ThorPIR: Single Server PIR via Homomorphic Thorp Shuffles

Ben Fisch   Arthur Lazzaretti   Zeyu Liu   Charalampos Papamanthou

Yale University

Abstract

Private Information Retrieval (PIR) is a two player protocol where the client, given some query $x \in [N]$, interacts with the server, which holds a N-bit string DB, in order to privately retrieve $\mathsf{DB}[x]$. In this work, we focus on the single-server client-preprocessing model, initially proposed by Corrigan-Gibbs and Kogan (EUROCRYPT 2020), where the client and server first run a joint preprocessing algorithm, after which the client can retrieve elements from DB privately in time sublinear in N. Most known constructions of single-server client-preprocessing PIR follow one of two paradigms: They feature either (1) a linear-bandwidth offline phase where the client downloads the whole database from the server, or (2) a sublinear-bandwidth offline phase where however the server has to compute a large-depth ($\Omega_\lambda(N)$) circuit under fully-homomorphic encryption (FHE) in order to execute the preprocessing phase.

In this paper, we propose ThorPIR, a single-server client preprocessing PIR scheme which achieves both sublinear offline bandwidth (asymptotically and concretely) and a low-depth, highly parallelizable preprocessing circuit. Our main insight is to use and significantly optimize the concrete circuit-depth of a much more efficient shuffling technique needed during preprocessing, called Thorp shuffle. A Thorp shuffle satisfies a weaker security property (e.g., compared to an AES permutation) which is "just enough" for our construction. We estimate that with a powerful server (e.g., hundreds of thousands of GPUs), ThorPIR's end-to-end preprocessing time is faster than any prior work. Additionally, compared to prior FHE-based works with sublinear bandwidth, our construction is at least around 10,000 times faster.

1

Contents

1 Introduction

Private information retrieval (PIR) [18, 38] allows a client to fetch a data entry from a public database without revealing any information about that entry. A major practical limitation of PIR is that fetching an entry requires $O(N)$ time, where N is the number of data entries in the database [8]—i.e., for security purposes all data entries have to be "touched". In 2020, however, in their groundbreaking work [23], Corrigan-Gibbs and Kogan introduced the notion of online/offline PIR in the 2-server model, where during an offline phase server A processes the database, generates *hints* (aggregates of a random subsets of data entries), and sends them to the client. In the online phase, the client can then use these hints to query the data entry it wants from server B. In this paradigm, the online time can be sublinear in N. Subsequent to [23], many other works have pushed our understanding of this PIR model [36, 47, 41].

One major limitation of the initial idea in [23] is that it requires two non-colluding servers. The client cannot send its queries to the same server that computed the hints or otherwise privacy can be breached. Thus, a series of works study how to achieve sublinear complexities in the single server scenario, either via streaming the database to the client and having the client compute the hints using sublinear space (e.g., [70, 53]) or via fully homomorphic encryption (FHE) (e.g., [22, 69, 40]), where the server generates the hints without learning which data entries correspond to which hints. Clearly the major disadvantage of the streaming approach is the requirement for $\Omega(N)$ communication. As such, in this work we focus on pushing the limits of the FHE approach, ensuring *the PIR bandwidth remains sublinear*.

To make progress towards this front, we focus on optimizing the circuit used to compute hints in PIR [22, 40, 69], and which will have to be executed with FHE in the case we wish to achieve sublinear bandwidth in single-server PIR. Unfortunately we observe that all such linear-size circuits have $O_\lambda(N)$ depth, which amounts to millions or even billions of levels of FHE multiplications for most if not all PIR use cases—this is clearly not practical. One exception is that in [42], the authors propose a scheme that only requires linear homomorphic encryption, requiring only one level of multiplication. However, while this is intriguing, their asymptotic offline bandwidth is $\mathsf{poly}(\lambda) \cdot N^{3/4} \cdot \mathsf{polylog}(N)$. Practically, the offline hint download is only smaller than downloading the whole database if there are at least 2^{45} entries by our estimation (and similarly for the clients' local storage). Therefore, the scheme is also not practical for most if not all PIR use cases.

In this work we therefore construct a new single-server PIR scheme, named ThorPIR, that has a preprocessing circuit that not only is shallow (due to the use of a permutation called Thorp shuffle [65]) but is also optimized to run with existing performant FHE schemes, such as SIMD-type schemes like BFV [15, 27] or BGV [14]. This is a strategy that is typically followed in other application domains as well. For example, in prior works on FHE-based machine learning, improving the algorithm efficiency completely rests upon reducing the circuit depth as well optimizing varius FHE-specific circuit parameters [61]. In summary, ThorPIR is a single-server offline/online PIR scheme that (1) achieves sublinear server time; (2) achieves sublinear offline communication (concretely much smaller than downloading the entire database); and (3) uses a linear-size, constant-depth circuit to compute the hints.

In the following, we present a summary of ThorPIR. To help with exposition, we first present a summary of ThorPIR that does not use FHE and therefore has linear bandwidth. Then we present a summary of our final FHE-based ThorPIR.

1.1 Warm-up: ThorPIR with linear bandwidth

Our starting point is the two-server PIR construction by Lazzaretti and Papamanthou (USENIX SECURITY 2024) [42] which we recall in detail in Section 2.6. In particular, the main overlap between ThorPIR and [42] is the fact that the hints are defined in exactly the same way, as we describe below.

In the preprocessing phase of the linear-bandwidth ThorPIR, the server streams the database to the client. The client splits the database DB into $K = o(N)$ partitions, $db_1, \ldots, db_K$, samples N/K permutations of $[K]$, $\tau_1, \ldots, \tau_{N/K}$, and applies τ_i to db_i for all $i \in [N/K]$. Then, the client computes and locally stores hints $h_1, \ldots, h_K$ where

$$h_j = \bigoplus_{i \in N/K} db_i[\tau_i(j)] \, .$$

As we mentioned, this is the same definition of hints as in [42], but due to the single-server, they are computed by the client streaming the database.

During the online phase, to retrieve element $(q,k) \in [N/K] \times [K]$, the client finds the positions of the other $N/K - 1$ points used to compute h_k and sends those positions to the server. The client furthermore sends $\tau_q(r)$ for some random point $r \in [K]$, so that information related to the queried element is not leaked. The server sends the queried database elements back to the client. The client then locally recovers element (q,k) by using the queried points as well as hint h_k.

Importantly, we note that in ThorPIR, as opposed to [42], our client *stores* all the database elements sent back from the server during a query: in case a subsequent query requires the use of some previously queried points, the client can use the already stored elements corresponding to answer the query, and simply perform a dummy query to the server. Due to this "caching" of server responses, every query adds to the local client storage an N/K factor, and therefore we allow ThorPIR to perform at most $o(K)$ queries to keep the client storage $o(N)$. For example, if $K = N^{2/3}$, then $N/K = N^{1/3}$, and performing $N^{1/3}$ queries keeps the client storage $N^{2/3}$. After the allowed "budget" of queries is exhausted, ThorPIR performs preprocessing from scratch, and maintains sublinear amortized complexities. (We can also deamortize ThorPIR.)

Our key insight: use of Thorp shuffle. As we mentioned above, ThorPIR performs only a sublinear number of queries before a preprocessing takes places. Therefore the server can "see" only a sublinear number of permuted elements from any given permutation τ_i. This observation allows us to use a specific and more efficient algorithm to permute the elements, the Thorp shuffle [65]. As shown in [52], a Thorp shuffle remains indistinguishable from a random permutation as long as the adversary sees only $o(N)$ permuted elements—which is exactly what we need! In addition (and due to this "weak" security property of Thorp shuffles), the Thorp shuffle can be implemented with an $O(\lambda)$-depth circuit which is ideal for our FHE setting. In particular, a Thorp shuffle has smaller depth than other common approaches used to compute permutations, such as Fisher-Yates [42], which has linear depth, or sorting networks [1, 59, 19], which have polylogarithmic depth (we expand on this on Section 8).

Improved Thorp shuffle bound. An important part of our contribution is a new analysis of the Thorp shuffle, showing that we can further reduce the depth of the circuit required to compute it, which is very important for any practical FHE implementation.

In particular, previous work [50] shows that, when running the Thorp shuffle for about $2r \log N$

Table 1: ThorPIR compared to previous state-of-the-art single-server client-preprocessing PIR schemes. Notice that schemes marked with a star have asymptotics described with big-O notation that hides polylog factors. Depth of PIANO and MIR are not included since the server just streams the database and preprocessing is run at the client.

Scheme	Preprocess				Query			Client Space	Update Time
	Depth	BW	Cli Time	Serv Time	BW	Time (both)	# queries		
PIANO [70]	N/A	$O(N)$	$O(N)$	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$\sqrt{N}$	$O(\sqrt{N})$	$O(\sqrt{N})$
MIR[53]	N/A	$O(N)$	$O(N)$	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$\sqrt{N}$	$O(\sqrt{N})$	$O(\sqrt{N})$
CHK1[22, Thm 4.1]*	$O(1)$	$\tilde{O}_\lambda(N^{3/4})$	$\tilde{O}_\lambda(N^{3/4})$	$\tilde{O}_\lambda(N)$	$\tilde{O}(N^{3/4})$	$\tilde{O}(N^{3/4})$	$N^{1/4}$	$\tilde{O}_\lambda(N^{3/4})$	$\tilde{O}_\lambda(N^{3/4})$
CHK2[22, Thm 5.1]*	$\tilde{O}_\lambda(N)$	$\tilde{O}_\lambda(\sqrt{N})$	$\tilde{O}_\lambda(\sqrt{N})$	$\tilde{O}_\lambda(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$\sqrt{N}$	$\tilde{O}_\lambda(\sqrt{N})$	$\tilde{O}_\lambda(\sqrt{N})$
ZLTS[69]*	$\tilde{O}_\lambda(N)$	$\tilde{O}_\lambda(\sqrt{N})$	$\tilde{O}_\lambda(\sqrt{N})$	$\tilde{O}_\lambda(N)$	$\tilde{O}_\lambda(1)$	$\tilde{O}_\lambda(\sqrt{N})$	$\sqrt{N}$	$\tilde{O}_\lambda(\sqrt{N})$	$\tilde{O}_\lambda(\sqrt{N})$
LP[40]*	$\tilde{O}_\lambda(N)$	$\tilde{O}_\lambda(\sqrt{N})$	$\tilde{O}_\lambda(\sqrt{N})$	$\tilde{O}_\lambda(N)$	$\tilde{O}_\lambda(1)$	$\tilde{O}_\lambda(\sqrt{N})$	$\sqrt{N}$	$\tilde{O}_\lambda(\sqrt{N})$	$\tilde{O}_\lambda(\sqrt{N})$
ThorPIR	$O_\lambda(1)$	$O(N^{2/3})$	$\tilde{O}(N^{2/3})$	$O_\lambda(N)$	$O(N^{1/3})$	$O(N^{1/3})$	$N^{1/3}$	$O(N^{2/3})$	$O_\lambda(1)$

levels, for some positive integer r, any (adaptive) q-query adversary has an advantage of at most

$$\frac{2q}{r+1}\left(\frac{4q\log N}{N}\right)^{r/2}$$

for distinguishing the Thorp Shuffle from a random permutation. With our new bound we show that with again $2r\log N$ levels, the adversary has an advantage of at most

$$\frac{2q}{r+1}\left(\frac{2q\log N}{N}\right)^{r} .$$

Practically, for the same advantage (e.g., $2^{-\lambda}$), our bound reduces the depth of the permutation's computation by about $2.5\times$: this comes from the change in the exponent and the factor inside the parenthesis.

Our proof technique refines the Markov chain coupling argument shown in [52]. Specifically, the analysis (both in our work and [52]) is to define a coupling between a Thorp Shuffle process and a (carefully picked) uniform process, and show that after T Markov chain updates, the probability that the chains are not coupled is very small. If the chains are coupled, then the Thorp Shuffle is by definition indistinguishable from a uniform permutation. Although we use the same proof setup, we refine the technique by uncovering additional conditions that would result in a coupling, and then calculating the new probability that the chains are coupled after T steps.

We describe below the further challenges that we address in this paper (apart from the low-depth Thorp shuffle) in order to run the preprocessing algorithm over FHE and avoid the linear offline bandwidth.

1.2 ThorPIR with sublinear bandwidth

Our final ThorPIR protocol eliminates the linear offline bandwidth of the scheme above by having the server compute the Thorp shuffle obliviously under FHE. However, while having a Thorp shuffle with only $O(\lambda)$ depth is a good start for better FHE efficiency (as described before), there are still two important challenges that we need to address so that to achieve better efficiency for the FHE computation of the Thorp shuffle.

- *Generating randomness with FHE efficiently.* First, to run the Thorp shuffle with FHE, the client will have to send an encryption of a locally-chosen seed to be used as an input to a

PRG to generate "encrypted randomness" that will be consumed by the Thorp shuffle (Note that a PRG is essential for our application, since the number of random bits needed for the Thorp shuffle is linear and therefore the client cannot send all the encrypted randomness to the server). However, a generic PRG can be very slow when evaluated using FHE. For example, when evaluating AES using FHE, it takes around 86 seconds to generate 128 bits [68], which is almost a second per bit. This would mean that a Thorp Shuffle that needs to sample 2^{41} bits (about the amount needed for a database with 2^{33} entries) would take more than 2.5 days of computation using 128K GPUs.[1]

- *Performing swaps with FHE in Thorp shuffle efficiently.* For the best amortized efficiency, we focus on FHE schemes with the SIMD (same instruction multiple data) feature, over a finite field, such as the BFV [15, 27] or BGV [14] schemes. For such schemes, a single ciphertext encrypts D plaintext elements at the same time and every operation over the ciphertext applies to all these D encrypted elements. However, for the Thorp shuffle computation, in each round, we need to move each element $i < N/2$ to position $2i$ or $2i+1$ which means that each element within a certain packed ciphertext (packing D elements) has to be moved to a different location within that ciphertext or even potentially to a different ciphertext for one iteration of the algorithm. This is very challenging for BFV/BGV because the optimized implementations usually operate on the packing as a whole rather than each ciphertext individually.

With these two points in mind, we propose the following solutions, which may be of independent interest.

A practical FHE-friendly PRG. For our protocol, we build a new, FHE-friendly pseudorandom generator (PRG) based on the Learning With Rounding (LWR) assumption [6]. Concretely, the runtime of our new PRG is only about 0.4 ms per bit, compared to 672 ms per bit when homomorphically evaluating a general PRG like AES [68]—a 1,680× improvement! As mentioned, we focus on the BFV [13, 27] and BGV [14] homomorphic encryption schemes, which render the best amortized efficiency. As suggested in [3, Appendix A.1], BGV/BFV can be used interchangeably. Thus, later in the paper, we use BFV only for simplicity.

To build our new PRG, we first observe that BFV works best (in terms of efficiency) over $\mathbb{Z}_t$ where t is a small prime of 15-30 bits (we explain why in more detail in Section 4.1). Therefore, to fit this constraint, we make our PRG work over $\mathbb{Z}_t$ as well. A natural idea is thus to build it from a lattice assumption, as lattice assumptions can work over prime fields of such size—we use the LWR assumption. At a high level, we compute $R(As)$ where A is chosen uniformly at random from $\mathbb{Z}_t^{m \times n}$ and s is chosen uniformly at random from $\mathbb{Z}_t^n$. Also $R(\vec{x}) : \mathbb{Z}_t^m \to \mathbb{Z}_2^m$ checks if $\vec{x}[j] \in [0, \lceil t/2 \rceil]$, its output vector has the j-th element being 0, and otherwise being 1. Then, as long as $m \geq n \log t$, we obtain a PRG, since this process takes in $n \log t$ bits as a seed, and outputs m bits.[2]

The above construction gives a PRG that outputs 0 with probability $\lceil t/2 \rceil / t \approx 1/2$ for each output bit. However, this is not enough. Essentially, we need a PRG that outputs 0 with probability $1/2 + \mathsf{negl}(\lambda)$. To do this, we sample k independent A_i's in $\mathbb{Z}_t^{m \times n}$, for $i \in [k]$, for some k such that $t^{-k} = \mathsf{negl}(n)$. Then, let the output be $\vec{u} \in \mathbb{Z}_2^m$ and let $\vec{v_i} = A_i s$ for $i \in [k]$. Then, let $\vec{u}[j] := R(\vec{v_i})[j]$ where i is the smallest number in $[k]$ such that $\vec{v_i}[j] \neq 0$. If all of them are 0, $\vec{u}[j] = 0$. Output $\vec{u}[j]$. This gives a PRG that outputs 0 with probability $1/2 + t^{-k} = 1/2 + \mathsf{negl}(\lambda)$.

---

[1]Based on [56], FHE can be accelerated by about 50× using a GPU.
[2]A is a public parameter used to define the PRG.

To make our PRG even more FHE-friendly, we design an alternative $R(\vec{x})$ function, that instead maps $\vec{x}[j]$ to 0 iff $\vec{x}[j]^{(t-1)/2} = -1$, which can be computed in only $\log t - 1$ multiplications. Essentially, this function maps half of $\mathbb{Z}_t^*$ to 0 and the other half to 1. Furthermore, checking whether $\vec{v_i} = 0$ is also easy: note that $x^{t-1} = 0$ iff $x = 0$ for any $x \in \mathbb{Z}_t$. Putting all these together, our PRG only takes $k \cdot \log t$ multiplications, and the security relies on a variant of LWR with a special rounding function $R(\cdot)$ described above.

Perfoming swaps with FHE. As mentioned, performing the Thorp shuffle with BFV naively is not very efficient. However, by [24, Lemma 1], n rounds of the Thorp Shuffle are equivalent to n rounds of a butterfly network, and the update rule of the butterfly network is much more BFV-friendly. In particular, at round $k \in [\log N]$, position i is swapped with position $i + 2^{k-1}$. This is easily done with BFV since all pairs that get swapped have the same interval in terms of positions. Thus we employ that instead.

Another difficulty that arises is that even with our tightened Thorp Shuffle bound, we still require hundreds of levels of multiplication under FHE. If we directly use regular BFV without bootstrapping, the parameters to support this many levels are too large. In addition, the naive alternative, regular BFV bootstrapping is very slow (hundreds of milliseconds per element). We circumvent both by leveraging a recent advancement in relaxed BFV bootstrapping, which requires only a couple of milliseconds per element bootstrapped [46]. Although the relaxed bootstrapping only guarantees correctness when each plaintext element is in a predetermined fixed subset of the plaintext space, it works in our use case since we can simply encode the database entries within those subsets!

Asymptotic and concrete performance. In Table 1 we present a comparison of ThorPIR with previous single-server client-preprocessing PIR schemes in terms of asymptotics. Out of all the schemes, ThorPIR and CHK1 [22] are the only two that have a sublinear-depth circuit for preprocessing, making these two the most suitable for preprocessing for resource-limited clients in practice. When comparing directly to CHK1, which shares the low-depth preprocessing circuit, in terms of asymptotics, we outperform the scheme in every efficiency metric (except depth).

In concrete terms, with a powerful server (e.g., 128K GPUs), ThorPIR has a better end-to-end time compared to the previous works with linear offline bandwidth (2 hours for our construction compared to 3-4 hours for prior constructions given a 360GB database) [70, 53] and at least about 10,000× better end-to-end time compared to previous works with sublinear offline bandwidth that uses FHE as well [22, 40, 69]. Compared to CHK1 [22], our end-to-end time is more than 20x faster, since the larger asymptotic client storage paired with hidden constants/polylog factors blow up their offline bandwidth and storage to more than 10x and 5x the database size, respectively. We discuss more details and concrete performance in Section 5.

2   Preliminaries

Notation. Let $\mathsf{Bern}(p)$ for $0 \le p \le 1$ denote the Bernoulli distribution with probability p: i.e., $\mathbf{P}_{x \leftarrow \mathsf{Bern}(p)}(x = 0) = p$ and $\mathbf{P}_{x \leftarrow \mathsf{Bern}(p)}(x = 1) = 1 - p$. $[x] := \{1, \dots, x\}$ for $x \in \mathbb{Z}^+$.

2.1   Hard Problems

Definition 2.1 (Decisional learning with rounding problem)**.** Let n, q, p be parameters dependent on λ, and $R : \mathbb{Z}_q \to \mathbb{Z}_p$ be a function. The learning with rounding (LWR) problem $\mathsf{LWR}_{n,q,p,R}$

states the following: for any $m = \mathsf{poly}(\lambda)$, distinguish $(A, R(\vec{s}A))$ and $(A, R(\vec{b}))$ (with noticeable advantage), where $A \leftarrow_\$ \mathbb{Z}_q^{n \times m}, \vec{s} \leftarrow_\$ \mathbb{Z}_q^n$ and $\vec{b} \leftarrow_\$ \mathbb{Z}_q^m$.

Let $\lfloor \cdot \rceil(x) := \lfloor p \cdot x/q \rceil$ (for the p, q above), then $\mathsf{LWR}_{\cdot, \cdot, \cdot, \lfloor \cdot \rceil}$ is the standard decisional learning with rounding problem introduced in [6].

2.2 Pseudorandom Generator

Definition 2.2. A (t, m, p)-PRG is a deterministic and polynomial-time computable function $f : \mathbb{Z}_t^n \to \{0,1\}^m$, such that for any PPT adversary $\mathcal{A}, |\Pr[\mathcal{A}(f(s)) = 1] - \Pr[\mathcal{A}(R)]| \leq \mathsf{negl}(n)$, where $s \leftarrow_\$ \mathbb{Z}_t^n, R \leftarrow_\$ \mathsf{Bern}(p)^m$, and $m > n\lceil \log(t) \rceil$.

A standard PRG is simply a $(t, m, 1/2)$-PRG, and we denote it as (t,m)-PRG and ignore $1/2$ for simplicity.

2.3 FHE

Fully Homomorphic Encryption (FHE), introduced by Rivest et al. [63] and first constructed by Gentry [30], enables evaluation of a circuit on encrypted data.

BFV FHE scheme. We use the Brakerski/Fan-Vercauteran (BFV) homomorphic encryption scheme [13, 27].

BFV scheme consists of the following PPT algorithms: $\mathsf{GenParams}(1^\lambda), \mathsf{KeyGen}(\mathsf{pp_{BFV}}), \mathsf{Enc}(\mathsf{pp_{BFV}}, \mathsf{pk}, m), \mathsf{Dec}(\mathsf{pp_{BFV}}, \mathsf{sk}, c)$ as normal PKE schemes. BFV is unconditionally correct and sound. Under the Ring-LWE hardness assumption, it also fulfills CPA security.

Given a polynomial from the cyclotomic ring $R_t = \mathbb{Z}_t[X]/(X^D + 1)$ (where D is a power-of-two, $t \equiv 1 \mod 2D$), the BFV scheme encrypts it into a ciphertext consisting of two polynomials, each of in $R_Q = \mathbb{Z}_Q[X]/(X^D + 1)$ for some $Q > t$. Here, t, Q, and D are called the plaintext modulus, the ciphertext modulus, and the ring dimension, respectively.

Plaintext encoding. In practice, instead of having a polynomial in $\mathcal{R}_t = \mathbb{Z}_t[X]/(X^D + 1)$ directly as input, applications usually hold a vector of messages $\vec{m} = (m_1, \ldots, m_D) \in \mathbb{Z}_t^D$. Thus, to encrypt such input messages, BFV first encodes the messages into a polynomial in $\mathcal{R}_t$ (via Inverse Number Theoretic Transform). We say that a BFV ciphertext has D *slots*, each of which is a $\mathbb{Z}_t$ element.

For simplicity, we assume $\mathsf{BFV.Enc}$ takes a vector of form $\mathbb{Z}_t^D$ as an input, and $\mathsf{BFV.Dec}$ outputs a vector of form $\mathbb{Z}_t^D$, and will handle encode and decode implicitly.

Operations. BFV supports the following operations.

- (Additions) For any two BFV ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2$, and $\mathsf{ct} \leftarrow \mathsf{ct}_1 + \mathsf{ct}_2$, it holds that $\mathsf{BFV.Dec}(\mathsf{ct}) = \mathsf{BFV.Dec}(\mathsf{ct}_1) + \mathsf{BFV.Dec}(\mathsf{ct}_2)$ (element-wise).

- (Multiplication) For any two BFV ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2$, and $\mathsf{ct} \leftarrow \mathsf{ct}_1 \times \mathsf{ct}_2$, it holds that $\mathsf{BFV.Dec}(\mathsf{ct}) = \mathsf{BFV.Dec}(\mathsf{ct}_1) \times \mathsf{BFV.Dec}(\mathsf{ct}_2)$ (element-wise).

- (Rotation) For any BFV ciphertexts ct, and $\mathsf{ct}' \leftarrow \mathsf{BFV.Rotate}(\mathsf{ct}, k)$ for some $k \in [D]$, let $\mathsf{BFV.Dec}(\mathsf{sk}, \mathsf{ct})[i] = \mathsf{BFV.Dec}(\mathsf{sk}, \mathsf{ct}')[i + k \mod D], \forall i \in [D]$ (i.e., all the slots $i \in [D]$ are shifted by k).

2.4   Private Information Retrieval

Given a database of N elements denoted DB given to a server and a query index $i \in [N]$ given to a client, the client wants to retrieve DB$[i]$ while not revealing to the server its index i. A PIR protocol should satisfy two properties:

- **Correctness:** In an honest execution by client and server, the client correctly retrieves the i-th entry of the database.

- **Privacy:** The server *learns nothing* about the index queried by the client.

This definition can be extended to the *client-preprocessing* setting, where the client and server jointly run a preprocessing phase, after which the client stores a hint which it then uses to perform a series of queries to the server.

Definition 2.3. A client-preprocessing Private Information Retrieval (PIR) scheme is a tuple of four procedures:

- Preprocess(DB,λ, T) $\rightarrow$ st: This is a protocol run by client and server which outputs the initial client state which will be used to perform queries later. Additional parameters λ and T define the security parameter and the number of queries supported before a client needs to re-run the Preprocess step.

- Query(st,x) $\rightarrow$ rq: This is a protocol run by a client given an index x to be queried and the client state st stored by the client and outputs a query request rq to be sent to the server.

- Answer(DB, rq) $\rightarrow a$: This algorithm is run by the server and takes in the database DB and the client's query q and outputs the server's answer a.

- Reconstruct(st,x,a) $\rightarrow$ DB$[i]$: This algorithm takes in the index queried, the client's hint h and the server's answer i and outputs the database's i-th index. It also updates the client's hint.

Our scheme will also include some public parameters which we will define later. We now formally define correctness and privacy for PIR. For this, we first define an interaction.

Definition 2.4 (Interaction)**.** A PIR interaction between client and server for PIR scheme = (Preprocess, Query, Answer, Reconstruct) is as follows:

- Client requests the database and runs Preprocess locally and saves the hint as its local state st.

- For each query x_t, for $t \in [T]$, client runs $q = $ Query(st,x_t) and sends q to the server.

- Server returns $a = $ Answer(DB.q) to the client.

- Client outputs Reconstruct(st, x_t, a).

After T queries, client must re-run the preprocessing to perform the next query. In addition, we allow the client to access and change its internal state st across algorithms.

This definition of interaction above assumes that the client runs the preprocessing. We slightly abuse the definition and allow the server to run the preprocessing for one of our theorems. In this case, the client runs an additional algorithm Init whose output it sends to the server, and saves the hints output from the server's preprocessing.

We now define correctness and privacy according to the interaction above.

Definition 2.5 (PIR correctness). A PIR scheme (Preprocess, Query, Answer, Reconstruct) is correct if, for any polynomial-sized sequence of queries $x_1, \ldots, x_Q$, the honest interaction (Definition 2.4) between a client and a server that stores a polynomial-sized database $\mathsf{DB} \in \{0,1\}^N$ outputs $\mathsf{DB}[x_1], \ldots,$ $\mathsf{DB}[x_Q]$ with probability $1 - \mathsf{negl}(\lambda)$.

Definition 2.6 (PIR privacy). A PIR scheme (Preprocess, Query, Answer, Reconstruct) is **private** if there exists a PPT simulator Sim, such that no PPT adversary $\mathcal{A}$ can distinguish the following experiments with non-negligible probability:

- **Expt$_0$**: Client interacts with $\mathcal{A}$ who acts as Server within the interaction. At every step t, $\mathcal{A}$ chooses the query index x_t, and Client is invoked with input x_t as its query.

- **Expt$_1$**: Sim interacts with $\mathcal{A}$ who acts as Server within the interaction. At every step t, $\mathcal{A}$ chooses the query index x_t, and Sim is invoked with no knowledge of x_t.

In the above definition our adversary $\mathcal{A}$ can deviate arbitrarily from the protocol.

2.5 Thorp Shuffle

The Thorp Shuffle is a shuffling algorithm introduced in [65]. It works as follows. We start with the ordered set $[1, \ldots, N]$ for some power-of-two $N = 2^n$ for some positive integer n. We will denote each element in this set a *card*. For every pair of cards in positions $i, i + N/2$, we flip a coin that decides which one will be placed in position $2i$, and which one will be placed in position $2i + 1$ on the next round. This is then repeated for multiple rounds. As the number of rounds t approaches infinity, the output of the Thorp Shuffle approaches the distribution of a uniformly sampled permutation [50]. We describe the algorithm below in Algorithm 1.

Algorithm 1 Thorp Shuffle. Parameters $t, N \in \mathbb{N}$, N even, $n = \lceil \log_2 N \rceil$.

1: **procedure** THORP SHUFFLE($A_1, \ldots A_N \in [N]$)
2:     **for** t steps **do**
3:         Let $F_i = A_i$ for $i \in \{1, \ldots, N/2\}$
4:         Let $S_i = A_{i+N/2}$ for $i \in \{1, \ldots, N/2\}$
5:         **for** j **in** $[1, \ldots, N/2]$ **do**
6:             Sample bit b uniformly at random
7:             **if** b **then**
8:                 Let $A_{2j-1} = S_j$, $A_{2j} = F_j$
9:             **else**
10:                Let $A_{2j-1} = F_j$, $A_{2j} = S_j$
11:     **return** $A_1, \ldots, A_N$

Algorithm 1 describes a "perfect" version of the Thorp Shuffle, using perfect randomness, for all elements needed to be shuffled. We will denote this perfect version of the Thorp Shuffle Th_t. In our

definition below, we will define our Thorp Shuffle with respect to operations to a single entry rather than the entire initial input $A_1, \ldots, A_N$. Specifically, in our definition, we will define two functions (other than the initial key generation) which compute the resulting position of a single element after either applying the shuffle or applying an inverse of the shuffle. Notice that either of these can be computed without requiring to perform the whole shuffle, and instead just checking a small number of bits. We define these operations in the context of a Thorp Shuffle using *computational randomness* (using a PRG to seed the bits used in each step) below.

Definition 2.7. A Thorp Shuffle Th is a tuple of three algorithms:

- Gen$(\lambda, N \in \mathbb{N}, q \in [N]) \rightarrow s \in \{0,1\}^{O(\lambda)}$: This algorithm takes in a security parameter and a set size N and outputs a PRG seed s to be used to both generate the Thorp Shuffle secure for q queries and also includes the number of rounds t to run for.

- Eval$(s, x \in [N])$: Takes in a seed s and an index $x \in N$ and outputs its Thorp Shuffle evaluation y in $O(\lambda)$ steps using s and a PRG to generate the bits necessary.

- Inv$(s, y \in [N])$: Takes in an index $y \in N$ and outputs its inverse Thorp Shuffle evaluation x in $O(\lambda)$ steps using s (outputs x such that Th.Eval$(s, x) = y$.

Concretely, the seed s is just a PRG seed of size λ and a description of how many rounds to run the protocol for, which is also a function of λ. We then use the seed s to generate the bits b in each step.

For efficiency reasons, we would like to minimize t, the number of rounds of Thorp Shuffle to run, *especially* if it will be run under FHE. Next, we will look at how to bound the Thorp Shuffle's distance from a true permutation. In this bound, we use $\mathsf{Th}_t(\cdot)$ and $\mathsf{Th}_t^{-1}(\cdot)$ to point queries to the perfect Thorp Shuffle and its inverse for a shuffle run for t steps (i.e., Thorp shuffle using true randomness instead of a PRG). For any $N = 2^n, n \in \mathbb{Z}^+, \lambda \in \mathbb{N}, q \in [N] = o(N), t \geq 1$, [52] prove that for any adaptive q-query CCA adversary $\mathcal{A}$, it follows that:

$$\left| \mathbf{P}\left(\mathcal{A}^{\mathsf{Th}_t(\cdot), \mathsf{Th}_t^{-1}(\cdot)}(\lambda, q) = 1 \right) - \mathbf{P}\left(\mathcal{A}^{\pi(\cdot), \pi^{-1}(\cdot)}(\lambda, q) = 1 \right) \right| \leq \frac{2q(4n+t)}{4n-4} \left(\frac{4qn}{N} \right)^{t/(4(n-2))}.$$

where $n = \log(N)$ and π is a random permutation. Note that this is the best-known concrete bound on the number of rounds for $q \approx O(\sqrt{N})$. The bound says that approximately every $4n$ rounds of Thorp allows us to reduce the adversary's advantage by a factor of $(4qn/N)$; given $q \approx \sqrt{N}$, this fraction is approximately $1/\sqrt{N}$. Looking ahead, in Section 4 we show how to improve this bound to cut down the number of rounds required to achieve computational indistinguishability and greatly reduce the computation depth needed for the Thorp Shuffle.

2.6 Two-server PIR in [42]

In this section, we recall the two-server PIR scheme introduced in [42], as our base construction (i.e., single-server PIR scheme with linear offline bandwidth) is greatly inspired by it (as mentioned, our hint is essentially constructed the same way as [42]).

High-level idea. First, in the preprocessing phase, Server 0 splits the database DB into $\sqrt{N}$ partitions, $\mathsf{db}_1, \ldots, \mathsf{db}_{\sqrt{N}}$ and samples $\sqrt{N}$ permutations of $[\sqrt{N}]$, $\tau_1, \ldots, \tau_{\sqrt{N}}$ and applies τ_i to db_i for all $i \in [\sqrt{N}]$. Then, it computes hints $h_1, \ldots, h_{\sqrt{N}}$ where each $h_j = \bigoplus_{i \in Q} \mathsf{db}_i[\tau_i(j)]$ (this is

as opposed to sampling independent subsets of $[N]$ and computing the parity of the subsets as in previous schemes). Notice that all the hints are *dependent* (if an element appears in one hint it certainly does not appear in another). Server 0 sends the permutations and hints to the client to store.

Online, the client can manipulate the permutations to generate both a query to Server 1 (which will allow it to retrieve its element of interest using h_j), and a "refresh" query to Server 0 (which allows it to edit the permutations in a way that they look uniform to Server 1 for the next query). This online phase greatly differs from ours.

The detailed pseudocode for their scheme is presented in Algorithm 8 in Appendix A.

Key difficulties when adapting to our model. The two major difficulties when adapting to a single-server model are that (1) the random permutation used in [42] has depth $O(N)$; and (2) we do not have a second server to perform an online "refresh" to edit the random permutation. Our base construction (single-server PIR with linear offline bandwidth) addresses these two points accordingly, and our additional techniques further optimize our base construction to build PIR with sub-linear offline bandwidth.

3 A Permutation-Based Single Server PIR Scheme

We start with a single server scheme with *linear* bandwidth based on the 2-server scheme from [39] (recalled in Section 2.6). In this model, previously explored in [70, 53], the server streams the database to the client in the offline phase, which computes (with sublinear storage) hints that can then be used at query time to perform queries that take $o(N)$ time. Looking ahead, this scheme will serve as a stepping stone to later construct a scheme that does not need linear bandwidth offline.

3.1 Building a Single Server PIR Scheme

Algorithm 2 defines a variant of our scheme ThorPIR with linear offline bandwidth. Our scheme is inspired by [42], but differs in several important aspects. We will first give a high-level intuition and present our scheme. Later we delve into the major differences between our scheme and [42].

3.1.1 Intuition.

At a high level, ThorPIR with linear bandwidth works as follows. The client first samples Q permutations of $[N/Q]$, $\tau_1, \ldots, \tau_Q$. Then, the server streams the database to the client one element at a time. For each database element it sees, it finds the appropriate hint (i.e., which chunk of data the element is to-be XOR-ed with) this element belongs to (it does this using the permutations sampled above, if we use the Thorp Shuffle to represent the permutations, finding each appropriate hint will take $O(\lambda)$ time) and xors this element into the appropriate hint. It repeats this for every element in the database.

At the end of the preprocessing, the client holds N/Q hints, where each hint h_j, for $j \in [N/Q]$, represents the XOR of $\mathsf{db}_i[\tau_i(j)]$ for all $i \in Q$ (where db_i represents the i-th chunk of db which is divided into Q equally sized chunks). At the end of the preprocessing phase, the client stores the permutations and hints permanently.

Online, for the first query to $x = (q,k) \in ([Q] \times [N/Q])$, the client first computes $j = \tau_q^{-1}(k)$, and uses this j to find all the other database elements it needs to recover $\mathsf{DB}[x]$ from the hint. In more detail, compute $o_i \leftarrow \tau_i(j)$ for $i \neq q$. Then, the client can compute $\mathsf{DB}[x] = h \oplus_{i \neq q} \mathsf{db}_i[\tau_i(j)]$

where h is the hint when getting $\mathsf{db}_i[o_i]$ back by sending $o_i = \tau_i(j)$. These o_i's are indistinguishable from uniformly at random from $[K]$ by the property of the underlying Thorp shuffle, and thus leaks nothing about (q,k). Note that, however, it should also send o_q, as otherwise, the server learns that $x = (q, \cdot)$. Thus, the client samples o_q uniformly at random from $[K]$ and sends it (notice that this uniformly sampled element is the query element with probability $1/K$, and so for each o_i that the server sees it is uniformly distributed in $[K]$). After querying, the client simply stores $\mathsf{db}_i[o_i]$ in USED_i (some dictionary storing $\mathsf{USED}_i[o_i] = \mathsf{db}_i[o_i]$) for all $i \in [Q]$.

Now, for the following queries, the client first obtains o_i as in the first query. However, note that now the server might have seen o_i already, which may leak some information (e.g., for the same $x = (q,k)$ as the first query, o_q can be different but everything else remains the same). However, not that for the o_i that is already seen, the client know what $\mathsf{db}_i[o_i]$ is as it is stored in USED_i. Thus, instead of directly sampling o_i, resample o_i uniformly at random from $[K] \setminus \mathsf{USED}_i$. Send the o_i's after resampling. For the ones that are not resampled, they are indistinguishable from $[K] \setminus \mathsf{USED}_i$ by the property of the Thorp shuffle. Again, every query stores $\mathsf{db}_i[o_i]$ in USED_i.

With all these intuitions, we show our construction in Algorithm 2, and can show the following theorem.

Algorithm 2 ThorPIR with linear offline bandwidth

1: **procedure** PREPROCESS($\mathsf{DB} \in \{0,1\}^N, \lambda, \mathrm{T}$)
2:    Let $K := N/Q$. $\mathsf{DB} = \mathsf{db}_1, \ldots, \mathsf{db}_Q$ where each $\mathsf{db}_i \in \{0,1\}^K$.
3:    Let $s_1, \ldots, s_Q$ be seeds generated with $\mathsf{Th.Gen}(\lambda, N, T)$.
4: ▷ The number of rounds t for the Thorp shuffle is chosen according to Theorem 3.2, such that the adversary has advantage $\leq 2^{-\lambda}$.
5:    Initialize $h_1, \ldots, h_K = 0$.
6:    **for** j in $[1, \ldots, K]$ **do**
7:       **for** i in $[1, \ldots, Q]$ **do**
8:          Let $k = \mathsf{Th.Eval}(s_i, j)$.
9:          Let $h_j = h_j \oplus \mathsf{db}_i[k]$.
10:    Initialize empty dictionaries Used_i for $i \in [Q]$.
11:    **return** $\mathsf{st} = (H = (h_1, \ldots, h_K), (\mathsf{Used}_i)_{i \in [Q]}, (s_1, \ldots, s_Q))$.
12: **procedure** QUERY($\mathsf{st}, x = (q,k) \in [Q] \times [K]$)
13:    Let $y = \mathsf{Th.Inv}(s_q, k)$ and add y to st.
14:    Let D_i denote the set of all the keys of $\mathsf{st.Used}_i$ for $i \in [Q]$.
15:    Let $o_i \leftarrow \mathsf{Th.Eval}(s_i, y)$ for $i \in [Q] \setminus \{q\}$ and $o_q \leftarrow_\$ [K] \setminus D_q$.
16:    For each $i \in [Q]$: **if** $o_i \in D_i$ **then** $o_i \leftarrow_\$ [K] \setminus D_i$.
17:    **return** $\mathsf{rq} = (o_1, \ldots, o_Q)$.
18: **procedure** ANSWER($(\mathsf{DB} = (\mathsf{db}_1, \ldots, \mathsf{db}_Q), \mathsf{rq} = (o_1, \ldots, o_Q))$)
19:    **return** $a = (\mathsf{db}_1[o_1], \ldots, \mathsf{db}_Q[o_Q])$.
20: **procedure** RECONSTRUCT($(\mathsf{st}, x = (q,k), a = \{\mathsf{db}_i[o_i]\}_{i \in [Q]})$)
21:    Let $\mathsf{st.Used}_i[o_i] \leftarrow \mathsf{db}_i[o_i]$ for all $i \in [Q]$.
22:    $a_i \leftarrow \mathsf{Th.Eval}(s_i, \mathsf{st}.y)$ for $i \in [Q] \setminus \{q\}$.
23:    Let $\mathsf{DB}[x] = \mathsf{db}_q[k] = \left(\bigoplus_{i \in [Q], i \neq q} \mathsf{st.Used}_i[a_i] \right) \oplus \mathsf{st}.h_y$.
24:    **return** $\mathsf{DB}[x]$.

Theorem 3.1. *The PIR scheme defined in Algorithm 2 is correct (Definition 2.5) and private (Definition 2.6) for $T = o(N)$ queries, and runs with the following complexities:*

- $O_\lambda(N)$ *preprocessing time.*

- $O(N)$ *offline bandwidth.*

- $O_\lambda(Q)$ *online client time.*

- $O(Q \log N)$ *online bandwidth*

- $O(Q)$ *online server time.*

- $O(N/Q + TQ)$ *client storage.*

- $O_\lambda(1)$ *update time.*

For improved readability, we defer the proof this theorem in Section 6, as the proof requires new techniques to adapt to the Thorp Shuffle instantiation only secure for T queries against adaptive adversaries and to accommodate the "caching" of the online phase.

Note that after T queries we need to re-run the scheme from scratch to achieve a scheme with the same amortized complexities that runs for unlimited queries.

Parameter setting. For our scheme to be sublinear across all complexities, we must pick Q,T such that $Q \cdot T = o(N)$. For example, for $Q = T = N^{1/3}$, we get a single server PIR scheme with $O(N^{2/3})$ amortized server time, $O(N^{2/3})$ offline and total bandwidth, $O(N^{1/3})$ online bandwidth and query time, and $O(N^{2/3})$ client storage. By using the Thorp Shuffle permutation, we actually circumvent the linear storage constraint present in [42] due to using the Fisher-Yates construction [28]. See Section 6 for why the update time is $O_\lambda(1)$. In previous works [40, 69, 53], schemes achieve a tradeoff of $\sqrt{N}$ client space and $\sqrt{N}$ amortized online time. ThorPIR requires storing more information after each query and cannot achieve this tradeoff exactly. We do however achieve much better update time ($O_\lambda(1)$) than previous schemes, along with a concretely small client storage since it is not dependent on the security parameter.

3.1.2 Comparison to [42]

The main differences between our scheme and the scheme presented in [42] are as follows:

1. Our permutations are realized by the Thorp Shuffle [65, 52]. This means, on one hand, that we can only show at most $T = o(N)$ points of each permutation sampled to the adversary before our shuffle's security no longer holds. However, this also means that we greatly reduce the depth of the computation, since [39] used Fisher-Yates which is a sequential algorithm with $O(N)$ depth.

2. The server now streams the database to the client upon request. Then, the client computes the hint itself.

3. We completely eliminate the refresh operation. This requires changing the entire online phase to work differently. Previous works have ported two-server schemes to single server schemes before by storing backup hints during the offline phase to later replace used hints [22, 40, 69].

This approach does not directly work in this case, since the hint at the client is not comprised of independent subsets of the database. Instead, we take a different approach and store elements seen, performing dummy queries when repeated elements are queried. This requires quite a few modifications to the online phase to ensure that the distribution the server sees is always uniform and independent of the queries.

4. Our scheme needs to be re-run from scratch every T queries. This is also true for previous single server client preprocessing scheme [22, 40, 69] and a consequence of not being able to refresh our state with the help of a second server as is the case for two-server schemes. Note that this is unrelated to the fact that our permutation only supports $q = o(N)$ queries, and is because we only store a sublinear number of backup hints we can use to refresh the client state and maintain the table's distribution (using a stronger shuffle algorithm would not help).

Condition (1) causes us to incur an extra λ factor in the preprocessing when compared to [42]. However, it also decreases the depth of the preprocessing computation from $O(N)$ to $O_\lambda(1)$. Conditions (2) and (3) are what allow us to eliminate completely the role of Server 0 in the original scheme and are what allow us to transform it into a single server scheme. Condition (4) is a consequence of eliminating Server 0. Since we can no longer receive fresh random elements to replace the ones we used, we can only support a limited number of queries before having to re-run the preprocessing.

3.2 An Improved Thorp Shuffle Bound

We know that the output of the Thorp Shuffle converges to a uniformly sampled permutation as the number of rounds we shuffle for approaches infinity [50]. More than that, [52] have analyzed the Thorp Shuffle and specifically looked at exactly how well a set of $N = 2^n$ cards are shuffled after t rounds of the the Thorp Shuffle. We define "how well" as the advantage an adversary would have in distinguishing the shuffled elements from a truly random permutation given q queries to it. Specifically, [52] bounds the total variational distance between the q queries of the Thorp Shuffle and q queries from a uniform permutation over N elements. We improve on the bound on [52] that was aforementioned with the following theorem.

Theorem 3.2. *Let $N = 2^n$ and $q \in \{1, \ldots, N\}$, $\{\mathsf{Th} : t \geq 1\}$ be the Thorp Shuffle of $[N]$ after t rounds. Then, for adaptive, q-query unbounded adversary $\mathcal{A}$:*

$$\left| \mathbf{P}\left(\mathcal{A}^{\mathsf{Th}_t(\cdot), \mathsf{Th}_t^{-1}(\cdot)}(\lambda, q) = 1\right) - \mathbf{P}\left(\mathcal{A}^{\pi(\cdot), \pi^{-1}(\cdot)}(\lambda, q) = 1\right) \right| \leq \frac{2q(n+t)}{n+1} \left(\frac{2qn}{N}\right)^{t/(2(n+1))}.$$

where π is a uniformly random permutation.

Recall Th_t denotes the perfect shuffle run for t rounds and we allow for oracle point queries and inverse queries to the shuffle (for a total of q queries adaptively). Notice that compared to the bound achieved before (stated in Section 2.5), our bound is smaller by about a factor of $(2qn/N)^{t/(2n)}$, which allows to use about half as many rounds while maintaining the same security. This directly reduces the depth of the computation for the shuffle.

Now, we provide an overview of the techniques used in our new bound's proof.

Techniques. The proof in Morris et al. and subsequently ours uses a *coupling argument*. We briefly give some intuition about what this is. A Markov chain can be defined by a transition matrix

15

P and initial state x. We denote $P^t(x)$ to mean the resulting state of applying the transition on initial state x for t times, sequentially. Let π be the stationary distribution over the group, where, for our purposes, the group is the group of all permutations of N cards, and the stationary distribution is a uniform sample from this group (a uniform permutation). The relevant measurement that we would like to minimize is the total variational distance between $P^t(x)$ and π, denoted $\Delta\left(P^t(x), \pi\right)$. In specific, if we can show that the distance is 0, then we can say that after t steps our initial state is indistinguishable from uniform. The first thing to verify is that the Thorp Shuffle, starting from any initial distribution, eventually converges to the stationary distribution after enough steps. This was shown in [50].

The technique to bound the total variational distance is is to define two pair processes, meaning two Markov chains with the same transition matrix P, but different starting states (where one gets the desired x and the other some sample from π). This makes it so such that if for any t, the two pair processes hit the same state, then they are the same from then onwards (since they are both updated in the same way).

In [52], they show that for pair processes $\{X_t\}$ and $\{Y_t\}$ such that $X_1 = x$, $Y_1 \xleftarrow{\$} \pi$, both with transition matrix P, it holds that:

$$\Delta\left(P^t(x), \pi\right) \le \mathbf{P}(X_t \ne Y_t) = \mathbf{P}(T > t),$$

where T is a random variable, $T = \min\{t : X_t = Y_t\}$. The insight here is that since $Y_1 \xleftarrow{\$} \pi$ and P is a probabilistic transition matrix, $Y_t \xleftarrow{\$} \pi$ for any t. Then, if after T steps, $X_T = Y_T$, we can conclude that X_T is indistinguishable from a uniform sample from the group.

At a very high level, our contribution in the proof is to find two new ways to prove $X_t = Y_t$ at step t in the algorithm. These two new 'convergence opportunities' between the pair processes allow us to increase the probability that at any given timestep t the two variables are matched and therefore reduce the number of steps we need to run the shuffle for by more than half.

Due to improved readability, we defer the full proof for Theorem 3.2 to Section 7.

3.3 A Conjectured Bound

As aforementioned, in this work, we show that for an adversary given q queries to a CCA oracle, the probability that the adversary can distinguish between this oracle being for a true permutation over N elements or for a Thorp Shuffle run for t rounds is less that or equal to $\frac{2q(n+t)}{n+1}\left(\frac{2qn}{N}\right)^{t/(2(n+1))}$., where $N = 2^n$ and n is a positive natural number.

Note first this bound is against a computationally unbounded adversary (albeit the adversary can make only q queries). We do not have any known bounds for computational adversaries. Second, for $q = \sqrt{N}$ queries, even in the fully unbounded adversary scenario, there is no known attack for any $t > \lambda$ rounds (given $N = \mathsf{poly}(\lambda) < 2^{\lambda/2}$). Therefore, we propose the following conjecture, and after give some intuition on why we believe it to hold.

Conjecture 3.3. Let $N = 2^n = \Omega(\lambda^c)$ for some constant c, and $q = \sqrt{N}$. Let $\{\mathsf{Th} : t \ge 1\}$ be the Thorp Shuffle of $[N]$ after $t = \lambda$ rounds. Then, for adaptive, q-query PPT adversary $\mathcal{A}$:

$$\left|\mathbf{P}\left(\mathcal{A}^{\mathsf{Th}_t(\cdot), \mathsf{Th}_t^{-1}(\cdot)}(\lambda, q) = 1\right) - \mathbf{P}\left(\mathcal{A}^{\pi(\cdot), \pi^{-1}(\cdot)}(\lambda, q) = 1\right)\right| \le 2^{-c\lambda}.$$

where π is a uniformly random permutation.

Intuition on our conjecture. The intuition for our conjecture is as follows. In the case where the adversary makes queries to the Thorp shuffle oracle, it will gain about two bits of information: since the query answer fixes the ending position of an input, we know what swap happened exactly before it. If we go back to two swaps, we can guess with $1/2$ certainty what coin flip happened there, and so on and so forth (going back X rounds, we guess with $1/2^X$ probability, thus $1/2^X$ bit of information). When we sum all this up, we get that about two bits of information are revealed. Then, for any adversary querying $\sqrt{N}$ positions, we reveal at most $2\sqrt{N} < 2\lambda^{c/2}$ bits that define our shuffle.

After $\sqrt{N}$ queries by the adversary, then, there are still $N/2 \cdot \lambda/2 - 2\lambda^{c/2} = (\lambda/2)^{c+1} - \lambda^{c/2} \gg c \cdot \lambda$ bits of entropy in the resulting elements of the permutation. In particular, this means that a computationally bounded adversary could not simply enumerate all possibilities in order to distinguish the Thorp Shuffle from a random permutation.

On the other hand, in general, high entropy is not sufficient to prevent an adversary from distinguishing the distribution from a uniform random permutation (e.g., the set of all permutations with the first bit fixed to 0 has high entropy but is easy to distinguish from random with enough samples). In the case of a Thorp Shuffle, it is unclear whether there is such a characteristic that would allow the adversary to distinguish it from a random permutation. Indeed, we have proven that with $O(\lambda)$ iterations, the Thorp Shuffle is indistinguishable (with $q = o(N)$ queries) from random except with negligible probability in λ.

We also know that for any $q > 2$, for a Thorp Shuffle over $\log N$ rounds, there exists an attack for which an adaptive adversary can distinguish between a truly random permutation and the Thorp Shuffle with greater than negligible probability. The high level idea is for the adversary to check where element '1' (the first input element) ended up, and where element '2' (the second input element) ended up. Since in the Thorp Shuffle with only $\log N$ steps, they always end up in different halves of the final permutation, while on a real permutation their positions are uniformly at random. Thus, the adversary would have advantage $1/2$ in distinguishing between a true permutation and a Thorp Shuffle with $\log N$ rounds. However, once we run a couple more rounds (say, $\lambda = \omega(\log N)$ rounds, since N is $\mathsf{poly}\lambda$), such an attack does not work, as the two elements may not end up in two different halves. To our knowledge, there is no known attack on being able to distinguish the Thorp Shuffle from a truly random permutation.

We thus leave it as a conjecture that a computationally bounded adversary making $q = O(\sqrt{N})$ ($N = O(\lambda^c)$ for some constant c) queries cannot distinguish between the Thorp Shuffle after λ iterations and a random permutation with probability greater than $2^{-c\lambda}$.

4 Homomorphic Thorp Shuffle

We have already motivated why the Thorp Shuffle is the best candidate shuffling algorithm to perform under FHE. However, we have not seen yet how exactly to deal with the problems alluded to in Section 1. Here, we recall the two main problems one faces when attempting to evaluate the Thorp Shuffle under FHE, and how we propose to deal with them.

The first issue comes from the homomorphic PRG evaluation. Evaluating a commonly used PRG like AES homomorphically is very slow. The state-of-the-art homomorphic AES evaluation takes 86 seconds to generate 128 random bits on a single CPU [68].

The second issue is the naive realization of Thorp Shuffle is very FHE-unfriendly and can be very costly in terms of runtime (as it needs to map elements inside the same ciphertext to different

locaitons and potentially different ciphertexts).

4.1 LWR-based FHE-friendly PRG

We start by addressing the first issue: building a FHE-friendly PRG.

Recall that operations over small finite fields are more FHE-friendly, since FHE supports only multiplications and additions over $\mathbb{Z}_t$ for some prime t.[3] Thus, with the motivation of building an FHE-friendly PRG, we focus on lattice-based constructions, as they can indeed work over some finite field $\mathbb{Z}_t$. Our goal is to construct a (t, m, p)-PRG with only multiplications and additions.

Construction with $p = r/t$ for $r \in \mathbb{Z}_t$. We start with a simpler requirement: sampling bits with distribution $\mathsf{Bern}(r/t)$ for $r \in \mathbb{Z}_t$. At a high level, the core idea is to rely on the Learning-with-rounding assumption: the seed is simply a vector $s \leftarrow_\$ \mathbb{Z}_t^n$. Then, the sampling is easy: first randomly sample $A \leftarrow_\$ \mathbb{Z}_t^{w \times n}$ and compute $R_r(As)$, where $R_r : \mathbb{Z}_t \to \{0,1\}$ is defined as

$$R_r(x) = \begin{cases} 1 & \text{if } x \in [0, r-1] \\ 0 & \text{o.w.} \end{cases}$$

Here the R_r can be evaluated via a degree-$(t-1)$ polynomial function (interpolated using R_r). Of course, one restriction is that we need $\mathsf{LWR}_{n,q,2,R_r}$ to be hard, which is assumed to hold as long as r is large enough.

Security of $\mathsf{LWR}_{n,q,2,R_r}$. While the rounding function R_r is not a standard rounding function, for a lot of values of r, we can in fact reduce a regular $\mathsf{LWR}_{n,q,q',\lfloor \cdot \rceil}$ instance for some $q' \geq 2$ to our LWR assumption with this special rounding function. We formalize it with the following lemma.

Lemma 4.1. *For any $n, q > 0$, and $r, q' > 0$ such that $\lfloor (r-1) \cdot (q/q') \rfloor = 0$ and $\lfloor r \cdot (q/q') \rfloor = 1$, it holds that $\mathsf{LWR}_{n,q,q',\lfloor \cdot \rceil} \leq \mathsf{LWR}_{n,q,2,R_r}$.[4]*

Proof. To prove this lemma, given an adversary $\mathcal{A}$ that breaks $\mathsf{LWR}_{n,q,2,R_r}$, we construct the following adversary that breaks $\mathsf{LWR}_{n,q,q',\lfloor \cdot \rceil}$.

Given an $\mathsf{LWR}_{n,q,q',\lfloor \cdot \rceil}$ sample $(A, \vec{b}) \in \mathbb{Z}_q^{w \times n} \times \mathbb{Z}_p^{w \times 1}$, let $\vec{b}'[i] \leftarrow 1$ if $\vec{b}'[i] \neq 0$ and $\vec{b}[i] \leftarrow 0$ otherwise, for $i \in [w]$. Send $(A, \vec{b}')$ to $\mathcal{A}$ and returns whatever returned from $\mathcal{A}$.

If the input is a valid $\mathsf{LWR}_{n,q,q',\lfloor \cdot \rceil}$ input, and $\vec{b}[i] = 0$, we know that $A s k \in [0, r-1]$ and otherwise $\vec{b}[i] \neq 0$. This thus gives us a valid $\mathsf{LWR}_{n,q,2,R_r}$ sample. Otherwise, we obtain a random sample. □

With this intuition, we define our algorithm in Algorithm 3.

Theorem 4.2. *For any $n > 0, t > 0$ being a prime, $m > n\lceil \log(t) \rceil$, and $r \in \mathbb{Z}_t$, and $A \leftarrow_\$ \mathbb{Z}_t^{m \times n}$, f_A in Algorithm 3 is a $(t, m, r/t)$-PRG (Definition 2.2), under $\mathsf{LWR}_{n,t,2,R_r}$ is hard.*

Proof. The proof is straightforward. We design a hybrid scheme: replacing f_A with uniformly sampling $u \leftarrow \mathsf{Bern}(r/t)^w$ and return u. The adversary that can distinguish this hybrid and the original scheme breaks $\mathsf{LWR}_{n,t,2,R_r}$. □

---

[3]Technically, BFV also works with t^r where t is a prime and $r > 1$. However, this reduces the amortized efficiency and we do not consider such parameterization.

[4]I.e., $\mathsf{LWR}_{n,q,2,R_r}$ reduces to $\mathsf{LWR}_{n,q,q',\lfloor \cdot \rceil}$.

Algorithm 3 $(t, m, r/t)$-PRG for $r \in \mathbb{Z}_t$

1: **procedure** $f_A(s)$ $\triangleright\ A \leftarrow_\$ \mathbb{Z}_t^{m \times n}$

2:     Define

$$R_r(x) = \begin{cases} 1 & \text{if } x \in [0, r-1] \\ 0 & \text{o.w.} \end{cases}$$

3:     **return** $R_r(As)$

Achieving $p = 1/2$. We have achieved values of $p \neq 1/2$, the choices are relatively restricted. The main obstacle to achieving arbitrary p is the finite field size t that we work over: it can be arbitrary and thus may not be compatible with p (i.e., there doesn't exist $r \in \mathbb{Z}_t$ such that $r/t = p$). One trivial solution, of course, is to choose t according to p. However, in the context of FHE, t is subject to certain constraints. Specifically, since we want to use FHE to evaluate the PRG, t is decided by the underlying FHE scheme, which can be constrained by other factors and has to be a prime for best efficiency (also, since $\mathsf{LWR}_{n,t,2,R}$ is trivially insecure for $t = 2$, that is not an option for us). For our construction, we need to achieve $p = 1/2$ for an arbitrary prime t.

One natural idea is that if t is a prime of size $\Omega(2^\lambda)$, we can pick $r = \lceil t/2 \rceil$, and it holds that $r/t - 1/2 = \mathsf{negl}(\lambda)$. However, if t is too large, it cannot be used as an FHE plaintext modulus in practice, as normally the FHE plaintext modulus $< 2^{30}$ for best efficiency: the noise growth of BFV is linear in t, which means that for a larger plaintext field t, the number of levels that can be supported is much smaller. Furthermore, another issue with this idea is that our PRG circuit has a depth in $\log(t)$, and thus if $\log(t)$ is too large, the PRG circuit also becomes impractical. Thus, to avoid these issues, we devise ways to obtain $p = 1/2$ for any small t.

Achieving $p = 1/2$ with small t. We can output bits with distribution arbitrarily close to uniform while maintaining a small t using the following algorithm which takes in a seed $s \in \mathbb{Z}_t^n$:

1. Sample $\vec{a} \leftarrow_\$ \mathbb{Z}_t^n$. Let $x = \langle \vec{a}, s \rangle$.

2. **If** $x = 0$, go to step 1. **Else** output $x > (t-1)/2$.

To make this algorithm deterministic, we can set a maximum number of k runs. If after k times, $\langle \vec{a}, s \rangle$ is 0 for all k repetitions, return 0. In this case, the probability of sampling $0 \leq 1/2 + t^{-k}$ and thus we can simply let $k = O(\lambda/\log(t))$ to obtain the exact security. With this intuition, we can construct a PRG with $p = 1/2$. For our PRG, it will also be important to make this algorithm oblivious by fixing the number of runs.

A more FHE-friendly circuit. Our goal is to make the sampling process FHE-friendly. Unfortunately, most FHE schemes only support multiplications and additions natively. Therefore, to evaluate a comparison to 0 (as above), requires transforming the circuit into a degree-$(t-1)$ function. Evaluation of this function would cost at least t multiplications for the comparison. Furthermore, when repeating k times as above, we need $k \cdot t$ multiplications.

To reduce such computation cost, we design an alternative function $R : \mathbb{Z}_t \to \mathbb{Z}_2$ that can be evaluated much more efficiently. Observe that for any prime t, $f(x) := x^{(t-1)/2}$ can only be $1, -1, 0$, and it is 0 if and only if $x = 0$. Thus, if $x \in \mathbb{Z}_t^*$, $x^{(t-1)/2}$ is 1 or -1. Moreover, half of the elements in $\mathbb{Z}_t^*$ are mapped to 1 and the other half to -1 [62]. With these observations, we construct the following function: $R(x) = (f(x) + 1) \cdot 2^{-1}$. Using repeated squaring, computing this function only

takes $\log(t)$ multiplications. Therefore, even repeating the process $k = O(\lambda/\log(t))$ times, it only takes $k \cdot \log(t) = O(\lambda/\log(t) \cdot \log(t)) = O(\lambda)$ multiplications (with a depth of $O(\lambda/\log(t) + \log(t))$). Moreover, to check whether x is 0 also becomes easy: simply compute $1 - f(x)^2$, which returns 1 if x is 0 and returns 0 otherwise.

With these optimizations, we formalize our construction as in Algorithm 4.

Algorithm 4 $(t, m, 1/2)$-PRG

1: $f(x) := x^{(t-1)/2}$
2: **procedure** $f_{A_1,\ldots,A_k}(s)$ $\triangleright A_i \leftarrow_\$ \mathbb{Z}_t^{m \times n}$ for $i \in [k]$
3:      **for** $i \in [k]$ **do**
4:          $\mathsf{tmp}_i \leftarrow f(A_i s_i)$
5:          $\mathsf{Rnd}_i \leftarrow (\mathsf{tmp}_i + 1) \cdot 2^{-1} \in \mathbb{Z}_t$
6:          $\mathsf{Ind}_i \leftarrow \mathsf{tmp}_i^2 \in \mathbb{Z}_t$
7:      $\mathsf{res} \leftarrow \mathsf{Rnd}_1 \cdot \mathsf{Ind}_1 + (1 - \mathsf{Ind}_1)(\mathsf{Rnd}_2 \cdot \mathsf{Ind}_2) + \cdots + (\prod_{i \in [k-1]}(1 - \mathsf{Ind}_i))(\mathsf{Rnd}_k \cdot \mathsf{Ind}_k)$
8:      **return** res

We will prove the security of Algorithm 4 through a series of hybrid experiments. For each Hybrid i, for $i \in [k]$, we will show indistinguishability from the previous hybrid by showing that $\mathsf{LWR}_{n,t,2,R_i}$ holds, where for each $i \in [k]$, we define $R_i(x)$ as:

$$R_i(x) = \begin{cases} 1 & \text{if } x^{(t-1)/2} = 1 \\ 0 & \text{if } x^{(t-1)/2} = t-1 \\ u_i & \text{otherwise} \end{cases}$$

where $u_i \leftarrow_\$ \mathsf{Bern}(p_i)$ for $p_i = \frac{(t^{k-i}-1)/2+1}{t^{k-i}}$, for $i \in [k-1]$; and $u_i = 0$ for $i = k$. We formalize the theorem as follows.

Theorem 4.3. *For any $n > 0, t > 0$ being a prime, $k > 0$ such that $t^{-k} = \mathsf{negl}(n)$, $m > n\lceil\log(t)\rceil$, then $f_{A_1,\ldots,A_k}$ in Algorithm 3 is a (t,m)-PRG (Definition 2.2) under $\mathsf{LWR}_{n,t,2,R_1}, \ldots, \mathsf{LWR}_{n,t,2,R_k}$ are hard.*

Proof. To prove this security, we design the following hybrids.

Hyb_0 : a rearrangement of our actually scheme, as follows.

1. If $A_1 s_1 \neq 0$: output $(R_1(A_1 s_1), d)$

2. If $A_2 s_2 \neq 0$: output $(R_2(A_2 s_2), d)$

3. ...

4. If $A_{k-1} s_{k-1} \neq 0$: output $(R_{k-1}(A_{k-1} s_{k-1}), d)$

5. output $(R_k(A_k s_k), d)$

Hyb_1 Same as Hyb_0, except that line 5 in Hyb_0 is replaced with: output $(R_k(u), d)$ where $u \leftarrow_\$ \mathbb{Z}_t^w$.

Hyb_2 : Same as Hyb_1, except that line 4 in Hyb_0 is replaced with: output $(R_{k-1}(u), d)$ where $u \leftarrow_\$ \mathbb{Z}_t^w$.

. . .

Hyb_{k-1} : Same as Hyb_{k-2}, except that line 2 in Hyb_0 is replaced with: output $(R_2(u), d)$ where $u \leftarrow_\$ \mathbb{Z}_t^w$.

Hyb_k : Same as Hyb_{k-1}, except that line 1 in Hyb_0 is replaced with: output $(R_1(u), d)$ where $u \leftarrow_\$ \mathbb{Z}_t^w$.

Clearly Hyb_0 is equivalent to our original scheme. Then, Hyb_i and Hyb_{i-1} for $i \in [k]$ are indistinguishable under $\mathsf{LWR}_{n,t,2,R_i}$ is hard. Furthermore, Hyb_k is by itself sampling from $\mathsf{Bern}(1/2^m)$ except with $O(t^{-k}) = \mathsf{negl}(\lambda)$ probability. Lastly, $k = \mathsf{poly}(\lambda)$, our original scheme is indistinguishable from Hyb_k, our scheme is a (t,m)-PRG. □

4.2   Shuffling via BFV

To perform the Thorp Shuffle obliviously, we will require an FHE scheme. The best performing scheme for this scenario is BFV. BFV encrypts a plaintext of form $\mathbb{Z}_t^D$ where D is the ring dimension. This means that a random vector of D random bits is generated using Random Sampling under BFV. Furthermore, directly realizing Thorp Shuffle is not very efficient using BFV, as it requires mapping an element at slot i to slot $2i$ or $2i+1$ for all $i \in [N/2]$. To do this in BFV, one needs to extract each element from the ciphertext and rotate it accordingly, which is very inefficient. Instead of directly realizing Thorp Shuffle, we use a butterfly shuffle: for round $\ell \in [n]$, position j is swapped with position $j + 2^{\ell-1}$ for all $j \in [N]$ such that $\lfloor \frac{j}{2^{\ell-1}} \rfloor$ is odd. As discussed in [24, Lemma 1], every n rounds of butterfly shuffle is *equivalent* to n rounds of Thorp Shuffle.

Then, the swapping becomes easy: for level ℓ, if $2^{\ell-1} < D$, it means that every elements are swapped with another element in the same ciphertext; thus, we separate a single ciphertext into two ciphertexts, use the random bits to swap them, and then recombine them. For example, if $\ell = 1$, we simply separate a ciphertext into two ciphertexts, where the first one contains all the odd slots of the original ciphertext while the second one contains all the even slots. Then, swap the two ciphertexts using the encrypted random bits. Note that one ciphertext contains D elements, but a swap inside a ciphertext requires only $D/2$ random bits. Thus, a ciphertext of random bits containing D bits can serve two swaps for two different ciphertexts. If $2^\ell \geq D$, simply swap the individual slots in two ciphertexts with a ciphertext containing D random bits.

The algorithm is formally presented below in Algorithm 5.

Leveraging relaxed bootstrapping. Naively realizing Algorithm 5 requires $(r+1)\log(N)$ levels for shuffling plus $\log(t) + \lambda/\log(t)$ levels for PRG evaluation. This can cause the BFV parameters to be very large without bootstrapping. However, a direct application of regular BFV bootstrapping is very costly. To avoid this issue, we employ the relaxed bootstrapping technique introduced in [46]. Essentially, relaxed bootstrapping says that the correctness of bootstrapping holds *only* when the input is a fixed subset of the plaintext space, and its (amortized) efficiency can be orders of magnitude faster than regular BFV bootstrapping. This is exactly our case: we can simply encode each database entry into a subset of the plaintext $\mathbb{Z}_t$ instead of using the entire plaintext space. Note that additionally, our PRG only generates bits, which is again a subset of $\mathbb{Z}_t$. Therefore, we first generate the random bits and then apply the (relaxed) bootstrapping, which allows c more levels of computation for some $c > 0$ according to the BFV bootstrapping parameters (practically speaking 5-10 levels as shown in [46]). Then, we apply bootstrapping for every c levels for $(r+1)\log(N)/c$ times.

Algorithm 5 Homomorphic Thorp Shuffle via BFV. BFV ring dimension D being a power of two, plaintext modulus t. Shuffling parameters $r, N \in \mathbb{N}$, and N/D is an even integer. $n = \log_2(N)$.

1: **procedure** $\mathsf{SingleShuffle}_1(\mathsf{pk}_{\mathsf{BFV}}, \mathsf{ct}_{\mathsf{bits}}, \mathsf{ct}_{\mathsf{db}_1}, \mathsf{ct}_{\mathsf{db}_2}, \ell)$ $\qquad\qquad$ ▷ All operations are under BFV
2: $\qquad$ Let $\vec{l} \leftarrow 1^L||0^L||1^L||\ldots||0^L \in \mathbb{Z}_t^D$ where $L \leftarrow 2^{\ell-1}$
3: $\qquad \mathsf{ct}_{\mathsf{db}_1,1} \leftarrow \mathsf{ct}_{\mathsf{db}_1} \cdot \vec{l}$
4: $\qquad \mathsf{ct}_{\mathsf{db}_1,2} \leftarrow \mathsf{Rotate}(\mathsf{ct}_{\mathsf{db}_1} - \mathsf{ct}_{\mathsf{db}_1,1}, -L)$
5: $\qquad \mathsf{ct}_{\mathsf{bits},1} \leftarrow \mathsf{ct}_{\mathsf{bits}} \cdot \vec{l}$
6: $\qquad \mathsf{ct}'_{\mathsf{db}_1,1} \leftarrow \mathsf{ct}_{\mathsf{db}_1,1} \cdot \mathsf{ct}_{\mathsf{bits},1}$
7: $\qquad \mathsf{ct}'_{\mathsf{db}_1,2} \leftarrow \mathsf{ct}_{\mathsf{db}_1,2} \cdot (1 - \mathsf{ct}_{\mathsf{bits},1})$
8: $\qquad \mathsf{ct}'_{\mathsf{db}_1} \leftarrow \mathsf{ct}'_{\mathsf{db}_1,1} + \mathsf{ct}'_{\mathsf{db}_1,2} + \mathsf{Rotate}(\mathsf{ct}_{\mathsf{db}_1,1} - \mathsf{ct}'_{\mathsf{db}_1,1} + \mathsf{ct}_{\mathsf{db}_1,2} - \mathsf{ct}'_{\mathsf{db}_1,2}, L)$
9: $\qquad \mathsf{ct}_{\mathsf{db}_2,1} \leftarrow \mathsf{ct}_{\mathsf{db}_2} \cdot (\vec{1} - \vec{l})$
10: $\qquad \mathsf{ct}_{\mathsf{db}_2,2} \leftarrow \mathsf{Rotate}(\mathsf{ct}_{\mathsf{db}_2} - \mathsf{ct}_{\mathsf{db}_2,1}, L)$
11: $\qquad \mathsf{ct}_{\mathsf{bits},2} \leftarrow \mathsf{ct}_{\mathsf{bits}} - \mathsf{ct}_{\mathsf{bits},1}$
12: $\qquad \mathsf{ct}'_{\mathsf{db}_2,1} \leftarrow \mathsf{ct}_{\mathsf{db}_2,1} \cdot \mathsf{ct}_{\mathsf{bits},2}$
13: $\qquad \mathsf{ct}'_{\mathsf{db}_2,2} \leftarrow \mathsf{ct}_{\mathsf{db}_2,2} \cdot (1 - \mathsf{ct}_{\mathsf{bits},2})$
14: $\qquad \mathsf{ct}'_{\mathsf{db}_2} \leftarrow \mathsf{ct}'_{\mathsf{db}_2,1} + \mathsf{ct}'_{\mathsf{db}_2,2} + \mathsf{Rotate}(\mathsf{ct}_{\mathsf{db}_2,1} - \mathsf{ct}'_{\mathsf{db}_2,1} + \mathsf{ct}_{\mathsf{db}_2,2} - \mathsf{ct}'_{\mathsf{db}_2,2}, -L)$
15: $\qquad$ **Return** $\mathsf{ct}'_{\mathsf{db}_1}, \mathsf{ct}'_{\mathsf{db}_2}$

16: **procedure** $\mathsf{SingleShuffle}_2(\mathsf{pk}_{\mathsf{BFV}}, \mathsf{ct}_{\mathsf{bits}}, \mathsf{ct}_{\mathsf{db}_1}, \mathsf{ct}_{\mathsf{db}_2})$ $\qquad\qquad$ ▷ All operations are under BFV
17: $\qquad \mathsf{ct}_{\mathsf{db}_1,1} \leftarrow \mathsf{ct}_{\mathsf{db}_1} \cdot \mathsf{ct}_{\mathsf{bits}}$
18: $\qquad \mathsf{ct}_{\mathsf{db}_1,2} \leftarrow \mathsf{ct}_{\mathsf{db}_1} - \mathsf{ct}_{\mathsf{db}_1,1}$
19: $\qquad \mathsf{ct}_{\mathsf{db}_2,1} \leftarrow \mathsf{ct}_{\mathsf{db}_2} \cdot (\vec{1} - \mathsf{ct}_{\mathsf{bits}})$
20: $\qquad \mathsf{ct}_{\mathsf{db}_2,2} \leftarrow \mathsf{ct}_{\mathsf{db}_2} - \mathsf{ct}_{\mathsf{db}_2,1}$
21: $\qquad \mathsf{ct}'_{\mathsf{db}_1} \leftarrow \mathsf{ct}_{\mathsf{db}_1,1} + \mathsf{ct}_{\mathsf{db}_2,1}$
22: $\qquad \mathsf{ct}'_{\mathsf{db}_2} \leftarrow \mathsf{ct}_{\mathsf{db}_1,2} + \mathsf{ct}_{\mathsf{db}_2,2}$
23: $\qquad$ **Return** $\mathsf{ct}'_{\mathsf{db}_1}, \mathsf{ct}'_{\mathsf{db}_2}$

24: **procedure** $\mathsf{bfvThorp}(\mathsf{db} = (\mathsf{db}[1], \ldots, \mathsf{db}[N]), r, \mathsf{pp}_{\mathsf{BFV}}, \mathsf{pk}_{\mathsf{BFV}}, (\mathsf{ct}_{\mathsf{bits},i,j})_{i \in [(r+1)n], j \in [N/D/2]})$
25: $\qquad \mathsf{ct}_{\mathsf{db},j} \leftarrow \mathsf{BFV.Enc}(\mathsf{pk}_{\mathsf{BFV}}, (\mathsf{db}[D \cdot j + 1], \ldots, \mathsf{db}[D \cdot (j+1) - 1])$ for $j \in [0, N/D - 1]$
26: $\qquad$ **for** t in $[1, \ldots, r+1]$ **do**
27: $\qquad\qquad$ **for** ℓ in $[1, \ldots, n]$ **do**
28: $\qquad\qquad\qquad$ **if** $2^{\ell-1} < D$ **then**
29: $\qquad\qquad\qquad\qquad$ **for** j in $[1, \ldots, N/D/2]$ **do**
30: $\qquad\qquad\qquad\qquad\qquad \mathsf{ct}'_{\mathsf{db},2j-1}, \mathsf{ct}'_{\mathsf{db},2j} \leftarrow \mathsf{SingleShuffle}_1(\mathsf{pk}_{\mathsf{BFV}}, \mathsf{ct}_{\mathsf{bits},(t-1)\cdot n+\ell,j}, \mathsf{ct}_{\mathsf{db},2j-1}, \mathsf{ct}_{\mathsf{db},2j}, \ell)$
31: $\qquad\qquad\qquad$ **else**
32: $\qquad\qquad\qquad\qquad L \leftarrow 2^{\ell-1}/D$
33: $\qquad\qquad\qquad\qquad$ **for** j in $[1, \ldots, N/D/2/L]$ **do**
34: $\qquad\qquad\qquad\qquad\qquad$ **for** k in $[1, \ldots, L]$ **do**
35: $\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ct}'_{\mathsf{db},2((j-1)L+k)-1}, \mathsf{ct}'_{\mathsf{db},2((j-1)L+k)} \leftarrow \mathsf{SingleShuffle}_2(\mathsf{pk}_{\mathsf{BFV}}, \mathsf{ct}_{\mathsf{bits},(t-1)\cdot n+\ell,(j-1)L+k},$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ct}_{\mathsf{db},(j-1)L+k}, \mathsf{ct}_{\mathsf{db},jL+k})$
$\qquad\qquad \mathsf{ct}_{\mathsf{db},\cdot} \leftarrow \mathsf{ct}'_{\mathsf{db},\cdot}$
36: $\qquad$ **return** $(\mathsf{ct}_{\mathsf{db},j})_{j \in [N/D]}$

4.3 Putting Everything Together for Homomorphic Thorp Shuffle

With all these tools, we now present our homomorphic Thorp Shuffle algorithm in Algorithm 6. Essentially, we use the PRG we presented in Algorithm 4 for generating the randomness needed by Thorp Shuffle (so sample $s \leftarrow_\$ \mathbb{Z}_t^n$ as seed, where n is some security parameter). Then, we use this seed to homomorphically generate $(r+1)\log(N) \cdot N$ bits. Lastly, we use these bits to homomorphically perform the Thorp Shuffle.

Algorithm 6 Oblivious Permutation

Database db size $N \in \mathbb{N}$ if a power of two. $n = \log_2(N)$.

$f_{A_1,\ldots,A_k}$ is a publicly known PRG (defined as in Algorithm 4).

1: **procedure** Setup(λ, q, N)
2:     Select BFV parameter $\mathsf{pp}_{\mathsf{BFV}} = (D, t, \ldots)$ such that the following computation can be done homomorphically and is semantically secure. $\triangleright$ BFV parameters other than D, t are irrelevant to us so we ignore it for simplicity.
3:     Generate BFV key pairs $\mathsf{pk}_{\mathsf{BFV}}, \mathsf{sk.BFV}$.
4:     Choose the minimum k such that $t^{-k} = \mathsf{negl}(\lambda)$
5:     Choose the minimum $n = \mathsf{poly}(\lambda)$ such that $\mathsf{LWR}_{n,t,2,R_i}$ holds for $i \in [k]$.
6: $\triangleright R_i$ defined in Theorem 4.3
7:     $s \leftarrow \mathbb{Z}_t^n$ $\triangleright$ The PRG random seed.
8:     $\mathsf{ct}_s \leftarrow \mathsf{BFV.Enc}(\mathsf{pk}_{\mathsf{BFV}}, s)$
9:     Choose the minimum t such that $\frac{2q(n+t)}{n+1}\left(\frac{2qn}{N}\right)^{t/(2(n+1))}$ is negligible in λ (recall $N = 2^n$).
10:     **return** $\mathsf{pp} = (\mathsf{pp}_{\mathsf{BFV}}, \mathsf{pk}_{\mathsf{BFV}}, \mathsf{pp}_{\mathsf{ExactSampler}}, \mathsf{ct}_s, t), \mathsf{sk.BFV}$
11: **procedure** HomomorphicThorp$(\mathsf{db}, \mathsf{pp})$
12:     Let $m = N\log(N)r$
13:     Homomorphically evaluate $f_{A_1,\ldots,A_k}$ to generate $(\mathsf{ct}_{\mathsf{bits},i,j})_{i \in [(r+1)\log(N)], j \in [N/D/2]}$ ciphertexts each containing N random bits.
14:     $\mathsf{db}_{\mathsf{ct}} \leftarrow \mathsf{bfvThorp}(\mathsf{db}, r, \mathsf{pp}_{\mathsf{BFV}}, \mathsf{pk}_{\mathsf{BFV}}, (\mathsf{ct}_{\mathsf{bits},i,j}))$
15:     **return** $\mathsf{db}_{\mathsf{ct}}$

Theorem 4.4. *If BFV with ring dimension D and plaintext t and among public parameters $\mathsf{pp}_{\mathsf{BFV}}$ is correct and semantically secure, for any database db of size $N = \mathsf{poly}(\lambda)$ being a power of two and that N/D is even, if $f_{A_1,\ldots,A_k}$ is a (t,m)-PRG (Definition 2.2) where m chosen in line 12, then for any adversary $\mathcal{A}$, any $q < N$, let $\mathsf{pp}, \mathsf{sk} \leftarrow \mathsf{Setup}(1^\lambda, q, N)$ and $\mathsf{ct} \leftarrow \mathsf{HomomorphicThorp}(\mathsf{db}, \mathsf{pp})$ (both procedures in Algorithm 6), and $\mathsf{db}_2 \leftarrow \pi(\mathsf{db})$ where π is a truly random permutation, let $\mathsf{db}_1 \leftarrow \mathsf{BFV.Dec}(\mathsf{sk}, \mathsf{ct})$, it holds that $|\Pr[\mathcal{A}^{\mathcal{O}(\mathsf{db}_1,q)}(\mathsf{pp}, \mathsf{ct}, \mathsf{db}) = 1] - \Pr[\mathcal{A}^{\mathcal{O}(\mathsf{db}_2,q)}(\mathsf{pp}, \mathsf{ct}, \mathsf{db}) = 1]| = \mathsf{negl}(\lambda)$, where $\mathcal{O}(\mathsf{db}, q)$ means an oracle access to an arbitrary location of string db for at most q times adaptively.*

Proof. This proof is trivial it is directly implied by the correctness of BFV, semantic security of BFV, (t,m)-PRG, and Theorem 3.2. $\square$

4.4 Removing the Linear Bandwidth Constraint in Our PIR Scheme

In this subsection, we discuss how to remove the linear bandwidth constraint from ThorPIR in Algorithm 2. Notice that all we have to do is to apply our oblivious permutation construction from

Section 4 to the preprocessing phase. Specifically, rather than download the whole database and compute the hints itself, the client can sample public parameters and send these to the server. The server can then obliviously compute the permutation using Algorithm 6. We present ThorPIR in Algorithm 7.

Algorithm 7 ThorPIR

1: **procedure** PREPROCESS(DB $\in \{0,1\}^N, \lambda$, T)
2: Let $K := N/Q$. DB $= \mathsf{db}_1, \ldots, \mathsf{db}_Q$ where each $\mathsf{db}_i \in \{0,1\}^K$ (shared by both the client and the server).
3: Client runs $\mathsf{pp}_{\mathsf{obliviousPerm}} \leftarrow \mathsf{Setup}(\lambda, T, N)$ (Algorithm 6) and sends $\mathsf{pp}_{\mathsf{obliviousPerm}}$ to the server
4: The client also sends Q PRGs $f_1, \ldots, f_Q$ (defined by seeds $s_1, \ldots s_Q$).
5: The server runs $\mathsf{db}_{\mathsf{ct},i} \leftarrow \mathsf{HomomorphicThorp}(\mathsf{db}_i, \mathsf{pp})$ with f_i for $i \in [Q]$
6: The server returns $\sum_{i \in [Q]} \mathsf{db}_{\mathsf{ct}_i}$
7: The client decrypts and store hints $h_1, \ldots h_{N/Q}$.

8: QUERY, ANSWER, RECONSTRUCT are exactly the same as in Algorithm 2. Note that now the i-th Thorp shuffle is defined by f_i (and the seed encrypted inside BFV sent to the server).

.

We then define Theorem 4.5. In this theorem, an update refers to the time it takes to update the client's hint, given an update to a certain database element.

Theorem 4.5. *The PIR scheme described in Algorithm 7 is correct Definition 2.5 and private Definition 2.6 for T queries, and runs with the following complexities:*

- $O_\lambda(N)$ *preprocessing time with* $O_\lambda(1)$ *computation depth.*

- $O(Q + N/Q)$ *offline bandwidth.*

- $O_\lambda(Q)$ *online bandwidth and query time.*

- $O(N/Q + T \cdot Q)$ *client storage.*

- $O_\lambda(1)$ *update time*

Proof. The proof follows directly from the proof of Theorem 3.1 and Theorem 4.4. □

5 Concrete Efficiency

Here, we estimate the depth and runtime of (1) our construction (Theorem 4.5), (2) prior works with linear offline bandwidth [70, 53], and (3) prior works with sublinear offline bandwidth [22, 40, 69] and present them in Table 2. We first focus on ThorPIR with proven security (Theorem 4.5) and discuss ThorPIR with conjectured security (Conjecture 3.3) in more detail at the end.

Parameter selection. We estimate times on a database of $N = 2^{30}$ entries, each with 360 bits, resulting in a 360GB database. And we estimate our parameters with ≥ 128 bits of computational security and ≥ 40 bits of statistical security (which is a commonly used statistical security guarantee, e.g. [21, 17, 60, 37, 45]).

We choose $Q = 2^{10}, K = 2^{20}$ as the parameters for our PIR scheme (allowing $Q = 2^{10}$ queries before re-doing the preprocessing); and choose $r = 480$ (i.e., the number of levels of Thorp shuffle) according to Theorem 3.2 such that the adversary querying for q queries have at most 2^{-40} advantage. For the underlying BFV, we choose $D = 2^{15}$ for the ring dimension; 860 bits for the ciphertext modulus; and $t = 65537$ for the plaintext modulus. Since there are no concrete LWR security estimators, we heuristically estimate the security of our construction using the LWE estimator [2].

Then, we setup parameters for our LWR-based PRG. To do this, we make two heuristic assumptions: (1) $\mathsf{LWR}_{n,t,2,R_i}$ is equivalent to $\mathsf{LWR}_{n,t,2,\lfloor\cdot\rceil}$ for any R_i ($i \in [k]$) in Algorithm 4; and (2) $\mathsf{LWR}_{n,t,2,\lfloor\cdot\rceil}$ is equivalent to LWE with secret dimension n, ciphertext modulus q, and error from Gaussian distribution with standard deviation σ such that $\Pr_{e \leftarrow \chi_\sigma}[|e| < t/2] > 1 - \mathsf{negl}(\lambda)$.[5] Under these two heuristic assumptions, we set $n = 220, t = 65537$ (and we use $\sigma = 128$ which is more than sufficient to satisfy the conditions). Additionally, we set $k = \log_t(2^{80}) = 5$ (i.e., the number of repetitions for random bit sampling), to obtain a statistical security parameter greater than 40-bits even after sampling $Nr/2 = 2^{30} \cdot 480/2 < 2^{38}$ random bits. With these parameters, we estimate our construction cost accordingly. See Section 5.1 for a detailed explanation of how we estimate the preprocessing runtime of our construction.

Estimation for other constructions. We estimate the preprocessing time of prior works using FHE with sublinear bandwidth [22, 69, 40] generously. In particular, we *only* estimate the number of FHE bootstrappings needed. See Section 5.1 for more details.

The online times for PIANO and MIR are extrapolated from the numbers found in [53]. Since there is no implementation for the schemes in [22], we extrapolate using the closest matching schemes to their online time [53]. Since the online times are very similar between these schemes and the only change is the number of elements accessed, we believe that extrapolating from [53] and adjusting for the number of random accesses gives us a very reasonable estimate of the online time for both of these schemes. Finally, we extrapolate the running time for ThorPIR from the implementation of [42].

Concrete server costs. Lastly, we estimate the server's cost for having 128K GPUs. According to [56], their benchmark is using RTX 3060Ti. Therefore, our estiamtion requires 128K GPUs that have comparable computational power to RTX 3060Ti. According to `https://akash.network/gpus/`, renting one RTX3060 Ti for an hour costs $0.09/hr on average. Thus, per client preprocessing costs about 25353 dollars. Unfortunately, this is still very high. In practice, client may choose to either streaming the whole database or pay the server this amount for the service. With our conjectured security, this cost can be brought down to roughly $172, which is much more reasonable, which we discuss in more detail later.

Missing values. We denote depth and server time for PIANO and MIR as N/A because the preprocessing is done in plaintext at the client end.

We also do not include query time estimates for ZLTS and LP because there is no known implementation of the privately puncturable PRF used in both schemes, we do however assume that it will be *significantly* slower than other schemes. A back of the envelope calculation on the privately puncturable PRF key sent in the online phase of ZLTS and LP given given the parameters in [10, 16] show us that the keys have size at least $O(\lambda^4 \log \lambda \log N)$. Hence, we estimate the online bandwidth of both those schemes to be at least that large.

---

[5][5] specifies the known best attacks for LWR, which are all included by [2] to estimate the security of the corresponding LWE instances. Thus we believe this heuristic estimation is reasonable.

Depth. As shown in the Table 2, our depth is concretely more than *six orders of magnitude* better than previous FHE-based constructions (CHK2, ZLTS, LP). This gives two consequantial benefits: (1) to perform our computation over FHE, we need to bootstrap a lot less times, which greatly increases our efficiency; (2) the circuit becomes inherently parallelizable compared to prior works: we can easily parallelize over 128K GPUs (in fact, we can parallelize over $2^{30}/2^{16} \cdot 2880/3 > 2^{27}$ computation instances), while prior FHE-based works cannot parallelize at all.

End-to-end preprocessing time. In terms of preprocessing, for 30MB/sec bandwidth,[6] MIR [53] and PIANO [70] takes about 3.4 hours for preprocessing, while our construction takes only about 2 hours given the setting we consider (i.e., the server is super powerful). Our advantages are even more obvious when the bandwidth is limited: for 10MB/sec bandwidth, both MIR and PIANO take more than 10 hours. Moreover, if the server is even more powerful, as mentioned above, ThorPIR can be even parallelized over more instances. For example, with 1M GPUs, our end-to-end time can be reduced to less than half an hour (with 30MB/sec bandwidth), while the prior work would still remain hours.

Online time and update time. We note that our online time is better than other schemes, This is expected from the asymptotics table in Table 1.

Although we do not include update time in Table 2, our update time is almost non-existent since it requires $O_\lambda(1)$ PRF evaluations and two exclusive or operations. In previous schemes benchmarked, the update time for each entry costs about as much as a query, given that clients have to go through each hint preprocessed to check whether that hint requires an update. This translates to time savings of orders of magnitude when comparing ThorPIR to the previous.

ThorPIR with conjectured security. Lastly, we discuss our construction with conjectured security. As conjectured in Conjecture 3.3., to acheive the computational security of 128 bit, with $K = 2^{20}$, we have $\lambda = 45$ being the number of levels we need to do the Thorp shuffle. Thus, we remove the use of bootstrapping completely, since there are only $\lambda = 45$ levels needed for the Thorp shuffle (and ${\sim}20$ levels for the PRG). Instead, we set ring dimension $D = 2^{17}$, plaintext modulus $t = 786433$, and ciphertext modulus to contain 3000 bits of noise budget (which guarantees > 128-bit of computational security [2]). Then, we re-estimate our runtime (see Section 5.1) for more details.

Then, it is easy to see that with such conjectured security, our preprocessing runtime decreases by about 200x. This means that with such a powerful server, our scheme is much faster than any prior construction. The concrete dollar costs are also brought down to \$172 per client. Even for a weaker server (e.g., with only 1024 RTX 3060 Ti GPUs), our server runtime (and end-to-end time) is still only about 1.9 hours (with a similar dollar cost).

5.1 Details on Estimation

In this section, we discuss how we estimate the runtime of our constructions. We assume familiarity with the operations in BFV [13, 27] and their corresponding implementation.

Runtime of basic operations. We first tested some basic operations on GCP instance N2 with CPU Intel Xeon Gold 6268, 64GB RAM. With ring dimension $D = 32768$, and ciphertext modulus being a 860 bit prime, a single thread, the operations runtimes are as follows (with SEAL [49] library):

---

[6]According to `https://www.speedtest.net/global-index`, in March 2024, the average internet speed in the USA is around 250Mb/sec, and thus we use 30MB/sec as a representative value.

Scheme	Preprocess					Query		Client Space
	Depth	BW	Cli Time	Serv Time	E2E Time	BW	Time (total)	
PIANO [70]	N/A	360GB	1.9H	N/A	3.4H	5.93MB	35.3ms	1.74GB
MIR[53]	N/A	360GB	1H	N/A	3.4H	62.1KB	10.3ms	1.42GB
CHK1[22, Thm 4.1]	1	5.7TB	1H	0.3H	52.8H	11.9MB	1.9s	1.9TB
CHK2[22, Thm 5.1]	$> 2^{30}$	2.4GB	65.8s	$> 19{,}000$H	$> 19{,}000$H	382KB	10.3ms	1.8GB
ZLTS[69]	$> 2^{30}$	2.4GB	65.8s	$> 19{,}000$H	$> 19{,}000$H	> 0.5GB	–	1.75GB
LP[40]	$> 2^{30}$	2.4GB	65.8s	$> 19{,}000$H	$> 19{,}000$H	> 0.5GB	–	1.75GB
ThorPIR Theorem 4.5 Proven security	480	1.4GB	38.4s	2.16H	2.17H	389KB	3.6ms	377MB
ThorPIR Conjecture 3.3 Conjectured security	45	1.4GB	38.4s	0.015H	0.03H	389KB	3.6ms	377MB

Table 2: Concrete numbers for ThorPIR compared to previous state-of-the-art single-server client-preprocessing PIR schemes for a database of 360GB (2^{30} elements of 360 bytes each). We assume that servers are run with 128K GPUs and that clients are run over 8-thread CPUs. We use BW to denote bandwidth, E2E time to denote total end-to-end time of running the preprocessing phase, assuming 30MB/sec bandwidth. Note that we do not count one-time FHE public keys as preprocessing BW since it can be used repeatedly for multiple preprocessings and thus amortized further. However, note that the public keys are < 1GB and thus have no effect on our conclusion.

- Ciphertext multiplication (including relinearization): 315 milli-seconds.

- Plaintext multiplication: 27 milli-seconds

- Plaintext multiplication *without* NTT: 1.5 milli-seconds

- Addition: 0.68 milli-seconds

- Rotation: 125 milli-seconds

- Decryption: 2.5 milli-seconds

- Relaxed bootstrapping with valid plaintext space being 3 bits introduced in [46]: 43 seconds

Estimating our PRG runtime. We start with estimating the runtime of our PRG Algorithm 4. Recall that we have $n = 220$, and thus the first step requires 220 plaintext multiplications (without NTT, since we can provide the ciphertext encrypting the seed in NTT form)[7]. Then, since $t = 2^{16}+1$ it takes 16 square which takes 16 ciphertext multiplications. Then, since $k = 5$, this process is repeated 5 times, and additional $k - 1 = 4$ ciphertext multiplications are needed. Lastly, for every two levels of ciphertext multiplications, we can modulus switching the ciphertext down by 60 bits[8], which slightly decreases the time (roughly by $60/Q$ where Q is the number of bits of the ciphertext modulus left, which is 800 at the beginning)[9]. By counting all these factors, our PRG takes about 23 seconds. This allows about 210 bits of noise left, allowing an additional 7 levels of ciphertext

---

[7]Or alternatively the server could preprocess this seed once for all the following random bit generation, and thus the amortized time is almost 0.

[8]The ciphertext modulus of 860 bits is composed of 16 primes each with 60 bits. Modulus switching for a time gets rid of one prime.

[9]Note that we start with the ciphertext modulus being 860, but 60 of which is the modulus for key generation, and thus each ciphertext by itself starts with 800.

multiplications. If reducing the input Q from 860 to 680 bits (i.e., only one level of multiplication left), the runtime reduces to roughly 12.3 seconds (single-thread CPU).

Estimating our PIR preprocessing runtime. For PIR preprocessing, recall that the ring dimension is 32768, which means that each PRG generation generates $32768 = 2^{15}$ random bits. Per our Thorp shuffle algorithm Algorithm 5, each 2^{15} random bits can swap two ciphertexts each containing 2^{15} elements (thus in total 2^{16} elements). Each swap takes 4 ciphertext multiplications, 4 rotations, 4 plaintext multiplications (with NTT), and 10 additions; and noise-wise, it takes one level of ciphertext multiplication followed by one level of plaintext multiplication, thus in total consuming about 50 bits of noise budget. Thus, a single level of Thorp shuffle runtime is the time for these operations plus one PRG call.

Each relaxed bootstrapping gives about 400 bits of noise left, thus allowing 8 levels of Thorp shuffle. To compute 480 levels of Thorp shuffle, it takes $\lceil 480/8 = 55 \rceil$ bootstrapping. Furthermore, each level has $2^{20}/2^{16} \cdot 2880/3 = 7680$ (recall that each entry has 2880 bits and each plaintext space allow 3 bits) ciphertexts. Thus, each level requires 7680 one-level Thorp shuffles (on average, performing on the ciphertexts with Q being 200 bits).

Then, per [56], GPU can accelerate BFV operations by about 50x, and thus with a single GPU, the preprocessing is about 21666 hours. Lastly, since our construction is highly parallelizable (can be fully parallelized over $2^{30}/2^{16} \cdot 2880/3$ instances), with 131072 (128K) GPUs, our preprocessing time becomes ~ 2 hours.

Our offline bandwidth and client time. As mentioned, each BFV ciphertext can contain at most $32768 \cdot 3$ bits. In total, we return $2^{20} \cdot 2880/32768/3$ ciphertexts, each of which contains two ring elements with each coefficient being 12 bits (we can modulus switching down the ciphertext to have 12 bits before returning). Thus, the bandwidth is $2^{20} \cdot 2880/32768/3 \cdot 32768 \cdot 12 \cdot 2 = 1.4$GB. Our client time is simply decrypting all these ciphertexts, each of which takes $2^{20} \cdot 2880/32768/3 \cdot 2.5/1000 = 38.4$ seconds single-thread. For eight threads, it is at least 4x faster (with at least four physical cores), which gives 9.6 seconds.

Estimating preprocessing time in prior works [22, 40, 69]. To be extremely generous, we only estimate the number of bootstrapping needed for prior works. The depth in prior work is $> 2^{30}$ so we estimate it using 2^{30} (each level using 30 bits of noise budget, which means that each relaxed bootstrapping allows about 13 levels of computation): which means at least $2^{30}/13$ bootstrapping operations are needed, each taking 0.86 seconds on a GPU, which results in 19,731 hours. This work cannot be parallelized so that is the time it takes to preprocess.

Estimating preprocessing time for ThorPIR with conjectured security. Now, for our scheme with conjectured security, we do not need bootstrapping, but instead requires a larger ring dimension $D = 2^{17}$. Unfortunately, SEAL [49] does not allow such a large ring dimension. Therefore, we estimate the runtime of each operation by the theoretic cost, which is $D \log(D)$. Then, instead of using $t = 65537$, we need a larger t since we need $t > D$ for the BFV parameter setting. We choose $t = 786433$, a 20-bit prime. Thus, our PRG changes k from 5 to 4. Furthermore, since we do not need relaxed bootstrapping, we can fully utilize the entire plaintext space (i.e., encode 20 bits per slot). In total, there are 45 levels of Thorp shuffle, requiring about 2250 bits of noise budget. Then, the PRG requires about 750 bits of noise budget. In total, about 3000 bits of noise budget is needed. With these, we re-estimate our construction similar to what we do above. The total runtime reduces to about 0.015 hours.

6 Proof of Theorem 3.1

In this section, we set out to prove Theorem 3.1.

6.1 A Privacy Theorem

We start by showing indistinguisability between two experiments which will exactly model our privacy requirement. We separate this part out (i.e., separate it from correctness and efficiency) since it is the most involved part of the proof. The privacy theorem we need is defined below in Theorem 6.1,

Theorem 6.1. *For any PPT adversary $\mathcal{A}$, Experiment 0 and Experiment 2 in Figure 1 are indistinguishable for any $T = o(N)$ (except with negligible probability), as long as the underlying Thorp shuffle is adaptive CCA secure with T queries (except with negligible probability) .*

Proof. It is straightforward that Experiment 0 outputs are independent of $(x_i, y_i)_{i \in [T]}$ thus guaranteeing perfect privacy. It is also straightforward that Experiment 2 is equivalent to our construction in Figure 1. To prove that these two experiments are indistinguishable to any PPT adversary $\mathcal{A}$, we define Experiment 1.

Note that the only difference between Experiments 1 and 2 is that we change all the independently sampled Thorp shuffles to independently sampled perfect random permutations. Thus, if there exists an $\mathcal{A}$ who can distinguish Experiments 1 and 2, we can construct an adversary $\mathcal{A}'$ breaking the T-CCA-security of the Q Thorp shuffles (i.e., distinguishing Q Thorp shuffles from Q random permutations using T adaptive queries to the shuffle and the inverse of the shuffle) as follows: given a T adaptive oracle accesses of each of Q permutations $P_1, \ldots, P_Q$ (and their inverses), either Q Thorp shuffles or Q random permutations:

- Initialize $\mathsf{USED}_i = \varnothing$ for all $i \in [Q]$.

- For $t \in [T]$:

    - Upon receiving (x_t, y_t) from $\mathcal{A}$,
    - Call the oracle to get $j_t \leftarrow P_{x_t}^{-1}(y_t)$
    - Call the oracle to get $o_i \leftarrow P_i(j_t)$ for $i \in [Q] \setminus \{x_t\}$
    - If $o_i \in \mathsf{USED}_i$, sampled o_i uniformly at random from $[K] \setminus \mathsf{USED}_i$ for $i \in [Q] \setminus \{x_t\}$, and also sampled $o_{x_t} \leftarrow_\$ [K] \setminus \{x_t\}$
    - Add o_i to USED_i for all $i \in [Q]$.
    - Feed $(o_1, \ldots, o_Q)$ to $\mathcal{A}$

- If $\mathcal{A}$ returns Experiment 0, return that $P_1, \ldots, P_Q$ are uniformly random permutations; otherwise, return that they are Thorp shuffles.

It is straightforward that if $P_1, \ldots, P_Q$ are uniformly random permutations, $\mathcal{A}$ sees exactly Experiment 1; otherwise, it sees exactly Experiment 2. Thus, $\mathcal{A}'$ distinguishes Q Thorp shuffles with Q uniformly random permutations with the same non-negligible probability as $\mathcal{A}$ distinguishes the two experiments, using T queries to each Thorp shuffle.[10] Lastly, note that by the CCA security

---

[10]Note that here we are using a Thorp shuffle using PRG-generated randomness, which should be indistinguishable from a truly random Thorp shuffle for any PPT adversary.

Experiment 0:

1. Experiment initializes $\mathsf{USED}_i = \varnothing$ for $i \in [Q]$.

2. **For** $t \in [T]$:

   (a) Sample z_t uniformly from $[K] \setminus \mathsf{USED}$.

   (b) Adversary outputs $(\cdot, \cdot) \in [Q] \times [K]$.

   (c) Experiment samples o_i uniformly random from $[K] \setminus \mathsf{USED}_i$ for $i \in [Q]$ and adds o_i to USED_i.

   (d) **Output** $(o_1, \ldots, o_Q)$.

Experiment 1

1. Experiment samples Q permutations $p_1, \ldots, p_Q$ uniformly from the set of permutations of $[K]$.

2. Experiment initializes $\mathsf{USED}_i = \varnothing$ for $i \in [Q]$.

3. **For** $t \in [T]$:

   (a) Adversary outputs $x_t, y_t \in [Q] \times [K]$

   (b) Let $j_t = p_{x_t}^{-1}(y_t)$.

   (c) Let $o_i \leftarrow p_i(j_t)$ for $i \in [Q] \setminus \{x_t\}$.

   (d) If $o_i \in \mathsf{USED}_i$, sampled o_i uniformly at random from $[K] \setminus \mathsf{USED}_i$ for $i \in [Q] \setminus \{x_t\}$, and also sampled $o_{x_t} \leftarrow_\$ [K] \setminus \{x_t\}$.

   (e) Add o_i to USED_i for all $i \in [Q]$.

   (f) **Output** $(o_1, \ldots, o_Q)$.

Experiment 2

1. Experiment samples $s_1, \ldots, s_Q$ where each $s_i = \mathsf{Th.Gen}(\lambda, N/Q, T)$ (Q Thorp Shuffles).

2. Experiment initializes $\mathsf{USED}_i = \varnothing$ for $i \in [Q]$.

3. **For** $t \in [T]$:

   (a) Adversary outputs $x_t, y_t \in [Q] \times [K]$.

   (b) Let $j_t = \mathsf{Th.Inv}(s_{x_t}, y_t)$.

   (c) Let $o_i \leftarrow \mathsf{Th.Eval}(s_i, j_t)$ for $i \in [Q] \setminus \{x_t\}$.

   (d) If $o_i \in \mathsf{USED}_i$, sampled o_i uniformly at random from $[K] \setminus \mathsf{USED}_i$ for $i \in [Q] \setminus \{x_t\}$, and also sampled $o_{x_t} \leftarrow_\$ [K] \setminus \{x_t\}$.

   (e) Add o_i to USED_i for all $i \in [Q]$.

   (f) **Output** $(o_1, \ldots, o_Q)$.

Figure 1: Experiments, where Experiment 0 is the ideal world with no privacy loss, and Experiment 2 is our real construction in Algorithm 2. Experiment 1 is a hybrid we need in the proof.

of Thorp shuffle, one can distinguish a Thorp shuffle from a uniformly random permutation with $\mathsf{negl}(\lambda)$ probability, and since $Q = \mathsf{poly}(\lambda)$, one can also only distinguishes Q Thorp shuffles with Q uniformly random permutations with $Q \cdot \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$ probability, which reach a contradiction. Thus, Experiments 1 and 2 are indistinguishable.

Thus, the only thing left to prove is that Experiments 0 and 1 are indistinguishable. To prove this, we use a sequence of hybrids.

- Hyb_1: same as Experiment 0, except that if $t = 1$, do step 3 in Experiment 1 instead of the step 3 in Experiment 0.

- Hyb_2: same as Hyb_1, except that if $t = 2$, do step 3 in Experiment 1 instead of the step 3 in Experiment 0.

- ...

- Hyb_{T-1}: same as Hyb_{T-2}, except that if $t = T - 1$, do step 3 in Experiment 1 instead of the step 3 in Experiment 0.

- Hyb_T: same as Experiment 1.

Experiment 1 and Hyb_1 are indistinguishable due to the following (recall that $t = 1$): since $p_1, \ldots, p_Q$ are independently drawn and all are uniformly random permutation, j_1 is a random element in $[K]$ and indistinguishable from z_1 even given (x_1, y_1), and thus $p_i(j_1)$ is indistinguishable from $p_i(z_1)$ for $i \neq x_1$; thus $(o_1, p_2(j_1), \ldots, p_Q(j_1))$ is indistinguishable from $(p_1(z_1), \ldots, p_Q(z_1))$, as o_1 is sampled uniformly at random from $[K]$ independent of $p_2(j_1), \ldots, p_Q(j_q)$.

Then, Hyb_1 and Hyb_2 are indistinguishable as follows:

- if $(x_2, y_2) = (x_1, y_1)$: then every o_i is sampled from $[K] \setminus \mathsf{USED}_i$

- else if $x_2 = x_1$ but $y_2 \neq y_1$: j_2 is indistinguishable from uniformly random from $K \setminus \{j_1\}$, which means that $p_i(x_2)$ for $i \neq x_2$ is uniformly at random from $[K] \setminus \mathsf{USED}_i$, and furthermore o_{x_2} is uniformly at random from $[K] \setminus \mathsf{USED}_{x_2}$ trivially

- else: j_2 is uniformly at random from $[K]$, and thus $p_i(j_2)$ for $i \neq x_t$ is uniformly at random from $[K]$, which means that o_i is uniformly at random from $[K] \setminus \mathsf{USED}_i$ (since if $o_i \in \mathsf{USED}_i$, it is resampled from the rest), for $i \neq x_t$; and again, o_{x_t} is trivially uniformly at random from $[K] \setminus \mathsf{USED}_{x_t}$

Inductively, for a similar argument, it is straightforward to see that Hyb_{i-1} and Hyb_i are indistinguishable for $i \in [3, T]$. Lastly, since Hyb_T and Experiment 1 are indistinguishable, and $T = \mathsf{poly}(\lambda)$, we have Experiments 1 and 0 are indistinguishable. Combining the argument that Experiments 1 and 2 are indistinguishable, we conclude that Experiments 0 and 2 are indistinguishable.

$\square$

6.2 A Proof of Theorem 3.1

Below, we then proof of Theorem 3.1.

Proof. **Correctness:**   Follows by construction of the scheme. At each step we either retrieve the element by computing the correct result from our stored hints xored with the relevant elements sent to the server during query time, *or,* we retrieve the element locally if it was seen in a previous query request.

Efficiency: Preprocessing takes $O(\lambda N)$ time since the Thorp Shuffle requires λ rounds with N operations. Our scheme saves at the client N/Q hints plus Q new elements for each of the T queries performed, totaling $N/Q + T \cdot Q$ storage. Since the server streams the database to the client at preprocessing, the bandwidth offline is $O(N)$.

Note that evaluating the Thorp Shuffle or its inverse requires $O(\lambda)$ operations, therefore, to execute our query step, which requires taking one inverse Thorp Shuffle and $Q - 1$ normal Thorp Shuffle evaluations. This takes $O(\lambda Q)$ time at the client, which then sends all the Q elements to the server (which indexes them and returns the elements in $O(Q)$ time). The client gets back the Q database values, totaling $Q \log N + Q$ bandwidth. The reconstruct time requires $O(Q)$ time to xor the elements and store them. Furthermore, performing a 'swap' requires an additional, separate datastructure to keep track of the swap since they are not natively supported by the Thorp Shuffle. Since we perform Q swaps in total, this adds an additional $O(Q)$ storage.

Finally, to update element $(q, k) \in [Q] \times [K]$, the server sends q, k, x, x' where x is the old database element and x' is the new element to be updated. The client computes $j \leftarrow \mathsf{Th.Inv}(s_q, k)$, and compute $\mathsf{st}.h_k = \mathsf{st}.h_k \oplus x \oplus x'$. Then, it also computes $\mathsf{st}.\mathsf{Used}_q[k] \leftarrow x'$. This together takes one inverse of Thorp shuffle, two xors, and a data structure update (e.g., a hash table), which together takes only $O_\lambda(1)$ time.

Privacy: Privacy, by Definition 2.6 requires defining an algorithm Sim which runs without knowledge of queries and is indistinguishable from real client queries for any PPT adversary $\mathcal{A}$. First, note that in the preprocessing phase, the client simply downloads the whole database to compute its hints so that cannot leak any information about the permutations it samples. Then, by Theorem 6.1, we can replace the online phase of scheme with Experiment 0 in Figure 1 and no adversary can distinguish between interacting with our real scheme or Experiment 0 except with negl probability. $\qquad\square$

7   Proof of the Improved Mixing Time of the Thorp Shuffle

Now we give the proof for Theorem 3.2. The initial setup of the proof follow closely the setup of Morris et al. [52].

Proof. First, we setup some variables and definitions we will need.

Let μ and ν be probability distributions on an event space Ω, where Ω is any finite non-empty set. We say that a pair of random variables $W = (X,Y)$ is a *coupling* of μ and ν if its marginal distributions (i.e., distributions of X and Y) are μ and ν respectively. That is, for any $S \subseteq \Omega$ we have $\mathbf{P}(X \in S) = \mu(S)$ and $\mathbf{P}(Y \in S) = \nu(S)$. The total variation distance between μ and ν can be expressed as:

$$\Delta(\mu, \nu) := \max_{S \subseteq \Omega} |\mu(S) - \nu(S)| = \min_{(X,Y):\ X \sim \mu \text{ and } Y \sim \nu} \mathbf{P}(X \neq Y) \tag{1}$$

where $X \sim \mu$ means that X has the distribution μ. The minimum is thus taken over all possible couplings of μ and ν. Further explanation on these statements can be found on standard texts on Markov Chains such as the book by Levin et al. [43, Section 4].

Now, let M_t be the Markov chain representing the Thorp Shuffle with N cards at step t, where **we define a card to be an element in** $C := \{0,1\}^n$ (recall $N = 2^n$ is a parameter)[11]. Then, the state space of M_t is any bijection from C to $\{0,1\}^n$. Let us define $M_t(z)$ to be the position of a card z at time t.

Since the adversary we are considering only ever sees a subset of q elements in the shuffle, we only need to bound the rate at which all q-subsets of the shuffle are indistinguishable from q-subsets of a real permutation. Let $z_1, \ldots, z_q$ be distinct cards. We define X_t to be the vector of random variables for the positions of cards $z_1, \ldots, z_q$ at time t on the Thorp Shuffle M_t. For $j \in \{1, \ldots, q\}$, $X_t(j)$ will represent the position of card z_j at time t.

Recall that we defined our process X_t as the position of a q-tuple vector of cards in M_t after t steps. Now we explicitly define the update rule from X_t to X_{t+1} in a self-contained fashion (without requiring knowledge of other cards' positions). X_1 is defined as the vector of the initial position of the q cards selected. Then, we will update each card's position according to the Thorp Shuffle definition (Algorithm 1). To define X_{t+1} in a self-contained fashion, we have to decide each card's next position in the q-tuple using only the information in X_t. To do this, we define an equivalent rule for generating X_{t+1} from X_t as follows.

We define two cards z_i and z_j **to be matched**[12] at a timestep t if their positions, u_i^t and u_j^t respectively, satisfy $u_i^t = u_j^t \mod (N/2)$.

For every card $z_j, j \in \{1, \ldots, \ell+1\}$, for each timestep t, we sample a coin $c_t^j \overset{\$}{\leftarrow} \mathsf{Bern}(1/2)$. We determine the position of card z_j at time $t+1$ as follows.

- **If** card z_j *is not matched to any card z_i where $i < j$*, it is moved to position $2(u_j^t \mod N/2) + c_t^j$.

- **Else If** card z_j *is* matched to a card z_i where $i < j$, it is moved to $2(u_j^t \mod N/2) + \neg c_t^i$.

For each card z_j, $j \in \{1, \ldots, \ell+1\}$, and each round t, we will also define a new variable d_t^j, where:

- $d_t^j = c_t^j$ **if** z_j is not matched to any z_i with $i < j$.

- $d_t^j = \neg c_t^i$ **if** z_j is matched to a card z_i with $i < j$, for $i \in \{1, \ldots, \ell+1\}$.

We say that d_t^j is **the coin associated with** card z_j at time t.

Next, we define another random process, U_t, to behave exactly as X_t, except we define U_1 to be q uniform samples without replacement from $\{0,1\}^n$, rather than to be defined from the cards' true initial state as in X_1.

Recall that our goal is to show that the Thorp shuffle (Algorithm 1) is indistinguishable from a real random permutation given only q (parallel) accesses.

Now, in order to bound the total variation distance between q-subsets of the Thorp Shuffle and a uniform permutation using the coupling above, we will introduce a lemma that relates the distance between the distributions to the expected distance between two distributions over conditional distributions.

---

[11] More generally, C can be any set of cardinality N, but here we restrict it to $\{0,1\}^n$ for concreteness.
[12] Two cards being matched at timestep t means that they will be placed next to each other in the next round.

Before introducing the lemma, we introduce some notation. For a distribution ν on distinct q-tuples of Ω, and $(Z_1, \ldots Z_\ell) \sim \nu$, we will use the following notation (at the risk of a slight abuse of notation):

$$\nu(u_1, \ldots u_\ell) = \mathbf{P}(Z_1 = u_1, \ldots, Z_\ell = u_\ell),$$

and

$$\nu(u_\ell \mid u_1, \ldots, u_{\ell-1}) = \mathbf{P}(Z_\ell = u_\ell \mid Z_1 = u_1, \ldots Z_{\ell-1} = u_{\ell-1}).$$

Note that $\nu(Z_\ell|u_1, \ldots, u_{\ell-1})$ is the distribution of Z_ℓ conditioned on $(Z_1 = u_1,\ldots,Z_{\ell-1} = u_{\ell-1})$ and thus $\nu(Z_{\ell+1}|Z_1, \ldots, Z_\ell)$ is a random variable over conditional distributions. For two q-tuple distributions μ and ν we will use the notation $\Delta(\mu(Z_{\ell+1}|Z_1, \ldots, Z_\ell), \nu(Z_{\ell+1}|Z_1, \ldots, Z_\ell))$ to denote the random variable representing the distance between the conditional distributions (*not to be confused with* the distance between the two random variables over conditional distributions), i.e. for any assignment $(u_1,\ldots,u_\ell)$ to $Z_1,\ldots,Z_\ell$ this random variable takes on a real-valued number $\Delta(\mu(Z_{\ell+1}|u_1, \ldots, u_\ell), \nu(Z_{\ell+1}|u_1, \ldots, u_\ell))$.

Lemma 7.1 (Lemma 2 in [52]). *Fix a finite nonempty set Ω and let μ and ν be probability distributions supported on $(\ell + 1)$-tuples of elements of Ω, and suppose that $(Z_1, \ldots, Z_q) \sim \mu$. Then,*

$$\Delta(\mu, \nu) \leq \sum_{\ell=0}^{q-1} \mathbf{E}\left(\Delta\left(\mu(Z_{\ell+1} \mid Z_1, \ldots, Z_\ell), \nu(Z_{\ell+1} \mid Z_1, \ldots, Z_\ell)\right)\right). \tag{2}$$

The proof of this lemma is shown in [52] so we omit the details.

Next, we will use this lemma to upperbound the total variation distance between the q-subsets of the Thorp Shuffle and a permutation. Define π to be the distribution of q uniform independent samples without replacement from $\{0,1\}^n$, and τ_t as the *distribution* of X_t. Note that with the update we define previously, for any $t \geq 1$, the distribution of X_t is exactly the same as the marginal distribution of the Thorp Shuffle (Algorithm 1; trivially by how the two processes are defined) over the q-subset. Furthermore, the distribution of U_t is always just π, since U_1 is q uniform samples and the Markov process converges to a uniform distribution with enough time [50]. Thus, by Equation (1), for any $\ell < q$ and any $u_1, \ldots, u_\ell$, and $(Z_1, \ldots, Z_{\ell+1}) \sim \tau_t$, we have:

$$\Delta(\tau_t(Z_{\ell+1}|Z_1 = u_1, \ldots, Z_\ell = u_\ell), \pi(Z_{\ell+1}|Z_1 = u_1, \ldots, Z_\ell = u_\ell)) \leq \mathbf{P}(X_t \neq U_t|Z_1 = u_1, \ldots, Z_\ell = u_\ell)$$
$$= \mathbf{P}(T_{\ell+1} > t|Z_1 = u_1, \ldots, Z_\ell = u_\ell).$$

Treating $\mathbf{P}(X_t \neq U_t|Z_1, \ldots, Z_\ell)$ as a random variable over probabilities, we can summarize this equation as:

$$\Delta(\tau_t(Z_{\ell+1}|Z_1, \ldots, Z_\ell), \pi(Z_{\ell+1}|Z_1, \ldots, Z_\ell)) \leq \mathbf{P}(X_t \neq U_t|Z_1, \ldots, Z_\ell))$$

By definition of our U_t and X_t processes, we have that:

$$\mathbf{P}(X_t \neq U_t \mid Z_1, \ldots, Z_\ell) = \mathbf{P}(T_{\ell+1} > t \mid Z_1, \ldots, Z_\ell),$$

where we define $T_{\ell+1} = \min\{t : X_t = U_t\}$, and again, the right-hand side is a random variable. With this, we conclude that the *expected* value of the variational distance is less than or equal to the unconditional probability that $T_{\ell+1} > t$, written as follows:

$$
\begin{aligned}
&\mathbf{E}\left(\Delta\left(\tau_t(Z_{\ell+1} \mid Z_1, \ldots Z_\ell), \pi(Z_{\ell+1} \mid Z_1, \ldots, Z_\ell)\right)\right) \\
&\leq \sum_{(u_1, \ldots, u_\ell)} \mathbf{P}\left(T_{\ell+1} > t \mid Z_1 = u_1, \ldots, Z_\ell = u_\ell\right) \mathbf{P}\left(Z_1 = u_1, \ldots, Z_\ell = u_\ell\right) \\
&= \mathbf{P}\left(T_{\ell+1} > t\right),
\end{aligned}
\tag{3}
$$

where each $u_i \in \{0,1\}^n$, for $i \in [\ell]$. Applying Lemma 7.1 with Equation (3), we get that:

$$
\Delta\left(\tau_t, \pi\right) \leq \sum_{\ell=0}^{q-1} \mathbf{E}\left(\Delta\left(\tau_t(Z_{\ell+1} \mid Z_1, \ldots, Z_\ell), \pi(Z_{\ell+1} \mid Z_1, \ldots, Z_\ell)\right)\right)
\tag{4}
$$

$$
\leq \sum_{\ell=0}^{q-1} \mathbf{P}(T_{\ell+1} > t),
\tag{5}
$$

therefore, for the rest of this proof this is what we will attempt to bound.

An important equation. Crucial to our proof will be the following equation [52, Page 8, eq 3], for any $t > n$:

$$
\mathbf{P}(z_i \text{ and } z_j \text{ are matched at time } t) \leq 2^{1-n}.
\tag{6}
$$

Recall that at each step t, for each card z_j, we flip a coin c_j^t for this card. If z_j is not matched to any z_i where $i < j$, then d_j^t, *the card associated with* z_j at round t is defined to be c_j^t; else, we define d_t^j to be $\neg c_t^i$. We will use Equation (6) to bound the probability that $d_t^j \neq c_t^j$ in the following equation.

In this equation, we bound the probability that card that the coin associated with card z_j at time t, d_j^t is not its own coin c_j^t to be less than or equal to 2^{-n}. Formally, for any $t > n$, for any $j \in \{1, \ldots q\}$,

$$
\mathbf{P}(d_t^j \neq c_t^j) \leq (j-1)2^{-n}
\tag{7}
$$

Proof. By definition, whenever z_j is not matched to any coin of smaller index, $c_j^t = d_t^j$. Furthermore, even if z_j is matched to some card of z_i where $i < j$, notice that if $c_i^t = \neg c_j^t$, it is still the case that $d_t^j = c_t^j$. So $d_t^j \neq c_t^j$ if and only if both (1) z_j is matched to some z_i for $i < j$ **and** (2) $c_t^i = c_t^j$. Notice that we can bound the probability of (1) to be less than or equal to $(j-1)2^{-n}$ using a union bound and Equation (6).

Also, the coins c_t^i, c_t^j are sampled from $\mathsf{Bern}(1/2)$, so the probability of (2) is $\frac{1}{2}$. Finally, the the positions of z_i and z_j at timestep t are independent from coins c_t^i, c_t^j, (since their positions are defined by timesteps up to $t-1$). Therefore, the events (A) $c_i^t = c_j^t$ **and** (B) whether z_i and z_j are matched are independent. So the probability of both happening is simply the product of the probability of each which is upper bounded by $(j-1)2^{1-n}\left(\frac{1}{2}\right) = (j-1)2^{-n}$. $\square$

Next, we will look at a second important lemma which, combined with Equation (7), will allow us to bound $P(T_{\ell+1} > t)$ for all $\ell \in \{0, \ldots, q-1\}$. This lemma tells us that a card's position is uniquely defined by that past n coins *associated with* that card, and any coin before that is not necessary to derive its position.

Lemma 7.2. *The past n coins associated with a card z uniquely define its current position u.*

Proof. This follows straight from the update rule defined. Consider a card z with position u_{t+1} at time $t+1$, with coins $d_{t-n+1}, \ldots, d_t$ used to update its position in the previous n timesteps. Then,

$$u_{t+1} = 2((2\left((2(u_{t-n+1} \mod N/2) + d_{t-n+1}) \mod N/2) + d_{t-n+2}\right)\ldots \mod N/2) + d_t$$
$$= \left(2^n (u_{t-n+1}) + \sum_{i=1}^{n} 2^{n-i} d_{t-n+i}\right) \equiv \sum_{i=1}^{n} 2^{n-i} d_{t-n+i} \mod N.$$

$\square$

Remark. ▶ An alternative view of Lemma 7.2 using bitwise operations may be help understanding. Given a card's current position as an $n-$bit string and its $t-$th coin d_t, each update chops off the most significant bit of the string, left shifts it by one position, and xors the bit d_t into it. From this perspective, it is clear to see that all the bits of a card's current position will be erased within n updates. ◀

Notice that Lemma 7.2 implies that, if $z_{\ell+1}$ is assigned using the same coins in both the X and U processes for n consecutive timesteps, then it must be that $z_{\ell+1}$ is in the same position in both processes. By what we just saw above, this means that if this happens, then $X_t = U_t$

We define event $A_{a:b}$ to be the probability that for some timestep t in $\{a, \ldots, b\}$, for any $i \in \{1, \ldots, \ell\}$, $z_{\ell+1}$ is associated with a different coin in processes X and U.

Recall from Equation (7), for any $t > n$, we have that $\mathbf{P}(d_t^{\ell+1} \neq c_t^{\ell+1}) \leq \ell \cdot 2^{-n}$. We can then bound that the probability that card $z_{\ell+1}$ *is not assigned* $c_t^{\ell+1}$ in any of the two processes, over any of the $b - a + 1$ steps (for $a \geq n$) to be less than or equal to $2\ell \cdot (b - a + 1) \cdot 2^{-n}$ (using union bound).

By definition of our Markov processes, once $z_{\ell+1}$ is the same position in both processes, it will remain in the same position in both processes from then onwards. Thus, we can bound:

$$\mathbf{P}(T_{\ell+1} > 2n) \leq \mathbf{P}(A_{n:2n-1}) \leq 2\ell n \cdot 2^{-n}. \tag{8}$$

Recall that $T_{\ell+1} = \min\{t : X_t = U_t\}$.[13]

Next, we will try to amplify this bound by considering what happens after more rounds. To do this, we need some more inequalities. Consider the probability of the cards z_i and z_j being matched for some timestep $t > kn$ given that event $A_{(k-1)n:kn-1}$ has taken place for any $k \geq 2$.

Lemma 7.3. *For any $k \geq 2$, $t > kn - 1$:*

$$\mathbf{P}\left(z_i \text{ and } z_j \text{ are matched at time } t \mid A_{(k-1)n:kn-1}\right) \leq 2^{1-n}.$$

Proof. By definition, z_i and z_j are matched if and only if the last $n - 1$ coins associated with their positions is the same.

For each timestep $t > kn - 1$, conditioned on $A_{(k-1)n:kn-1}$, recall that the coins associated with z_i and z_j's position are $(d_{t-1}^i, \ldots, d_{t-n}^i)$ and $(d_{t-1}^j, \ldots, d_{t-n}^j)$, respectively. We will then bound the probability of these sets of coins being equal using two cases.

---

[13]Up to now, our result and argument is very similar to before, with an improvement of a factor of two w.r.t. [52].

Case 1: z_i and z_j are matched for some time step $m \in \{t-1, \ldots, t-n+1\}$. First, notice that if for any $m \in \{t-1, \ldots, t-n+1\}$, z_i and z_j are matched, then by definition, $d_m^i \neq d_m^j$ and therefore the probability of $(d_{t-1}^i, \ldots, d_{t-n}^i)$ being equal to $(d_{t-1}^j, \ldots, d_{t-n}^j)$ is 0 conditioned on z_i being matched to z_j at some $m \in \{t-1, \ldots, t-n+1\}$.

Case 2: z_i and z_j are not matched for any time step $m \in \{t-1, \ldots, t-n+1\}$. Let us bound the probability of $d_m^i = d_m^j$ for each step $m \in \{t-1, \ldots, t-n+1\}$ individually. We will show that for every $m \in \{t-1, \ldots, t-n+1\}$, d_m^i and d_m^j are defined by *independent* samples from $\mathsf{Bern}(1/2)$.

W.l.o.g. consider z_i. In each step, z_i is either matched to no cards with index smaller than it, in which case its coin is sampled from $\mathsf{Bern}(1/2)$ or it is matched to some card of smaller index than it, in which case it gets assigned the negation of that card's coin, which is a sample from $\mathsf{Bern}(1/2)$.

The same is true for z_j, but notice that (1) we are restricting to the case that z_i and z_j are not matched, and (2) even if z_i and z_j are both matched to cards of smaller indices, they cannot be matched to the same card.[14] Then, it follows that in this case, d_m^i and d_m^j are always defined by distinct samples of $\mathsf{Bern}(1/2)$ and therefore the probability that they are equal is $1/2$.

Let B be the event that at time t, z_i and z_j were matched in any of the past $n-1$ steps. We have shown that:

$$\mathbf{P}\left(z_i \text{ and } z_j \text{ are matched at time } t \mid A_{(k-1)n:kn-1}, \neg B\right) = 2^{1-n}.$$

And we have also shown that:

$$\mathbf{P}\left(z_i \text{ and } z_j \text{ are matched at time } t \mid A_{(k-1)n:kn-1}, B\right) = 0.$$

These two equations suffice to prove our lemma. □

Now, given Lemma 7.3, we can show the following for any $t \geq kn$, for any $j \in \{1, \ldots q\}$:

$$\mathbf{P}\left(d_j^t \neq c_j^t \mid A_{(k-1)n:kn-1}\right) \leq (j-1)2^{-n} \tag{9}$$

This holds by the same argument as in Equation (7), given Lemma 7.3.

Finally, by the same arguments as previously, we can use a union bound to get that for any $k \geq 2$:,

$$\mathbf{P}\left(A_{kn:(k+1)n-1} \mid A_{(k-1)n:kn-1}\right) \leq 2\ell n\left(2^{-n}\right) = p. \tag{10}$$

Since two coins' adjacency is defined by its coins at the past $n-1$ timesteps, it is also straightforward that any event that happened further than n steps ago does not affect the probability of adjacency between two cards. In other words,

$$\mathbf{P}\left(A_{kn:(k+1)n-1} \mid A_{(k-1)n:kn-1}, A_{(k-2)n:(k-1)n-1}\right) \tag{11}$$

$$= \mathbf{P}\left(A_{kn:(k+1)n-1} \mid A_{(k-1)n:kn-1}\right) \leq p. \tag{12}$$

---

[14]This follows from the pigeonhole principle. If three cards share the same last $n-1$ bits, then at least two of them must have shared those $n-1$ bits in the previous round, and therefore have a distinct last bit. A contradiction.

Applying this more generally, we have that, for any $k > 0$:

$$\mathbf{P}(A_{n:2n-1} \wedge A_{2n:3n-1} \wedge \ldots \wedge A_{kn:(k+1)n-1})$$
$$= \mathbf{P}(A_{n:2n-1}) \cdot \mathbf{P}(A_{2n:3n-1} \mid A_{n:2n-1}) \cdot \ldots \cdot \mathbf{P}(A_{kn:(k+1)n-1} \mid A_{n:2n-1}, \ldots, A_{(k-1)n:kn-1})$$
$$= \mathbf{P}(A_{n:2n-1}) \cdot \mathbf{P}(A_{2n:3n-1} \mid A_{n:2n-1}) \cdot \ldots \cdot \mathbf{P}(A_{kn:(k+1)n-1} \mid A_{(k-1)n:kn-1}) \leq p^k$$

We can go from the second line to the third line by Equation (11). Then, the inequality on the third line follows by Equation (10). Looking back to our original goal of bounding $P(T_\ell > t)$, notice that unless A happens in every n interval, the two processes will be coupled. Therefore, we can now say that for any $r > 0$, for any $\ell \in \{0, \ldots, q-1\}$, $P(T_\ell > (r+1)n) \leq p^r$.

Finally, we put it all together to get:

$$\Delta\left(\tau_{t=(r+1)n}, \pi\right) \leq \sum_{\ell=0}^{q-1} \mathbf{P}\left(T_{\ell+1} > (r+1)n\right) \leq \sum_{\ell=0}^{q-1} \left(\frac{2\ell n}{N}\right)^r$$
$$\leq \frac{q}{r+1}\left(\frac{2qn}{N}\right)^r$$
$$= \frac{q(n+t+1)}{n+1}\left(\frac{2qn}{N}\right)^{t/(n+1)}.$$

The first line holds by a union bound, the second line holds because we take the integral (we define $t = (r+1)n$ to facilitate integration) and then just plug in. By [52], the total variational distance between two distributions is exactly equal to the CPA advantage of any stateful non-adaptive adversary distinguishing between them.

Furthermore, given a bound for non adaptive q-query adversaries in the CPA experiment, the result by Maurer et al. [48] tells us that we can enhance the secure to hold for any *adaptive, CCA* adversary with few additional rounds. Specifically, applying [48] to our work gives us that for any adaptive q-query adversary, the probability that it can distinguish a Thorp shuffle over $N = 2^n$ cards with t rounds from a true random permutation in a CCA experiment is less than or equal to

$$\frac{q(2n+t+2)}{n+1}\left(\frac{2qn}{N}\right)^{t/(2(n+1))}.$$

$\square$

8 Related Work

8.1 Why not use sorting networks?

A simple solution to performing a shuffle given a generic FHE scheme and a generic PRG $g : \{0,1\}^\lambda \rightarrow \{0,1\}^*$ is for the client to generate a PRG seed s, encrypt it under FHE, and send it to the server. It also sends its public key. The server then homomorphically evaluates $g(s)$ and assigns a random value (encrypted under FHE) to each database entry. It then uses a sorting network to sort the database elements according to the random values, where each gate in the network is encrypted under FHE. We already went into the difficulty of how to evaluate a PRG under FHE (and our proposed solution). However, it is still unclear why we go through the trouble of using a q-Thorp Shuffle when there exists low-depth sorting networks that can be used to solve the problem. The sorting network solution poses two additional *major* drawbacks.

1. Although there exist known sorting networks with optimal $O(N \log N)$ size and $O(\log N)$ depth [1, 59, 19], these sorting networks are notoriously complex and incur huge constant factors in the depth (the best known sorting network with $O(\log N)$ depth has depth about $1800 \log N$). Sorting networks with $O(N \log^2 N)$ gates and $O(\log^2 N)$ depth [7, 57, 64] perform better empirically, but still are asymptotically suboptimal, and furthermore suffer from the drawback below.

2. The ciphertexts randomly assigned by the server to each block must encrypt plaintext values of at least λ bits to ensure the sorting network generates a computationally random permutation (to avoid being assigned to the same random element except with 2^λ probability). This imposes an additional λ multiplicative factor to the depth of the FHE computation of a permutation on any sorting network, since comparing these two elements homomorphically requires a circuit of depth and size $O(\lambda)$. Thus combining with item (1), the total depth is $\Omega(\lambda \log(N))$. [15]

8.2 Oblivious Shuffles.

Oblivious shuffles have been studied under different definitions and contexts. Informally, we define a shuffle to be *oblivious* if the access pattern of the shuffling does not reveal any information about the shuffle itself. This definition is different than what is seen in [34, 51], whose schemes does not satisfy the definition above. Oblivious permutations (as defined above) have been studied to a large amount in the client server model, where the client wants to shuffle some N encrypted blocks stored at the server according to some pseudorandom permutation σ which should be hidden from the server. Works in this model [54, 58, 4] achieve near-optimal asymptotics with very little client storage usage, however, all known works require the client and server to communicate $O(N)$ information to perform these shuffles. In our work, we want to avoid the $O(N)$ communication, although for scenarios with very large database elements, it might be worth looking into performing an oblivious permutation with linear bandwidth that is perhaps independent of element size.

8.3 Permutation Networks.

Permutation networks are networks supposed to shuffle its inputs. There are two best known networks: Butterfly networks (which can equivalently be seen as the Thorp Shuffle [65] or a maximally unbalanced Feistel network), and Benes networks [9]. Known mixing times for the Thorp Shuffle are tighter than those for Benes networks [29], so we focus on the Thorp Shuffle in this work. Another permutation network, proposed by Waksman [66], achieves optimal size and depth, but it requires random access on a permutation matrix of size $N \times N$ which is costly when trying to evaluate the circuit under FHE.

---

[15]Notice that we cannot simply assign random output to each comparison in the sorting network. The output of the sorting network for this case would not necessarily be equivalent to sampling a permutation uniformly (although it certainly is true that every possible permutation can be output in this method, the distribution of each is not uniform). It is an open question whether there exists a low-depth sorting network that can be used with random comparators to output something indistinguishable from a uniformly sampled permutation.

8.4 Private Information Retrieval.

PIR has been studied in a series of different contexts and models. Other than the client preprocessing model studied in this work, another popular model is the server preprocessing model. In the server preprocessing model, the server runs one very expensive preprocessing and stores additional bits at the server which it can then use to respond to queries by *any* client in sublinear time. This has been studied in a series of works using non-standard assumptions or trusted setups [35, 12]. Recently Lin et al. [44] pushed a breakthrough on the server preprocessing model, achieving near-optimality under standard assumptions. The concrete costs in preprocessing time and server storage are still prohibitive [55], however it is still very recent work and there are likely many more practical improvements to be made. PIR has also been extensively studied with the simplifying assumption of requiring two non-colluding servers. Using the scheme by [31, 11, 32] we can run extremely efficient non-preprocessing PIR at rates very close to optimal, with the caveat of the non-colluding servers assumption.

Other works have studied single server PIR in the context of preprocessing with a smaller client hint [33, 25]. These works still incur linear server time during the query phase, however, all expensive operations are performed offline and the online phase can be viewed as a single scan through the database. Another emerging interesting line of work within PIR is the line of work on Authenticated PIR [20, 26, 67]. In this line of work, the client can be guaranteed that the server followed the protocol honestly, a guarantee not inherent in standard PIR.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 1–9, New York, NY, USA, Dec. 1983. Association for Computing Machinery.

[2] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, pages 169–203, 2015.

[3] J. Alperin-Sheriff and C. Peikert. Practical bootstrapping in quasilinear time. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 1–20, Santa Barbara, CA, USA, Aug. 18–22, 2013. Springer, Heidelberg, Germany.

[4] G. Asharov, T.-H. H. Chan, K. Nayak, R. Pass, L. Ren, and E. Shi. Bucket Oblivious Sort: An Extremely Simple Oblivious Sort. In *Symposium on Simplicity in Algorithms (SOSA)*, Proceedings, pages 8–14. Society for Industrial and Applied Mathematics, Dec. 2019.

[5] S. Bai, K. Boudgoust, D. Das, A. Roux-Langlois, W. Wen, and Z. Zhang. Middle-product learning with rounding problem and its applications. In S. D. Galbraith and S. Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019*, pages 55–81, Cham, 2019. Springer International Publishing.

[6] A. Banerjee, C. Peikert, and A. Rosen. Pseudorandom functions and lattices. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 719–737, Cambridge, UK, Apr. 15–19, 2012. Springer, Heidelberg, Germany.

[7] K. E. Batcher. Sorting networks and their applications. *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*, page 307, 1968. Conference Name: the April 30–May 2, 1968, spring joint computer conference Place: Atlantic City, New Jersey Publisher: ACM Press.

[8] A. Beimel, Y. Ishai, and T. Malkin. Reducing the Servers Computation in Private Information Retrieval: PIR with Preprocessing. In M. Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, Lecture Notes in Computer Science, pages 55–73, Berlin, Heidelberg, 2000. Springer.

[9] V. E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, 1965. Google-Books-ID: nQcjAAAAMAAJ.

[10] D. Boneh, S. Kim, and H. Montgomery. Private Puncturable PRFs from Standard Lattice Assumptions. In J.-S. Coron and J. B. Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, Lecture Notes in Computer Science, pages 415–445, Cham, 2017. Springer International Publishing.

[11] E. Boyle, N. Gilboa, and Y. Ishai. Function Secret Sharing: Improvements and Extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1292–1303, New York, NY, USA, Oct. 2016. Association for Computing Machinery.

[12] E. Boyle, Y. Ishai, R. Pass, and M. Wootters. Can We Access a Database Both Locally and Privately? pages 662–693, Nov. 2017.

[13] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO 2012*, LNCS. Springer, Aug. 19–23, 2012.

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS 2012*. ACM, Jan. 8–10, 2012.

[15] Z. Brakerski and V. Vaikuntanathan. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In P. Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, Lecture Notes in Computer Science, pages 505–524, Berlin, Heidelberg, 2011. Springer.

[16] R. Canetti and Y. Chen. Constraint-Hiding Constrained PRFs for NC^{1}from LWE. In J.-S. Coron and J. B. Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, Lecture Notes in Computer Science, pages 446–476, Cham, 2017. Springer International Publishing.

[17] H. Chen, Z. Huang, K. Laine, and P. Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 1223–1237, Toronto, ON, Canada, Oct. 15–19, 2018. ACM Press.

[18] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. pages 41–41. IEEE Computer Society, Oct. 1995. ISSN: 0272-5428.

[19] V. Chvátal. ON THE NEW AKS SORTING NETWORK. 1992.

[20] S. Colombo, K. Nikitin, H. Corrigan-Gibbs, D. J. Wu, and B. Ford. Authenticated private information retrieval. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, pages 3835–3851, USA, Aug. 2023. USENIX Association.

[21] K. Cong, R. C. Moreno, M. B. da Gama, W. Dai, I. Iliashenko, K. Laine, and M. Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 1135–1150, Virtual Event, Republic of Korea, Nov. 15–19, 2021. ACM Press.

[22] H. Corrigan-Gibbs, A. Henzinger, and D. Kogan. Single-Server Private Information Retrieval with Sublinear Amortized Time. In *Advances in Cryptology – EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 – June 3, 2022, Proceedings, Part II*, pages 3–33, Berlin, Heidelberg, May 2022. Springer-Verlag.

[23] H. Corrigan-Gibbs and D. Kogan. Private Information Retrieval with Sublinear Online Time. In A. Canteaut and Y. Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, Lecture Notes in Computer Science, pages 44–75, Cham, 2020. Springer International Publishing.

[24] A. Czumaj and B. Vöcking. Thorp Shuffling, Butterflies, and Non-Markovian Couplings. In *Automata, Languages, and Programming*, Lecture Notes in Computer Science, pages 344–355, Berlin, Heidelberg, 2014. Springer.

[25] A. Davidson, G. Pestana, and S. Celi. FrodoPIR: Simple, Scalable, Single-Server Private Information Retrieval. *Proceedings on Privacy Enhancing Technologies*, 2023(1), 2023.

[26] M. Dietz and S. Tessaro. Fully Malicious Authenticated PIR, 2023. Publication info: Preprint.

[27] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://ia.cr/2012/144.

[28] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research, edited by R.A. Fisher and F. Yates. 6th ed.* Edinburgh: Oliver and Boyd, 1963. Accepted: 2006-06-27T07:57:52Z.

[29] E. Gelman and A. Ta-Shma. The Benes Network is q*(q-1)/2n-Almost q-set-wise Independent. In *DROPS-IDN/v2/document/10.4230/LIPIcs.FSTTCS.2014.327*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2014.

[30] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing*, STOC '09, page 169–178. ACM, 2009.

[31] N. Gilboa and Y. Ishai. Distributed Point Functions and Their Applications. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, Lecture Notes in Computer Science, pages 640–658, Berlin, Heidelberg, 2014. Springer.

[32] S. M. Hafiz and R. Henry. A Bit More Than a Bit Is More Than a Bit Better: Faster (essentially) optimal-rate many-server PIR. *Proceedings on Privacy Enhancing Technologies*, 2019(4):112–131, Oct. 2019.

[33] A. Henzinger, M. M. Hong, H. Corrigan-Gibbs, S. Meiklejohn, and V. Vaikuntanathan. One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval. page 27, 2022.

[34] V. T. Hoang, B. Morris, and P. Rogaway. An Enciphering Scheme Based on a Card Shuffle. Technical Report arXiv:1208.1176, arXiv, Nov. 2014. arXiv:1208.1176 [cs] type: article.

[35] J. Holmgren, R. Canetti, and S. Richelson. Towards Doubly Efficient Private Information Retrieval. Technical Report 568, 2017.

[36] D. Kogan and H. Corrigan-Gibbs. Private Blocklist Lookups with Checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892. USENIX Association, 2021.

[37] V. Kolesnikov, J. B. Nielsen, M. Rosulek, N. Trieu, and R. Trifiletti. DUPLO: Unifying cut-and-choose for garbled circuits. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 3–20, Dallas, TX, USA, Oct. 31 – Nov. 2, 2017. ACM Press.

[38] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 364–373, Miami Beach, FL, USA, 1997. IEEE Comput. Soc.

[39] A. Lazzaretti, Z. Liu, B. Fisch, and C. Papamanthou. Single Server PIR via Homomorphic Thorp Shuffles. Preprint (under submission), 2024.

[40] A. Lazzaretti and C. Papamanthou. Near-Optimal Private Information Retrieval with Preprocessing. In G. Rothblum and H. Wee, editors, *Theory of Cryptography*, Lecture Notes in Computer Science, pages 406–435, Cham, 2023. Springer Nature Switzerland.

[41] A. Lazzaretti and C. Papamanthou. TreePIR: Sublinear-Time and Polylog-Bandwidth Private Information Retrieval from DDH. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part II*, pages 284–314, Berlin, Heidelberg, Aug. 2023. Springer-Verlag.

[42] A. Lazzaretti and C. Papamanthou. Single Pass Client Preprocessing Private Information Retrieval. In *(to appear) USENIX Security 2024*, 2024.

[43] D. A. Levin, Y. Peres, and E. L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Soc., 2009. Google-Books-ID: 6Cg5Nq5sSv4C.

[44] W.-K. Lin, E. Mook, and D. Wichs. Doubly Efficient Private Information Retrieval and Fully Homomorphic RAM Computation from Ring LWE, 2022. Report Number: 1703.

[45] Z. Liu and Y. Wang. Amortized functional bootstrapping in less than 7 ms, with $\tilde{O}(1)$ polynomial multiplications. LNCS, pages 101–132. Springer, Heidelberg, Germany, Dec. 7–11, 2023.

[46] Z. Liu and Y. Wang. Relaxed functional bootstrapping: A new perspective on bgv/bfv bootstrapping. Cryptology ePrint Archive, Paper 2024/172, 2024. `https://eprint.iacr.org/2024/172`.

[47] Y. Ma, Z. Ke, T. Rabin, and S. Angel. Incremental Offline/Online PIR (extended version). In *USENIX Security 2022*, 2022.

[48] U. Maurer, K. Pietrzak, and R. Renner. Indistinguishability Amplification. In A. Menezes, editor, *Advances in Cryptology - CRYPTO 2007*, pages 130–149, Berlin, Heidelberg, 2007. Springer.

[49] Microsoft SEAL (release 3.6). `https://github.com/Microsoft/SEAL`, Nov. 2020. Microsoft Research, Redmond, WA.

[50] B. Morris. Improved mixing time bounds for the Thorp shuffle and L-reversal chain. *The Annals of Probability*, 37(2):453–477, Mar. 2009. Publisher: Institute of Mathematical Statistics.

[51] B. Morris and P. Rogaway. Sometimes-Recurse Shuffle. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, Lecture Notes in Computer Science, pages 311–326, Berlin, Heidelberg, 2014. Springer.

[52] B. Morris, P. Rogaway, and T. Stegers. How to Encipher Messages on a Small Domain. In *Advances in Cryptology - CRYPTO 2009*, volume 5677, pages 286–302. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[53] M. H. Mughees, S. I, and L. Ren. Simple and Practical Amortized Sublinear Private Information Retrieval, 2023. Publication info: Preprint.

[54] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal. The Melbourne Shuffle: Improving Oblivious Storage in the Cloud. In *Automata, Languages, and Programming*, Lecture Notes in Computer Science, pages 556–567, Berlin, Heidelberg, 2014. Springer.

[55] H. Okada, R. Player, S. Pohmann, and C. Weinert. Towards Practical Doubly-Efficient Private Information Retrieval, 2023. Publication info: Published elsewhere. Minor revision. Financial Cryptography and Data Security 2024.

[56] A. S. Özcan, C. Ayduman, E. R. Turkoglu, and E. Savas. Homomorphic encryption on gpu. *IEEE Access*, 11:84168–84186, 2023.

[57] I. Parberry. The Pairwise Sorting Network. *Parallel Processing Letters*, 2:205–211, Sept. 1992.

[58] S. Patel, G. Persiano, and K. Yeo. CacheShuffle: An Oblivious Shuffle Algorithm Using Caches. *ArXiv*, May 2017.

[59] M. S. Paterson. Improved sorting networks withO(logN) depth. *Algorithmica*, 5(1):75–92, June 1990.

[60] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai. SpOT-light: Lightweight private set intersection from sparse OT extension. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 401–431, Santa Barbara, CA, USA, Aug. 18–22, 2019. Springer, Heidelberg, Germany.

[61] R. Podschwadt, D. Takabi, P. Hu, M. H. Rafiei, and Z. Cai. A survey of deep learning architectures for privacy-preserving machine learning with fully homomorphic encryption. *IEEE Access*, 10:117477–117500, 2022.

[62] W. Raji. *An Introductory Course in Elementary Number Theory.* July 2013.

[63] R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, pages 169–179, 1978.

[64] F. Shi, Z. Yan, and M. Wagh. An enhanced multiway sorting network based on n-sorters. *2014 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 60–64, Dec. 2014. Conference Name: 2014 IEEE Global Conference on Signal and Information Processing (GlobalSIP) ISBN: 9781479970889 Place: Atlanta, GA, USA Publisher: IEEE.

[65] E. Thorp. Nonrandom Shuffling with Applications to the Game of Faro. *Journal of The American Statistical Association - J AMER STATIST ASSN*, 68:842–847, Dec. 1973.

[66] A. Waksman. A Permutation Network. *Journal of the ACM*, 15(1):159–163, Jan. 1968.

[67] Y. Wang, J. Zhang, J. Liu, and X. Yang. Crust: Verifiable And Efficient Private Information Retrieval with Sublinear Online Time, 2023. Publication info: Preprint.

[68] B. Wei, R. Wang, Z. Li, Q. Liu, and X. Lu. Fregata: Faster homomorphic evaluation of aes via tfhe. In E. Athanasopoulos and B. Mennink, editors, *Information Security*, pages 392–412, Cham, 2023. Springer Nature Switzerland.

[69] M. Zhou, W.-K. Lin, Y. Tselekounis, and E. Shi. Optimal Single-Server Private Information Retrieval. *ePrint IACR*, 2022.

[70] M. Zhou, A. Park, E. Shi, and W. Zheng. Piano: Extremely Simple, Single-Server PIR with Sublinear Server Computation, 2023. Publication info: Published elsewhere. Major revision. IEEE S&P 2024.

A Two-Server PIR by [42]

As recalled in Section 2.6, our scheme is inspired by the two server PIR scheme in [42]. Here we recall the entire pseudocode from [42]. We provide the scheme in Algorithm 8.

Algorithm 8 The two-server PIR scheme from [42]. Let $Q, N \in \mathbb{N}$ such that $Q|N$. Let $m \in \mathbb{N} = N/Q$. Let DB be an array of N elements of size w. For $i \in [Q]$, let $\mathsf{DB}_i = \mathsf{DB}[i * m : (i+1)m]$.

1: **procedure** PREPROCESS(DB)   ▷ Note that λ, T are not necessary
2:    Sample $(P_1, \ldots, P_Q)$, permutations of N/Q elements, uniformly at random from the set of all permutations.
3:    Let $h_1, \ldots, h_m = 0$.
4:    **for** j in $[1, \ldots, m]$ **do**
5:        Let $h_j = \bigoplus_{i \in Q} P_i(j)$.
6:    **return** $(P_1, \ldots, P_Q), (h_1, \ldots, h_m)$.

7: **procedure** QUERY($\mathsf{st} = (P_1, \ldots, P_Q), x = (i^*, j^*) \in ([Q] \times [m])$)   ▷ Note that sf also includes the hints, which are not used here and thus omitted
8:    Find ind such that $P_{i^*}(ind) = j^*$.
9:    Let $S_1 = [P_j(ind) : j \in [Q]]$. Let $S[i^*] = r^* \xleftarrow{\$} [m]$.
10:    Sample $r_1, \ldots, r_Q \xleftarrow{\$} [m]^Q$.
11:    Let $S_0 = [P_i(r_i) : i \in [Q]]$.
12:    For $i \in [Q], i \neq i^*$, swap $P_i(ind)$ and $P_i(r_i)$.
13:    **return** Output $\mathsf{rq} = (q_0 = S_0, q_1 = S_1)$.

14: **procedure** ANSWER(DB, $\mathsf{rq} = (q_b = (a_1, \ldots, a_Q))_{b \in \{0,1\}})$
15:    **return** $a = (A_b = [\mathsf{DB}_i(a_i) : i \in [Q]])_{b \in \{0,1\}}$.

16: **procedure** RECONSTRUCT($\mathsf{st} = \{h_j\}_{j \in [m]}, x = (i^*, j^*), a = (A_0, A_1)$)   ▷ Note that sf also includes the permutations, which are not used here and thus omitted
17:    Let $\mathsf{DB}[x] = \mathsf{DB}_{i^*}[j^*] = h_{j^*} \oplus \left(\bigoplus_{i \in [Q], i \neq i^*} A_1[i] \right)$.
18:    **for** i in $[1, \ldots, Q]$, $i \neq i^*$ **do**
19:        Update $h_{ind} = h_{ind} \oplus A_1[i] \oplus A_0[i]$.
20:        Update $h_{r_i} = h_{r_i} \oplus A_1[i] \oplus A_0[i]$.
21:    **return** $\mathsf{DB}[x]$.