitment V and updates the value at index i from v to v'. It outputs an updated commitment V'.*

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 PPT adversary the probability of finding a commitment V and tuples $(i, v, \pi), (i, v', \pi')$ such that $v \neq v'$ and

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

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 [20,36,24]. At a high level, the algorithm takes as input a list of the 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 be used to sample an unbiased permutation of the set $[N]$ in $O(N)$ time. We restate the following lemma.

Lemma 1 ([20,36,24]). *For any $N \in \mathbb{N}$, there exists an algorithm $\mathsf{Permute}(N)$ that can output a permutation of the set $[N]$ sampled uniformly from the set of all permutations of $[N]$, denoted σ_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) enables a client to retrieve an element from a database without allowing the server holding the database to learn which element was accessed. 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 [17,37,41,42] schemes achieve both sublinear query communication and computational cost under computational security guarantees. In these schemes, the servers and client jointly preprocess the database, resulting in 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., [1,4,9,15,19,22]. In the next section, we extend the prior work and formally introduce two-server APIR with client-preprocessing and support for updates.

3 Two-Server APIR with Preprocessing

In this section, we formalize two-server 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, integrity, and privacy with abort (§ 3.2).

We distinguish between initial fixed-value protocol parameters and those that may change during execution, for example, via updates. Additionally, we assume that the database is stored as part of the server state, as is not uncommon in the authenticated dictionaries literature.

Definition 2. *A **two-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* $b \in \{0,1\}$ *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 from* $\mathsf{DB}_{\mathsf{init}}$ *and possibly modified).*

- $(\mathsf{st}_C, \mathsf{hq}_0, \mathsf{hq}_1) \leftarrow \mathsf{RequestHint}(\mathsf{pp}_0, \mathsf{pp}_1)$ *takes the servers' public parameters as input and outputs the client state* st_C *and hint queries* hq_0 *and* hq_1.
- $(\mathsf{st}'_{S_b}, \mathsf{resp}_b) \leftarrow \mathsf{AnsHintReq}(\mathsf{st}_{S_b}, \mathsf{hq}_b)$ *is executed by each server* $b \in \{0,1\}$ *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.
- $(\mathsf{st}'_C, \mathsf{hint}) \leftarrow \mathsf{VerSetup}(\mathsf{st}_C, \mathsf{resp}_0, \mathsf{resp}_1)$ *takes client state* st_C, *and server responses* resp_0 *and* resp_1. *It outputs updated client state* st'_C, *and hint* hint.
- $(\mathsf{st}'_C, \mathsf{qry}_0, \mathsf{qry}_1) \leftarrow \mathsf{Query}(\mathsf{st}_C, \mathsf{idx})$ *takes client state* st_C *and index* $\mathsf{idx} \in [N]$. *It returns updated client state* st'_C *and queries* qry_0, qry_1.
- $(\mathsf{st}'_{S_b}, \mathsf{ans}_b) \leftarrow \mathsf{Answer}(\mathsf{st}_{S_b}, \mathsf{qry}_b)$ *is executed by each server* $b \in \{0,1\}$; *it takes server state* st_{S_b} *and query* qry_b *and returns server state* st'_{S_b} *and answer* ans_b.
- $(\mathsf{st}'_C, x, \mathsf{hint}') \leftarrow \mathsf{Recon}(\mathsf{st}_C, \mathsf{hint}, \mathsf{ans}_0, \mathsf{ans}_1)$ *takes as input client state* st_C, *hint* hint, *and answers* ans_0, ans_1. *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 two-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 server state* st'_{S_b}, *updated public parameters* pp'_b, *and update information* U_b.
- $(\mathsf{st}'_C, \mathsf{hint}') \leftarrow \mathsf{UpdateHint}(\mathsf{st}_C, \mathsf{hint}, \mathsf{pp}_0, \mathsf{pp}_1, U_0, U_b)$ *takes as input the client state* st_C, *hint* hint, *and from each server the parameters* $\mathsf{pp}_0, \mathsf{pp}_1$ *and update information* U_0, U_0. *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 below.

3.1 Threat Model

Standard multi-server PIR schemes do not provide integrity of the responses and, as such, cannot guarantee correctness if even one of the servers is malicious. In this work, we consider a stronger threat model in which privacy and integrity are guaranteed so long as one of the two servers is honest. We assume that the two servers are non-colluding, as is standard in the PIR literature. The malicious server may deviate from the protocol at any phase of the protocol and may serve incorrect responses or "garbage" to the client.

We further assume that both servers have the same copy of the public database and that both servers are able to produce the same commitment to the database (i.e., using a VC scheme). Depending on the VC scheme used to

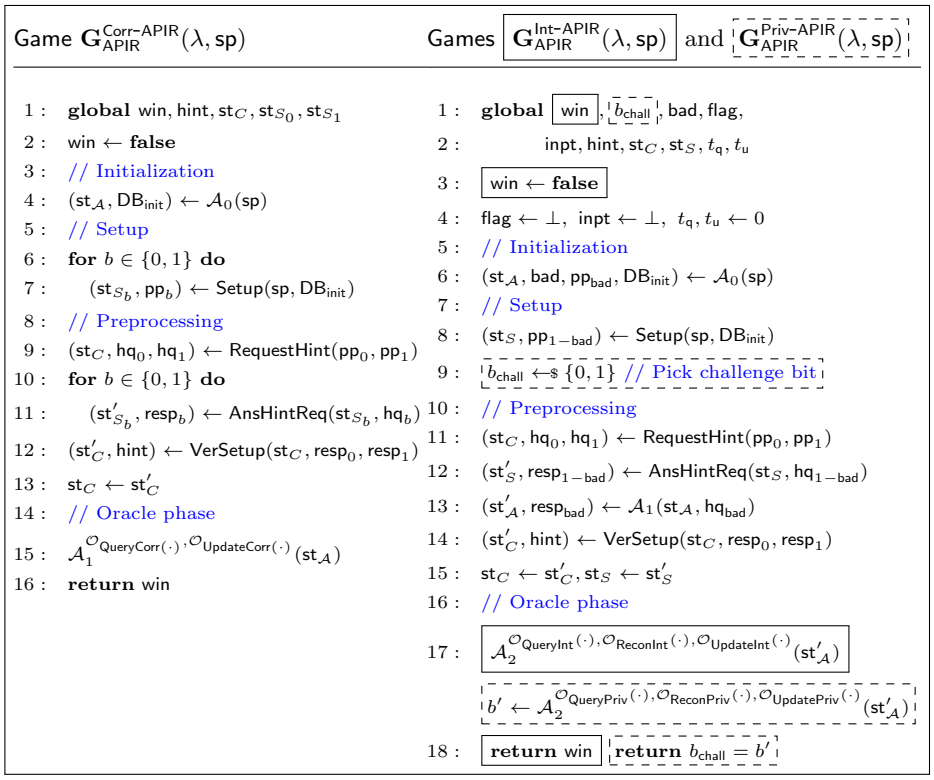

Fig. 1: The games for APIR correctness (left), integrity (right, solid box), and privacy (right, dashed box). The corresponding oracles are defined in Fig. 2.

instantiate our APIR scheme, we may also require that the public parameters are honestly generated (e.g., through a trusted setup, as in Pointproofs [30]).

We note that if both servers act maliciously, they could coincidentally modify the database in the same way, potentially causing the client to accept incorrect information. Therefore, the size and number of records must be large enough to make this probability negligibly small. Although this threat model would be stronger, it is also less generic; for this reason, we only mention it and do not use it in our analysis.

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.

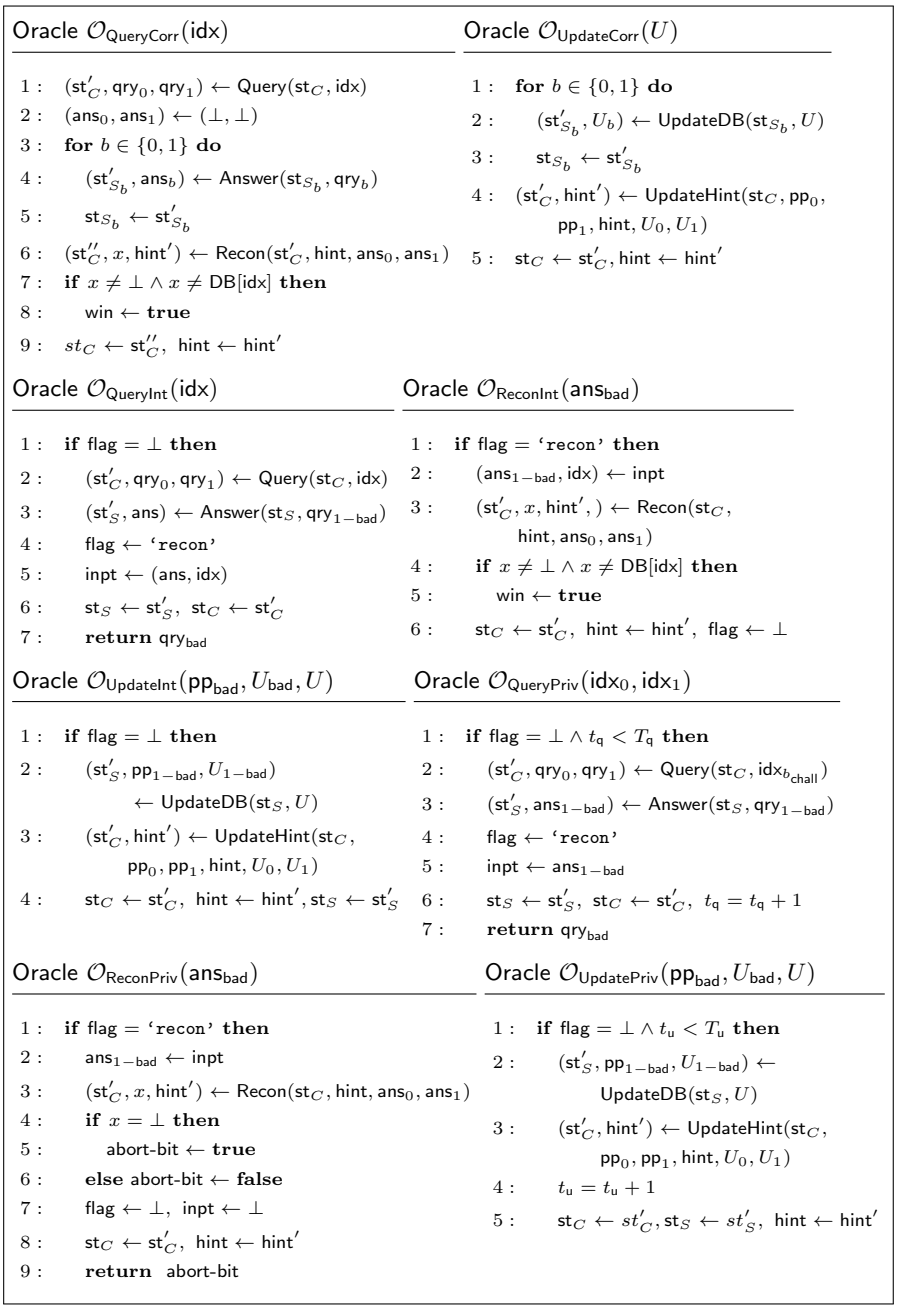

Fig. 2: Oracles for $\mathbf{G}^{\mathsf{Corr\text{-}APIR}}_{\mathsf{APIR}}$ (top), $\mathbf{G}^{\mathsf{Int\text{-}APIR}}_{\mathsf{APIR}}$ (middle), and $\mathbf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ (bottom).

Let APIR be a (possibly updateable) two-server APIR scheme with preprocessing. We assume all adversary outputs to be well-formed and within the correct domain. The oracles used in our correctness, integrity, and privacy games can be found in Fig. 2. We assume an efficient adversary that makes at most T_{q} calls to the Query Oracle and at most T_{u} calls to the update oracle, s.t. $T_{\mathsf{q}} + T_{\mathsf{u}} \in \mathsf{poly}(\lambda)$.

Correctness. An APIR scheme is correct if, when a client requests the record at index idx, it receives the correct value stored at that database index, i.e., $\mathsf{DB}[\mathsf{idx}]$. Correctness holds only when both the client and the servers behave honestly.

Definition 4. *Let* APIR *be an updatable two-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* $\mathbf{G}^{\mathsf{Corr\text{-}APIR}}_{\mathsf{APIR}}$ *be the correctness game for* APIR *defined in Figures 1 and 2. The advantage of an adversary* $\mathcal{A}$ *playing this game is defined as*

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

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

Integrity. We define integrity against adaptive queries (and updates) for APIR schemes. Let an honest client interact with two non-colluding servers, of which at most one may be malicious. 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 two-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* $\mathbf{G}^{\mathsf{Int\text{-}APIR}}_{\mathsf{APIR}}$ *be the adaptive integrity game for* APIR *defined in Figures 1 and 2. The advantage of an adversary* $\mathcal{A}$ *playing this game is defined as*

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

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* $\mathcal{A}$*.*

Privacy. An APIR scheme is private in the presence of non-colluding servers if neither server learns any information about the index or record queried by the client, even when one server is malicious. This guarantee must hold even if the servers learn whether the client outputs the error symbol $\perp$ and aborts during reconstruction. This stronger notion of privacy was introduced in [15] 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 $\mathcal{O}_{\mathsf{QueryPriv}}$ 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 two-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* $\mathbf{G}^{\mathsf{Priv\text{-}APIR}}_{\mathsf{APIR}}$ *be the privacy game against adaptive queries (and updates) for* APIR *defined in Figures 1 and 2. The advantage of an adversary* $\mathcal{A}$ *playing this game is defined as*

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

APIR *achieves* ***privacy*** *against adaptive queries if* $\mathsf{Adv}^{\mathsf{priv}}_{\mathcal{A},\mathsf{APIR}}(\lambda, T_{\mathsf{q}}, T_{\mathsf{u}}) \leq \mathsf{negl}(\lambda)$ *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 [42]. We then outline the challenges of and our solutions for upgrading SinglePass to achieve malicious security.

4.1 An Overview of SinglePass

To achieve privacy in SinglePass, the two servers are assigned distinct roles. Server 0 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 1 is responsible for supplying the database 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 should not have access to the database values used to reconstruct the queried record. Otherwise, it could correlate the hint with the response and thereby infer the queried index, violating query privacy.

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. See Fig. 3a for a database partition for $N = 12$ and $Q = 3$.

Offline phase. The offline phase starts with Server 0 sampling a permutation σ_q for each partition DB_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 $\mathsf{qry}_1 = [\sigma_q(\mathsf{ind}) \ : \ q \in [Q]]$. Next, the client replaces $\mathsf{qry}_1[m^*]$ 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 $\sigma_2(\mathsf{ind}) = 3$ and sets $\mathsf{qry}_1 = [\sigma_1(1), r, \sigma_3(1)] = [2, r, 3]$ where $r \leftarrow\!\!\$\ [4]$ is random.

In parallel to computing array qry_1, the client also samples $r_1, ..., r_Q \leftarrow\!\!\$\ [M]^Q$, and sets $\mathsf{qry}_0 = [\sigma_q(r_q) \ : \ q \in [Q]]$. The arrays qry_0 and qry_1 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 1, $[\mathsf{DB}_q[\mathsf{qry}_1[q]] \ : \ q \in [Q]]$, the client can recover the desired record as follows:

$$\mathsf{DB}[\mathsf{idx}] = \mathsf{DB}_{q^*}[m^*] = \left(\oplus_{\substack{q \in [Q] \\ q \neq q^*}} \mathsf{DB}_q[\mathsf{qry}_1[q]] \right) \oplus h_{\mathsf{ind}}.$$

Finally, the client takes the answer from Server 0, $[\mathsf{DB}_q[\mathsf{qry}_0[q]] \ : \ q \in [Q]]$, and updates the hint, such that $q \neq q^*$ for $q \in [Q]$:

$$h_{\mathsf{ind}} = h_{\mathsf{ind}} \oplus \mathsf{DB}[\mathsf{qry}_0[q]] \oplus \mathsf{DB}[\mathsf{qry}_1[q]]$$
$$h_{r_q} = h_{r_q} \oplus \mathsf{DB}[\mathsf{qry}_0[q]] \oplus \mathsf{DB}[\mathsf{qry}_1[q]].$$

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. The index to be queried is replaced 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. This is stated and proven with the Show-and-Shuffle theorem [42] which we restate in Lemma 2 in Appendix A.1.

4.2 Upgrading to Malicious Security

We now outline the challenges 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. Under malicious security, however, the server cannot be trusted to compute this hint correctly. Our key idea is to make hint generation both private and verifiable. While this could be achieved using heavy cryptographic tools that still incur large communication costs (e.g., distributed multi-point functions [6,39] or custom MPC), we prioritize computational efficiency. We instead assume a streaming model where both servers stream the database to the client. The client checks consistency between the two streams and computes the hint

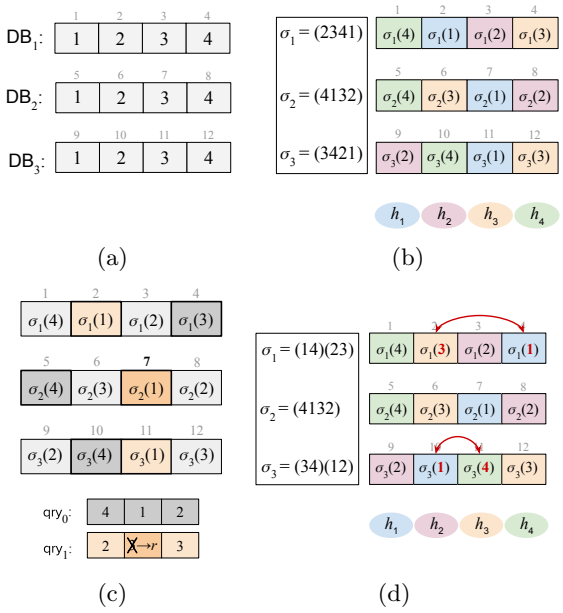

Fig. 3: An example of SinglePass [42] for a database DB with $N = 12$ records, and $Q = 3$ partitions of size $M = 4$. (a) Partitions of DB with database indices above and partition indices within the boxes. (b) Permutation and hints. The m-th hint is computed by XOR-ing the records at indices $\sigma_q(m)$ for $q \in [3]$ (records contributing to the same hint are highlighted in the same color). (c) The queries for retrieving $\mathsf{DB}[7] = \mathsf{DB}_2[3]$ are $\mathsf{qry}_1 = [\sigma_1(3), r^*, \sigma_3(3)]$ where $r^* \leftarrow\!\$ [4]$ and qry_0 is randomly sampled. (d) The updated permutation and hints obtained by swapping the mappings and hints, as indicated by the arrows.

locally. Streaming—a standard PIR technique [50,25,26,4]—reveals nothing to the servers and significantly reduces their computation compared to SinglePass.

Challenge 2: Preventing selective failure. Earlier multi-server APIR schemes [15] give servers symmetric roles, but in client-preprocessing schemes one server typically generates and refreshes the hint while the other only answers queries. To protect against a malicious server in the online phase, we employ vector commitments [10], a generalization of Merkle trees and Pedersen commitments, both of which have been used in prior [9,15,19,22]. All records in the servers' answer are verified, instead of verifying only the reconstructed record that was queried. If even one verification fails, the client aborts. This ensures that no information is leaked to the servers through the abort. To protect against a malicious server in the offline phase, we require both servers to participate identically in setup. This symmetry prevents either server from gaining an advantage in violating privacy. We adopt the stream-and-compare approach, though alternatives such as zero-knowledge proofs are possible future directions. This requirement could be weakened with a trusted data source making the database and its honestly generated digest publicly available.

Challenge 3: Proving updates secure. Another challenge is proving integrity and privacy under adaptive lookup and update queries. The original SinglePass work [42] only proved security against non-adaptive queries to a static database. We extend this by introducing new game-based definitions for updateable APIR with preprocessing. Our security games (see Fig. 1) give the adversary access to query, reconstruction, and update oracles, with the ability to interleave queries adaptively. Care is taken to avoid trivial wins/losses and only allow calls to the reconstruction oracle immediately after a call to the query oracle.

5 The Tapir Scheme

Our scheme proceeds in two phases: an *offline* phase, where the client preprocesses the database, and 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 that to updatable databases (§ 5.2). Lastly, § 5.3 states the correctness, integrity, and security of Tapir. In § 6.5, we discuss the application of Tapir to key transparency.

We give the pseudocode for the offline and online phase of our APIR scheme in Figs. 4 and 5.

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 continue in any actual retrievals or computation; hence, the protocol is effectively aborted.

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 and enables a trade-off between communication cost and computation time. The setup parameters also include any values needed to parameterize the underlying VC scheme, which we leave implicit.

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

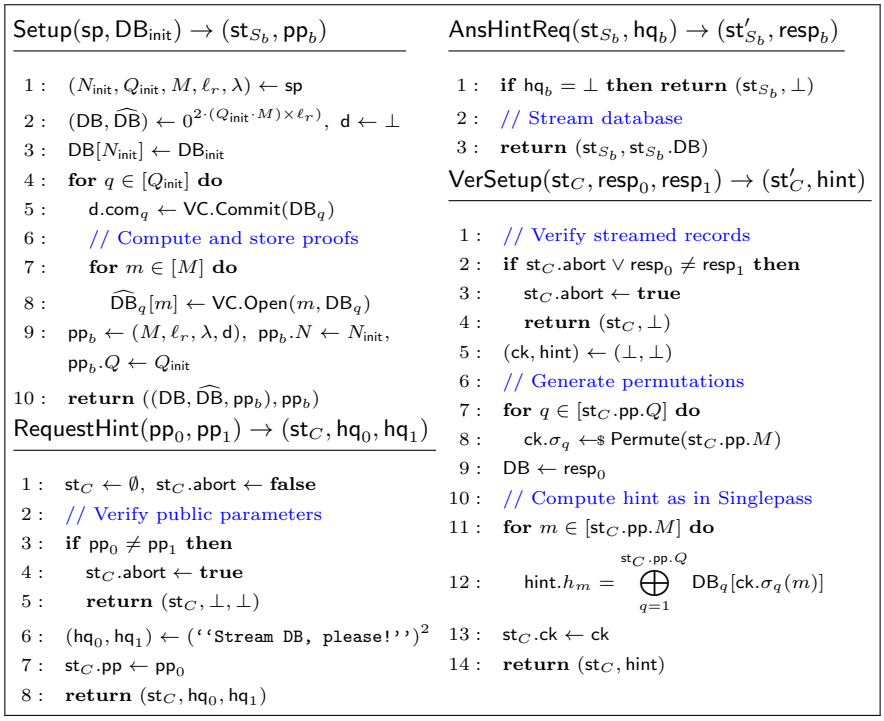

Fig. 4: The offline phase of our two-server APIR scheme.

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 $\mathsf{RequestHint}$, during which it requests two things from the servers: (1) the public parameters and the database digest, and (2) a streamed copy of the database from which it can compute a SinglePass-like hint. It first initializes the abort bit $\mathsf{st}_C.\mathsf{abort} \leftarrow \mathbf{false}$. To obtain (1), the client downloads the public parameters pp_0 and pp_1 from Servers 0 and 1, respectively, and checks if $\mathsf{pp}_0 \neq \mathsf{pp}_1$. Since at least one server is honest, equality guarantees that the digest was generated correctly. 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 a hint query hq_b, Server b executes $\mathsf{AnsHintReq}$. 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 $\sigma_1, ..., \sigma_Q : [M] \to [M]$ (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

$\mathsf{Query}(\mathsf{st}_C, \mathsf{idx}) \to (\mathsf{st}'_C, \mathsf{qry}_0, \mathsf{qry}_1)$

1 : **if** $\mathsf{st}_C.\mathsf{abort}$ **then return** $(\mathsf{st}_C, \bot, \bot)$
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 : $(\mathsf{qry}_0,\ \mathsf{qry}_1) \leftarrow (\bot, \bot)$
8 : **for** $q \in [\mathsf{pp}.Q]$ **do**
9 : $\quad \mathsf{qry}_0.\mathsf{qry}_{0,q} \leftarrow \sigma_q(r_q)$
10 : $\quad \mathsf{qry}_1.\mathsf{qry}_{1,q} \leftarrow \sigma_q(\mathsf{ind})$
11 : $\quad$ **if** $q \neq q^*$ **then**
12 : $\quad\quad$ Swap $\sigma_q(\mathsf{ind})$ and $\sigma_q(r_q)$
13 : $\quad$ **else** $\mathsf{qry}_1.\mathsf{qry}_{1,q^*} \leftarrow r^*$
14 : $\mathsf{st}_C.\mathsf{ck} \leftarrow \{\sigma_1, ..., \sigma_{\mathsf{pp}.Q}\}$
15 : $\mathsf{st}_C \leftarrow (q^*, \mathsf{ind}, \mathsf{qry}_0, \mathsf{qry}_1)$
16 : **return** $(\mathsf{st}_C, \mathsf{qry}_0, \mathsf{qry}_1)$

$\mathsf{Answer}(\mathsf{st}_{S_b}, \mathsf{qry}_b) \to (\mathsf{st}'_{S_b}, \mathsf{ans}_b)$

1 : **if** $\mathsf{qry}_b = \bot$ **then return** $(\mathsf{st}_{S_b}, \bot)$
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** $[\mathsf{DB}_q[\mathsf{qry}_{b,q}] || \widehat{\mathsf{DB}}_q[\mathsf{qry}_{b,q}] \mid q \in [Q]]$

$\mathsf{Recon}(\mathsf{st}_C, \mathsf{hint}, \mathsf{ans}_0, \mathsf{ans}_1) \to (\mathsf{st}'_C, x, \mathsf{hint}')$

1 : **if** $\mathsf{st}_C.\mathsf{abort}$ **then return** $(\mathsf{st}_C, \bot, \bot)$
2 : $\mathsf{pp} \leftarrow \mathsf{st}_C.\mathsf{pp},\ \mathsf{ind} \leftarrow \mathsf{st}_C.\mathsf{ind},\ q^* \leftarrow \mathsf{st}_C.q^*$
3 : $(\mathsf{qry}_{0,1}, ..., \mathsf{qry}_{0,\mathsf{pp}.Q}) \leftarrow \mathsf{st}_C.\mathsf{qry}_0$
4 : $(\mathsf{qry}_{1,1}, ..., \mathsf{qry}_{1,\mathsf{pp}.Q}) \leftarrow \mathsf{st}_C.\mathsf{qry}_1$
5 : $((\mathsf{rec}_{0,1}||\pi_{0,1}), ..., (\mathsf{rec}_{0,\mathsf{pp}.Q}||\pi_{0,\mathsf{pp}.Q})) \leftarrow \mathsf{ans}_0$
6 : $((\mathsf{rec}_{1,1}||\pi_{1,1}), ..., (\mathsf{rec}_{1,\mathsf{pp}.Q}||\pi_{1,\mathsf{pp}.Q})) \leftarrow \mathsf{ans}_1$
7 : $(\mathsf{com}_1, ..., \mathsf{com}_{\mathsf{pp}.Q}) \leftarrow \mathsf{pp}.\mathsf{d}$
8 : $(h_1, ..., h_{\mathsf{pp}.M}) \leftarrow \mathsf{hint}$
9 : // Verify commitments
10 : **for** $q \in [\mathsf{pp}.Q]$ **do**
11 : $\quad$ **for** $b \in \{0, 1\}$ **do**
12 : $\quad\quad v \leftarrow \mathsf{VC.Verify}(\mathsf{com}_q, \mathsf{qry}_{b,q}, \mathsf{rec}_{b,q}, \pi_{b,q})$
13 : $\quad\quad$ **if** $\neg v$ **then**
14 : $\quad\quad\quad \mathsf{st}_C.\mathsf{abort} \leftarrow$ **true**
15 : $\quad\quad\quad$ **return** $(\mathsf{st}_C, \bot, \bot)$
16 : // Recover record
17 : $x \leftarrow \left(\bigoplus_{q \in [\mathsf{pp}.Q] \setminus \{q^*\}} \mathsf{rec}_{1,q} \right) \oplus h_{\mathsf{ind}}$
18 : **for** $q \in [\mathsf{pp}.Q] \setminus \{q^*\}$ **do** // Update hint
19 : $\quad h_{\mathsf{ind}} \leftarrow h_{\mathsf{ind}} \oplus \mathsf{rec}_{0,q} \oplus \mathsf{rec}_{1,q}$
20 : $\quad h_{r_q} \leftarrow h_{r_q} \oplus \mathsf{rec}_{0,q} \oplus \mathsf{rec}_{1,q}$
21 : $\mathsf{st}_C.\mathsf{ck} \leftarrow \{\sigma_q \mid q \in [\mathsf{pp}.Q]\}$
22 : **return** $(\mathsf{st}_C, x, (h_1, ..., h_{\mathsf{pp}.M}))$

Fig. 5: The online phase of our two-server APIR scheme.

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.

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 $\bot$ 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_0 and ans_1 from the servers, the client executes Recon. If it did not abort earlier it verifies *all* the proofs output by the servers and aborts if any verification fails. If all checks pass, then the client reconstructs the requested record by combining the records in ans_1 with the hint, exactly as in SinglePass.

It updates the hint with the records in ans_0, ensuring privacy 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 [42] and restated in Fig. 10. 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, and privacy is preserved. 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 updates, specifically appends (add) and edits (edit). We refer to Fig. 6 for the pseudocode. 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.

Updating the Database. On input of the server state $\mathsf{st}_{S_b} = (\mathsf{pp}_b, \mathsf{DB}, \widehat{\mathsf{DB}})$ and a sequence of updates U, the servers first execute UpdateDB. Let $\mathsf{pp}' \leftarrow \mathsf{pp}_b$ and $\mathsf{st}'_{S_b} \leftarrow \mathsf{st}_{S_b}$. This procedure first loops through the updates U and assigns an appropriate index to new records. This allows all operations to be applied consistently since add operations to an existing database partition $q \in [\mathsf{pp}.Q]$ are the same as edits. This procedure also computes new database parameters and stores them in the updated set of parameters pp', where $\mathsf{pp}'.Q = \lceil \mathsf{pp}'.N/M \rceil$ is the new number of database partitions after the update and $\mathsf{pp}.Q$ is the number of partitions before. It iterates over each partition q in $[\mathsf{pp}'.Q]$ to apply the set of update operations U'_q. For each update $(\mathsf{op}, \mathsf{idx}, x_{\text{new}}) \in U'_q$, where op denotes the operation, idx is the index of position m in partition DB_q, x_{new} is the new value, and x_{old} is the old value, the following steps are performed:

- The database record is updated $\mathsf{DB}_q[m] \leftarrow x_{new}$.
- The partition's vector commitment is updated at index m to x_{new} if this is not a new partition, i.e., $q \leq \mathsf{pp}.Q$.
- The update information $(\mathsf{op}, \mathsf{idx}, x_{new} \oplus x_{old})$ is added to the output update list U_b. Note that this list includes the delta of the old and new values.

After the creation of a new partition, a new vector commitment com_q (of the partition) is computed and added to the digest $\mathsf{pp}'.\mathsf{d}$. Finally, UpdateDB

UpdateDB$(\mathsf{st}_{S_b}, U) \to (\mathsf{st}'_{S_b}, \mathsf{pp}'_b, U_b)$

1 : $\mathsf{pp}' \leftarrow \mathsf{st}_{S_b}.\mathsf{pp},\ \mathsf{st}'_{S_b} \leftarrow \mathsf{st}_{S_b},\ Q_{old} \leftarrow \mathsf{pp}.Q$
2 : $U', U_b \leftarrow \emptyset$
3 : **for** $(\mathsf{op}, \mathsf{idx}, x) \in U$ **do**
4 : // Add index to new records
5 : **if** $\mathsf{op} = \mathsf{add}$ **then**
6 : $\mathsf{pp}'.N = \mathsf{pp}'.N + 1$
7 : $U' \leftarrow U' \cup \{(\mathsf{op}, \mathsf{pp}'.N, x)\}$
8 : **else** $U' \leftarrow U' \cup \{(\mathsf{op}, \mathsf{idx}, x)\}$
9 : // Update number of partitions
10 : $\mathsf{pp}'.Q \leftarrow \lceil \mathsf{pp}'.N / \mathsf{pp}'.M \rceil$
11 : Extend $\mathsf{st}'_{S_b}.\mathsf{DB},\ \mathsf{st}'_{S_b}.\widehat{\mathsf{DB}}$, $\mathsf{pp}'.\mathsf{d}$ to $\mathsf{pp}'.Q$ partitions/values
12 : $(\mathsf{com}'_1, \ldots, \mathsf{com}'_{\mathsf{pp}'.Q}) \leftarrow \mathsf{pp}'.\mathsf{d}$
13 : **for** $q \in [\mathsf{pp}'.Q]$ **do**
14 : // Get update ops. for partition q
15 : $U'_q \leftarrow \{(\mathsf{op}, \mathsf{idx}, x) \in U' : \mathsf{pp}'.M \cdot (q-1) < \mathsf{idx} \le \mathsf{pp}'.M \cdot q\}$
16 : **for** $(\mathsf{op}, \mathsf{idx}, x_{new}) \in U'_q$ **do**
17 : $m \leftarrow ((\mathsf{idx} - 1) \bmod \mathsf{pp}'.M) + 1$
18 : $x_{old} \leftarrow \mathsf{st}'_{S_b}.\mathsf{DB}_q[m]$
19 : $\mathsf{st}'_{S_b}.\mathsf{DB}_q[m] \leftarrow x_{new}$
20 : // Update partition commitment
21 : **if** $q \le Q_{old}$ **then**
22 : $\mathsf{com}'_q \leftarrow \mathsf{VC.Update}(\mathsf{com}'_q, m, x_{old}, x_{new})$
23 : $U'_q \leftarrow U'_q \cup \{(\mathsf{op}, \mathsf{idx}, x_{new} \oplus x_{old})\}$
24 : // New partition commitment
25 : **if** $q > Q_{old}$ **then**
26 : $\mathsf{com}'_q \leftarrow \mathsf{VC.Commit}(\mathsf{st}'_{S_b}.\mathsf{DB}_q)$
27 : // Update opening proofs.
28 : **for** $m \in [\mathsf{pp}'.M]$ **do**
29 : $\mathsf{st}'_{S_b}.\widehat{\mathsf{DB}}_q[m] \leftarrow \mathsf{VC.Open}(m, \mathsf{st}'_{S_b}.\mathsf{DB}_q)$
30 : $U_b \leftarrow U_b \cup U'_q$
31 : $\mathsf{pp}'.\mathsf{d} \leftarrow (\mathsf{com}'_1, \ldots, \mathsf{com}'_{\mathsf{pp}'.Q}),\ \mathsf{st}'_{S_b}.\mathsf{pp} \leftarrow \mathsf{pp}'$
32 : **return** $(\mathsf{st}'_{S_b}, \mathsf{pp}', U_b)$

UpdateHint$(\mathsf{st}_C, \mathsf{hint}, \mathsf{pp}_0, \mathsf{pp}_1, U_0, U_1) \to (\mathsf{st}'_C, \mathsf{hint}')$

1 : // Verify update info
2 : **if** $\mathsf{st}_C.\mathsf{abort} \vee (\mathsf{pp}_0 \neq \mathsf{pp}_1) \vee (U_0 \neq U_1)$ **then**
3 : $\mathsf{st}_C.\mathsf{abort} \leftarrow \mathbf{true}$
4 : **return** $(\mathsf{st}_C, \perp)$
5 : $Q' \leftarrow \mathsf{st}_C.\mathsf{pp}.Q,\ \mathsf{ck}' \leftarrow \mathsf{st}_C.\mathsf{ck}$
6 : $(\mathsf{hint}_1, \ldots, \mathsf{hint}_{\mathsf{pp}_0.M}) \leftarrow \mathsf{hint}$
7 : // Generate new permutation(s)
8 : **for** $q \in [\mathsf{pp}_0.Q - \mathsf{st}_C.\mathsf{pp}.Q]$
9 : $Q' = Q' + 1$
10 : $\mathsf{ck}'.\sigma_q \leftarrow\!\!\$\ \mathsf{Permute}(\mathsf{pp}_0.M)$
11 : // Update hints
12 : **for** $(\mathsf{op}, \mathsf{idx}, x) \in U_0$ **do**
13 : $\hat{q} \leftarrow \lceil \mathsf{idx} / \mathsf{pp}_0.M \rceil$
14 : $\hat{m} \leftarrow ((\mathsf{idx} - 1) \bmod \mathsf{pp}_0.M + 1$
15 : $\mathsf{ind} \leftarrow \mathsf{ck}'.\sigma_{\hat{q}}^{-1}(\hat{m})$
16 : $\mathsf{hint}_{\mathsf{ind}} \leftarrow \mathsf{hint}_{\mathsf{ind}} \oplus x$
17 : $\mathsf{st}_C.\mathsf{pp} \leftarrow \mathsf{pp}_0,\ \mathsf{st}_C.\mathsf{ck} \leftarrow \mathsf{ck}'$
18 : **return** $(\mathsf{st}_C, (\mathsf{hint}_1, \ldots, \mathsf{hint}_{\mathsf{pp}_0.M}))$

Fig. 6: UpdateDB and UpdateHint algorithm descriptions for TAPIR

refreshes the opening proofs for all $m \in [M]$ and outputs the updated server state st'_{S_b}, public parameters pp', and update information U_b.

Updating the Hint. Updates on the client side require three steps: (i) ensuring integrity of the update information, (ii) generating new permutations for new partitions, and (iii) applying the verified updates to the hint. The procedure $\mathsf{UpdateHint}$ takes as input the client state st_C, the current hint hint, the updated public parameters pp_0 and pp_1, and the update information U_0 and U_1 from the servers, and then executes the following.

Step (i) verifies that the new public parameters and update information were not maliciously generated. The client checks if $\mathsf{pp}_0 \neq \mathsf{pp}_1$ or $U_0 \neq U_1$, and if yes, sets the abort bit $\mathsf{st}_C.\mathsf{abort}$ to **true** and returns $(\mathsf{st}_C, \perp)$. Since at least one server is assumed to be honest, equality guarantees correctness. Step (ii) requires generating new permutations using $\mathsf{Permute}$ for any new database partitions that were added during $\mathsf{UpdateDB}$ and then adding them to the client key $\mathsf{st}_C.\mathsf{ck}$. Step (iii) updates the hint with the verified update information, ensuring correctness of future lookup queries. For each update $(\mathsf{op}, \mathsf{idx}, x) \in U_0$, the client uses its secret permutations $\{\sigma_q\}_q$ to determine the index in the hint to update. 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 updates $\mathsf{hint}_{\mathsf{ind}} \leftarrow \mathsf{hint}_{\mathsf{ind}} \oplus x$. Finally, the client outputs the updated state st_C (containing the new public parameters) and the updated hint.

5.3 Scheme Correctness and Security

Theorem 1. *Let* VC *be a vector commitment scheme with binding such that* $\mathsf{VC.Commit}$ *is deterministic. The updateable two-server APIR scheme with preprocessing described in Figs. 4, 5, and 6 satisfies correctness (Def. 4).*

Theorem 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* $\mathsf{VC.Commit}$ *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 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* $\mathsf{VC.Commit}$ *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 for correctness (Theorem 1), integrity (Theorem 2), and privacy (Theorem 3) for our scheme are given in the full version of this paper [23].

6 Evaluation

This section evaluates the performance of Tapir and related work. We implement our scheme with both Pointproofs [30] and Merkle trees as the vector commitment scheme (we refer to these implementations as TAPIR-PP and TAPIR-MT, respectively). We compare our approach against the two-server APIR schemes of Colombo et al. [15], 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 [42] to demonstrate the overhead incurred by making it maliciously secure.

Our protocol is implemented in Go without multi-threading and makes use of existing libraries for Merkle trees[1], and Pointproofs[2], which we adapt as needed. 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. 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.

We use initial database sizes of $N \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}, 2^{24}\}$, records of size 32 B, and, where applicable, Q partitions of size $M = \sqrt{N}$. 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 online phase measurements 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 comes from the servers' digest generation and the need to stream the database from both servers to ensure

[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`

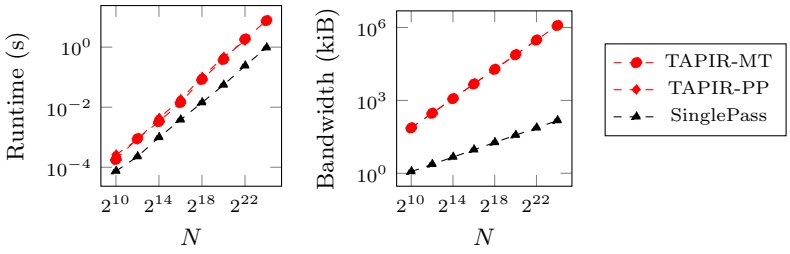

Fig. 7: Offline per-client performance comparison of Tapir and the *unauthenticated* base scheme SinglePass. Results shown for databases with N records of size 32 B. All axes use logarithmic scaling.

the integrity. In the online phase, the increase in bandwidth comes from the retrieved records' proofs that are sent together with the records, and the increase in runtime comes 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 larger 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 reports online costs for SinglePass, Tapir, and three related APIR schemes. The results align with expectations: Tapir introduces only modest bandwidth overhead compared to SinglePass—just 0.11 % for TAPIR-PP ($N = 2^{22}$) and up to 13.11× for TAPIR-MT ($N = 2^{24}$). TAPIR-PP significantly outperforms TAPIR-MT, as Pointproofs optimize bandwidth and verification cost via proof aggregation. This optimization yields communication nearly identical to SinglePass. The additional overhead in Tapir stems solely from proofs included in the answer: Pointproofs proofs are compact (one group element), whereas Merkle proofs scale with path length $O(\log_2 M)$.

Both Tapir variants incur higher online runtimes than SinglePass, with TAPIR-MT outperforming TAPIR-PP. The higher cost of TAPIR-PP stems from group exponentiations and pairings required for proof aggregation and verification, whereas TAPIR-MT only requires $Q \log_2 M$ hash evaluations, which are cheap. For $N = 2^{24}$, TAPIR-MT is 22.64× slower than SinglePass overall, with just 1.37× server and 54.15× client overhead. In contrast, TAPIR-PP at $N = 2^{22}$ is 5.8 orders of magnitude slower, primarily due to Pointproofs aggregation in Answer and verification in Recon (Fig. 8).

Tapir supports a runtime and bandwidth trade-off (Fig. 5 and Tab. 1). For instance, using Merkle trees yields the smallest runtimes among multi-server APIR schemes, while using Pointproofs [30] matches the online bandwidth of SinglePass. In contrast, linear schemes like [15] show little variation across vec-

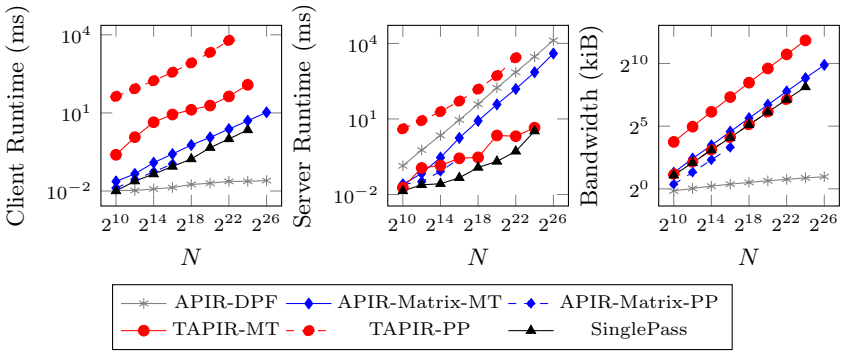

Fig. 8: Online performance of our scheme Tapir and prior work. Results shown for DBs with N records of size 32 B. All axes use logarithmic scaling.

tor commitments. 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.

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, and APIR-Matrix-PP [15].

Linear PIR schemes with VCs, like APIR-Matrix-MT and APIR-Matrix-PP, require servers to generate digests verified by the client offline. Tapir also incurs digest and proof costs, however, by applying the vector commitments over each database partition 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 contrast, APIR-DPF does not require any preprocessing and the APIR-Matrix schemes require only digest equality checks. In the online phase, we expect TAPIR-MT to have lower runtimes than all other APIR schemes for larger database sizes due to the preprocessing and fast verification of proofs. Table 1 reports benchmarking results, and Fig. 8 shows online costs.

For online bandwidth, Merkle tree schemes are more costly than Pointproofs due to logarithmic proof sizes. TAPIR-PP and SinglePass have nearly identical costs, with only minimal overhead introduced by Pointproofs. APIR-MATRIX schemes use less bandwidth, sending only one record and one proof, whereas Tapir transmits Q records and proofs. APIR-DPF achieves the lowest bandwidth at the cost of server runtime linear in the database size.

Client runtime is fastest for APIR-DPF, even outperforming unauthenticated SinglePass, while APIR-Matrix-MT and APIR-Matrix-PP are similar, and Tapir incurs the highest overhead due to the more complex Recon algorithm versus the cheap XOR operations in APIR-MATRIX. Server runtime is lowest for TAPIR-MT, with only 1.37× overhead compared to its unauthenticated

N	(A)PIR	Offline BW [kiB]	Offline RT [s] (1-Time)	Offline RT [s] (Per-Client)	Online BW [kiB]	Online 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: Performance comparison of runtime (RT) and bandwidth (BW) of (A)PIR schemes for record size 32 B. Offline cost is split into one-time server cost and per-client cost. The best APIR performance for each database size is highlighted in gray. Results for TAPIR-PP ($N = 2^{24}$) and APIR-Matrix-PP are omitted due to high Pointproofs setup costs.

base. APIR-DPF and APIR-MATRIX scale linearly with N, whereas TAPIR is sublinear. For $N \geq 2^{20}$, TAPIR-MT is the fastest online, outperforming APIR-Matrix-MT and APIR-DPF by up to 5.83× and 23.82×, respectively.

We note that TAPIR is highly parallelizable, hence we expect lower runtimes in practice.

6.4 Database Updates

To evaluate updates, we measured the bandwidth and runtime cost of applying a batch of 500 update operations to unique 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 a partition to be equally fast, as they are handled identically. Client runtime and bandwidth should be independent of the VC scheme. In contrast, server runtime should be higher for Pointproofs than for Merkle trees, reflecting the greater cost of generating and updating commitments and proofs. Bandwidth should be similar across update types,

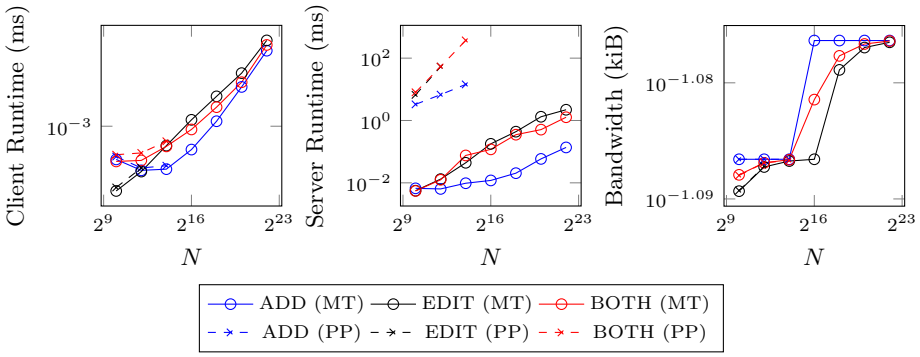

Fig. 9: Amortized update costs for TAPIR with Merkle trees (MT) and Pointproofs (PP), comparing ADD, EDIT, and BOTH operations. Results are for DBs of N records of size 32 B and batch size 500. Axes use logarithmic scaling.

except when new partitions are created: ADD always incurs this cost, BOTH on average only half as often, and EDIT never.

Client runtime is very low, since updates only involve lightweight tasks such as evaluating permutations and basic arithmetic (XOR, mod, division) to refresh the hint. Even at $N = 2^{22}$, the amortized cost is only 0.005 ms (TAPIR-MT). TAPIR-PP is slightly slower due to equality checks over bilinear group elements rather than hash outputs. Server runtime is higher but practical, as 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 slower overall due to the costly bilinear pairings of Pointproofs. EDIT is typically more expensive than ADD, since each edit may update commitments across up to 500 partitions, while additions only initialize a few new ones.

Bandwidth is modest, ranging from 83.48 B at $N = 2^{10}$ to 86.01 B at $N = 2^{22}$ (TAPIR-MT). Costs for TAPIR-MT and TAPIR-PP are nearly identical across ADD, EDIT, and BOTH, since the client receives only update tuples and a new digest (Merkle root vs. group element). ADD incurs the highest bandwidth due to new vector commitments, EDIT the smallest as no partitions are created, and BOTH in between since half the operations are additions on average. A gap appears at $N = 2^{16}$ due to partitioning: with $M = 256$, ADD needs about two new partitions, EDIT none, and BOTH one.

6.5 Application: Key Transparency

Key Transparency (KT) is widely deployed by companies such as Google [29], WhatsApp [43], Proton [28], and Keybase [35] to enable users to verify public keys for end-to-end encrypted communication, and it is even undergoing standardization [38]. 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 [11,31]). 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 even against adversarial servers.

Both WhatsApp's [47] and Proton's [28] white papers state that their KT systems use Curve25519 as the Public Identity 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. 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.

Acknowledgments. The authors thank Tianxin Tang, Christian Weinert, and 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

1. Alon, B., Beimel, A.: On the definition of malicious private information retrieval. In: The sixth Information-Theoretic Cryptography (ITC) (2025)
2. Angel, S., Setty, S.T.V.: Unobservable communication over fully untrusted infrastructure. In: OSDI. pp. 551–569. USENIX Association (2016)
3. Apple Machine Learning Research: Combining Machine Learning and Homomorphic Encryption in the Apple Ecosystem. `https://machinelearning.apple.com/research/homomorphic-encryption` (oct 2024), accessed: 26-09-2025
4. Arunachalaramanan, P.S., Ren, L.: Single-server stateful PIR with verifiability and balanced efficiency. Cryptology ePrint Archive, Paper 2025/1055 (2025), `https://eprint.iacr.org/2025/1055`
5. Asi, H., Boemer, F., Genise, N., Mughees, M.H., Ogilvie, T., Rishi, R., Rothblum, G.N., Talwar, K., Tarbe, K., Zhu, R., Zuliani, M.: Scalable Private Search with Wally. CoRR **abs/2406.06761** (2024)
6. Boyle, E., Gilboa, N., Hamilis, M., Ishai, Y., Tu, Y.: Improved Constructions for Distributed Multi-Point Functions . In: 2025 IEEE Symposium on Security and Privacy. pp. 2414–2432. IEEE Computer Society Press (May 2025). `https://doi.org/10.1109/SP61157.2025.00044`
7. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part II. LNCS, vol. 9057, pp. 337–367. Springer, Berlin, Heidelberg (Apr 2015). `https://doi.org/10.1007/978-3-662-46803-6_12`
8. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing: Improvements and extensions. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S.

(eds.) ACM CCS 2016. pp. 1292–1303. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978429
9. de Castro, L., Lee, K.: VeriSimplePIR: Verifiability in SimplePIR at no online cost for honest servers. In: Balzarotti, D., Xu, W. (eds.) USENIX Security 2024. USENIX Association (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/de-castro
10. Catalano, D., Fiore, D.: Vector commitments and their applications. In: Kurosawa, K., Hanaoka, G. (eds.) PKC 2013. LNCS, vol. 7778, pp. 55–72. Springer, Berlin, Heidelberg (Feb / Mar 2013). https://doi.org/10.1007/978-3-642-36362-7_5
11. Celi, S., Davidson, A.: Call me by my name: Simple, practical private information retrieval for keyword queries. In: Luo, B., Liao, X., Xu, J., Kirda, E., Lie, D. (eds.) ACM CCS 2024. pp. 4107–4121. ACM Press (Oct 2024). https://doi.org/10.1145/3658644.3670271
12. Chase, M., Deshpande, A., Ghosh, E., Malvai, H.: SEEMless: Secure end-to-end encrypted messaging with less trust. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 1639–1656. ACM Press (Nov 2019). https://doi.org/10.1145/3319535.3363202
13. Cheng, R., Scott, W., Masserova, E., Zhang, I., Goyal, V., Anderson, T.E., Krishnamurthy, A., Parno, B.: Talek: Private group messaging with hidden access patterns. In: ACSAC. pp. 84–99. ACM (2020)
14. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: 36th FOCS. pp. 41–50. IEEE Computer Society Press (Oct 1995). https://doi.org/10.1109/SFCS.1995.492461
15. Colombo, S., Nikitin, K., Corrigan-Gibbs, H., Wu, D.J., Ford, B.: Authenticated private information retrieval. In: Calandrino, J.A., Troncoso, C. (eds.) USENIX Security 2023. pp. 3835–3851. USENIX Association (Aug 2023), https://www.usenix.org/conference/usenixsecurity23/presentation/colombo
16. Corrigan-Gibbs, H., Henzinger, A., Kogan, D.: Single-server private information retrieval with sublinear amortized time. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, vol. 13276, pp. 3–33. Springer, Cham (May / Jun 2022). https://doi.org/10.1007/978-3-031-07085-3_1
17. Corrigan-Gibbs, H., Kogan, D.: Private information retrieval with sublinear online time. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part I. LNCS, vol. 12105, pp. 44–75. Springer, Cham (May 2020). https://doi.org/10.1007/978-3-030-45721-1_3
18. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Berlin, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-32009-5_38
19. Dietz, M., Tessaro, S.: Fully malicious authenticated PIR. In: Reyzin, L., Stebila, D. (eds.) CRYPTO 2024, Part IX. LNCS, vol. 14928, pp. 113–147. Springer, Cham (Aug 2024). https://doi.org/10.1007/978-3-031-68400-5_4
20. Durstenfeld, R.: Algorithm 235: Random permutation. Commun. ACM **7**(7), 420 (Jul 1964). https://doi.org/10.1145/364520.364540, https://doi.org/10.1145/364520.364540
21. Eriguchi, R., Kurosawa, K., Nuida, K.: On the optimal communication complexity of error-correcting multi-server PIR. In: Kiltz, E., Vaikuntanathan, V. (eds.) TCC 2022, Part III. LNCS, vol. 13749, pp. 60–88. Springer, Cham (Nov 2022). https://doi.org/10.1007/978-3-031-22368-6_3
22. Falk, B.H., Mishra, P., Shtepel, M.: Malicious security for PIR (almost) for free. In: CRYPTO 2025, Part I. LNCS, Springer, Cham (Aug 2025)

23. Falzon, F., Hetz, L., O'Toole, A.: TAPIR: A two-server authenticated PIR scheme with preprocessing. Cryptology ePrint Archive, Paper 2025/2177 (2025), https://eprint.iacr.org/2025/2177
24. Fisher, R.A., Yates, F.: Statistical tables for biological, agricultural and medical research. Hafner Publishing Company (1953)
25. Ghoshal, A., Zhou, M., Shi, E.: Efficient pre-processing PIR without public-key cryptography. In: Joye, M., Leander, G. (eds.) EUROCRYPT 2024, Part VI. LNCS, vol. 14656, pp. 210–240. Springer, Cham (May 2024). https://doi.org/10.1007/978-3-031-58751-1_8
26. Ghoshal, A., Zhou, M., Shi, E., Peng, B.: Pseudorandom functions with weak programming privacy and applications to private information retrieval. In: Fehr, S., Fouque, P.A. (eds.) EUROCRYPT 2025, Part VII. LNCS, vol. 15607, pp. 284–313. Springer, Cham (May 2025). https://doi.org/10.1007/978-3-031-91098-2_11
27. Gilboa, N., Ishai, Y.: Distributed point functions and their applications. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 640–658. Springer, Berlin, Heidelberg (May 2014). https://doi.org/10.1007/978-3-642-55220-5_35
28. Göbel, T., Huigens, D.: Proton Key Transparency Whitepaper. https://proton.me/files/proton_keytransparency_whitepaper.pdf (2024), accessed: 26-09-2025
29. Google, Inc.: Key Transparency. https://github.com/google/keytransparency (2024), accessed: 26-09-2025
30. Gorbunov, S., Reyzin, L., Wee, H., Zhang, Z.: Pointproofs: Aggregating proofs for multiple vector commitments. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020. pp. 2007–2023. ACM Press (Nov 2020). https://doi.org/10.1145/3372297.3417244
31. Hao, M., Liu, W., Peng, L., Zhang, C., Wu, P., Zhang, L., Li, H., Deng, R.H.: Practical keyword private information retrieval from key-to-index mappings. In: USENIX Security 2025. USENIX Association (Aug 2025)
32. Henzinger, A., Dauterman, E., Corrigan-Gibbs, H., Zeldovich, N.: Private Web Search with Tiptoe. In: SOSP. pp. 396–416. ACM (2023)
33. Henzinger, A., Hong, M.M., Corrigan-Gibbs, H., Meiklejohn, S., Vaikuntanathan, V.: One server for the price of two: Simple and fast single-server private information retrieval. In: Calandrino, J.A., Troncoso, C. (eds.) USENIX Security 2023. pp. 3889–3905. USENIX Association (Aug 2023), https://www.usenix.org/conference/usenixsecurity23/presentation/henzinger
34. Hu, Y., Hooshmand, K., Kalidhindi, H., Yang, S.J., Popa, R.A.: Merkle^2: A low-latency transparency log system. In: 2021 IEEE Symposium on Security and Privacy. pp. 285–303. IEEE Computer Society Press (May 2021). https://doi.org/10.1109/SP40001.2021.00088
35. keybase.io: Keybase Chat. book.keybase.io/docs/chat (2022), accessed: 26-09-2025
36. Knuth, D.E.: The art of computer programming, vol. 3. Pearson Education (1997)
37. Kogan, D., Corrigan-Gibbs, H.: Private blocklist lookups with checklist. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021. pp. 875–892. USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/kogan
38. KT Internet Engineering Task Force Working Group: Key Transparency (keytrans) . https://datatracker.ietf.org/wg/keytrans/about/ (2025), accessed: 26-09-2025

39. Külaots, E., Krips, T., Eerikson, H., Pullonen-Raudvere, P.: SLAMP-FSS: Two-party multi-point function secret sharing from simple linear algebra. Cryptology ePrint Archive, Report 2024/1394 (2024), https://eprint.iacr.org/2024/1394
40. Kushilevitz, E., Ostrovsky, R.: Replication is NOT needed: SINGLE database, computationally-private information retrieval. In: 38th FOCS. pp. 364–373. IEEE Computer Society Press (Oct 1997). https://doi.org/10.1109/SFCS.1997.646125
41. Lazzaretti, A., Papamanthou, C.: TreePIR: Sublinear-time and polylog-bandwidth private information retrieval from DDH. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part II. LNCS, vol. 14082, pp. 284–314. Springer, Cham (Aug 2023). https://doi.org/10.1007/978-3-031-38545-2_10
42. Lazzaretti, A., Papamanthou, C.: Single pass client-preprocessing private information retrieval. In: Balzarotti, D., Xu, W. (eds.) USENIX Security 2024. USENIX Association (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/lazzaretti
43. Lewi, K., Lawlor, S.: Introducing Auditable Key Transparency for End-to-End Encrypted Messaging. https://engineering.fb.com/2023/04/13/security/whatsapp-key-transparency/ (Apr 2023), accessed: 26-09-2025
44. Liu, Z., Tromer, E., Wang, Y.: PerfOMR: Oblivious message retrieval with reduced communication and computation. In: Balzarotti, D., Xu, W. (eds.) USENIX Security 2024. USENIX Association (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/liu-zeyu
45. Malvai, H., Kokoris-Kogias, L., Sonnino, A., Ghosh, E., Oztürk, E., Lewi, K., Lawlor, S.F.: Parakeet: Practical key transparency for end-to-end encrypted messaging. In: NDSS 2023. The Internet Society (Feb 2023)
46. Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: CONIKS: Bringing key transparency to end users. In: Jung, J., Holz, T. (eds.) USENIX Security 2015. pp. 383–398. USENIX Association (Aug 2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara
47. Meta: WhatsApp Key Transparency Overview. https://www.whatsapp.com/security/WhatsApp-Key-Transparency-Whitepaper.pdf (2023), accessed: 26-09-2025
48. Tomescu, A., Bhupatiraju, V., Papadopoulos, D., Papamanthou, C., Triandopoulos, N., Devadas, S.: Transparency logs via append-only authenticated dictionaries. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 1299–1316. ACM Press (Nov 2019). https://doi.org/10.1145/3319535.3345652
49. Wang, Y., Liu, X., Zhang, J., Liu, J., Yang, X.: Crust: Verifiable and efficient private information retrieval with sublinear online time. Cryptology ePrint Archive, Report 2023/1607 (2023), https://eprint.iacr.org/2023/1607
50. Zhou, M., Park, A., Zheng, W., Shi, E.: Piano: Extremely simple, single-server PIR with sublinear server computation. In: 2024 IEEE Symposium on Security and Privacy. pp. 4296–4314. IEEE Computer Society Press (May 2024). https://doi.org/10.1109/SP54263.2024.00055

A Additional Preliminaries

A.1 Show-and-Shuffle

In SinglePass [42], 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 Lemma 2, and refer to [42, § 3.1] for the full proof. The game is parameterized over $M, Q \in \mathbb{N}$.

Game $\mathbf{G}^{\mathsf{Ind\text{-}SaS}}(M, Q)$

$(\sigma_1, ..., \sigma_Q) \leftarrow\!\!\$\ \mathsf{Permute}(M)^Q$
$(\mathsf{st}_{\mathcal{A}}, x) \leftarrow \mathcal{A}_0(M, Q)$ where $x = (q^*, m^*) \in ([Q] \times [M])$
Find $\mathsf{ind} \in [M]$, s.t. $\sigma_{q^*}(\mathsf{ind}) = m^*$
$\mathbf{v} \leftarrow \perp$
for $q \in [Q]$ **do**
  if $q \neq q^*$ **then** $\mathbf{v}.v_q \leftarrow \sigma_q(\mathsf{ind})$
  else $\mathbf{v}.v_q \leftarrow\!\!\$\ [M]$
$b \leftarrow\!\!\$\ \{0, 1\}$
$\mathcal{R}_0 = (F_1, ..., F_Q) \leftarrow\!\!\$\ \mathsf{Permute}(M)^Q$
for $q \in [Q] \wedge q \neq q^*$ **do**
  $r_q \leftarrow\!\!\$\ [M]$
  $\sigma'_q \leftarrow \sigma_q$
  Swap $\sigma'_q(\mathsf{ind})$ and $\sigma'_q(r_q)$
$\mathcal{R}_1 \leftarrow (\sigma'_1, ..., \sigma'_{q^*-1}, \sigma_{q^*}, \sigma'_{q^*+1}, ...\sigma'_Q)$
$b' \leftarrow \mathcal{A}_1(\mathsf{st}_{\mathcal{A}}, q, m, \mathbf{v}, \mathcal{R}_b)$
return $b = b'$

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

Lemma 2. *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:*

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