Batched Differentially Private Information Retrieval

Kinan Dak Albab[*,†] Rawane Issa[*,‡] Mayank Varia[‡] Kalman Graffi[§]

Abstract

Private Information Retrieval (PIR) allows several clients to query a database held by one or more servers, such that the contents of their queries remain private. Prior PIR schemes have achieved sublinear communication and computation by leveraging computational assumptions, federating trust among many servers, relaxing security to permit differentially private leakage, refactoring effort into a pre-processing stage to reduce online costs, or amortizing costs over a large batch of queries.

In this work, we present an efficient PIR protocol that combines all of the above techniques to achieve *constant* amortized communication and computation complexity in the size of the database and constant client work. We leverage differential private leakage in order to provide better trade-offs between privacy and efficiency. Our protocol achieves speed-ups up to and exceeding 10x in practical settings compared to state of the art PIR protocols, and can scale to 10^5 queries per second on cheap commodity AWS machines. Our protocol builds upon a new secret sharing scheme that is both incremental and non-malleable, which may be of interest to a wider audience. Our protocol provides security up to abort against malicious adversaries that can corrupt all but one party.

1 Introduction

Private Information Retrieval (PIR) [26, 45] is a cryptographic primitive that allows a client to retrieve a record from a public database held by a single or multiple servers without revealing the content of her query. PIR protocols have been developed for a variety of settings, including information theoretic PIR where the database is replicated across several servers [26], and computational PIR using single server [45]. The different settings of PIR are limited by various lower bounds on their computation or communication complexity. In essence, a server must "touch" every entry in the database when responding to a query, otherwise the server learns information about the query, namely what the query is not!

Recent PIR protocols [27, 43, 51, 59] achieve sub-linear computation and communication by relying on a preprocessing/offline stage that shifts the bulk of computation into off-peak hours [12], relaxing the security to allow limited leakage [61], or batching queries that originate from the **same** client. These advances allowed PIR to be used in a variety of privacy preserving applications including private presence discovery [16,58], anonymous communication and messaging [9,23,46,56], private media and advertisement consumption [36, 37], certificate transparency [51], and privacy preserving route recommendation [69], among others.

[*]These authors contributed equally to this effort.
[†]Brown University. Email: kinan_dak_albab@brown.edu
[‡]Boston University. Email: {ra1issa,varia}@bu.edu
[§]Honda Research Institute EU. Email: kalman.graffi@honda-ri.de

Existing sublinear PIR protocols are able to handle medium to large databases of size n and still respond to queries reasonably quickly. However, they scale poorly as the number of queries increase: the sub-linear cost (e.g. $\sqrt{n}$ for Checklist [43]) of handling each query quickly adds up when the number of queries approach or exceeds the size of the database into a super-linear overall cost (e.g. $n\sqrt{n}$). Efficiently batching such queries and amortizing their overheads is an open problem when these queries are made by **different** clients: existing work that batches such queries require an impractical number of queries for the amortization to become significant [51] at the cost of sub-linear communication, or requires clients to coordinate or share secrets when preprocessing is used [12]. This complicates efforts to deploy PIR in a variety of important applications including software updates, contact tracing, content moderation, blacklisting of fake news, software vulnerability look-up, and similar large-scale automated services. We demonstrate this empirically in section 2.

In this work, we introduce DP-PIR, a novel differentially private PIR protocol tuned to efficiently handle large batches of queries approaching or exceeding the size of the underlying database. Our protocol batches queries from different non-coordinating clients. DP-PIR is the first protocol to achieve constant amortized server computation and communication, as well as constant client computation and communication.

While the details of our protocol are different from earlier work, at a high level our construction combines three ideas:

1. Offloading public key operations to an offline stage so that the online stage consists only of cheap operations [27, 59].

2. High throughput batched shuffling of messages by mix-nets and secure messaging systems [47, 48, 64, 66].

3. Relaxing the security of oblivious data structures and protocols to differentially private leakage [54].

DP-PIR Overview Our protocol is a batched information-theoretic multi-server PIR protocol optimized for queries approaching or exceeding the database size. DP-PIR is secure up to selective aborts against a dishonest majority of *malicious* servers, as long as at least one server is honest. Our protocol induces a per-batch overhead linear in the size of the database, this overhead is independent of the number of queries q in that batch, with a total computation complexity of $n + q$ per entire batch. When the number of queries approaches or exceeds the size of the database, the amortized computation complexity per query is constant. Furthermore, our protocol only requires constant computation, communication, and storage on the client side, regardless of amortization. We describe the details of our construction in section 5.

Our protocol achieves this by relaxing the security guarantees of PIR to differential privacy (DP) [30]. Unlike traditional PIR protocols, servers in DP-PIR learn a noised differentially private histogram of the queries made in a batch. Clients secret share their queries and communicate them to the servers, which are organized in a chain similar to a mixnet. Our servers take turns shuffling these queries and injecting them with generated noise queries similar to Vuvuzela [66]. The last server reconstructs the queries (both real and noise) revealing a noisy histogram, and looks them up in the database. The servers similarly secret share and de-shuffle responses, while removing responses corresponding to their noise, and then send them to their respective clients for final reconstruction. The noise queries are generated from a particular distribution to ensure that the revealed histogram is (ϵ, δ)-differentially private, so that the smaller ϵ and δ get, the more noise

queries need to be added. The number of these queries is linear in n and $\frac{1}{\epsilon}$ and independent of the number of queries in a batch. The noise does not affect the accuracy or correctness of any client's output. Section 3 describes our threat model and how our differentially private guarantees can be interpreted and compared to those of traditional PIR protocols.

Our protocol offloads all expensive public key operations to a similarly amortizable offline preprocessing stage. This stage produces correlated secret material that our protocol then uses online. Our online stage uses only a cheap information-theoretic secret sharing scheme, consisting solely of a few field operations, which modern CPUs can execute in a handful of cycles. The security of our protocol requires this secret sharing scheme, which we define in section 4, be both incremental and non-malleable. Finally, section 6 describes how our protocol can be parallelized over additional machines to exhibit linear improvements in latency and throughput.

Our Contribution We make three main contributions:

1. We introduce the first PIR protocol that achieves constant amortized server complexity with constant client computation and communication, including both its offline and online stage, when the number of queries is similar or larger than the size of the database, even when the queries are made by different clients. Our offline stage performs public key operations linear in the database and queries size, and the online stage consists exclusively of cheap arithmetic operations.

2. We achieve a crypto-free online stage via a novel secret sharing scheme that is both incremental and non-malleable, based only on modular arithmetic for both sharing and reconstruction. To our knowledge, this is the first information theoretic scheme that exhibits both properties combined. This scheme may be of independent interest in scenarios involving Mixnets, (Distributed) ORAMs, and other shuffling and oblivious data structures.

3. We implement this protocol and demonstrate its performance and scaling to loads with several million queries, while achieving an online latency of few seconds on cheap low-resource cloud environments. The experiments identify a criterion describing application settings where our protocol is most effective compared to existing protocols, based on the ratio of the number of queries over the database size.

2 Motivation

Private Information Retrieval is a powerful primitive that conceptually applies to a wide range of privacy preserving applications. Existing PIR protocols are well suited for applications with medium to large databases and small or infrequent number of queries [9, 36, 57, 69]. However, they are impractical for a large class of applications with a large number of queries, especially periodic automated services, and applications with heavy peak loads.

We consider one such example: checking for software updates on mobile app stores. As of 2020, the Google Play and iOS app stores contain an estimated 2.56 and 1.85 million applications each [2], while the number of active Android and iOS devices exceed 3 and 1.65 billion, respectively [3]. These devices perform periodic (e.g. daily) background checks to ensure that their installed applications are up to date. Currently, these checks are done without privacy: The relevant app store knows all applications installed on a device, and can perform checks to determine if they are up-to-date

quickly. However, the installed applications on one's device constitute sensitive information. They can reveal information about the user's activity (e.g. which bank they use), or whether the device has applications with known exploits.

It is desirable to hide the sensitive application information from the app store as well as potential attackers. A device can send a PIR query for each application installed, and the servers can privately respond with the most up-to-date version label of each application. The device can compare the received version label against the installed version. If the application is out of date, the device can construct a URL for downloading it, assuming the stores use a reasonable URL naming convention (e.g. store.com/app_name/version_label). Downloading the application must be done via some anonymous channel, such as Tor, to avoid revealing installed applications to the host.

Existing PIR protocols cannot practically realize this application. The number of devices is about 1000x larger than the size of the database, and each device may have tens to hundreds of applications installed. With such large loads, the sub-linear overhead per query quickly adds up. This is further exacerbated with time as more applications are added to the store. We demonstrate this below with experiments using two state of the art PIR protocols: Checklist [43] and SealPIR [8].

We believe that a large class of applications demonstrate similar properties ideal for DP-PIR. In privacy-preserving automated exposure notification for contact tracing [20, 62, 63], the number of recent cases in a city or region (i.e. the size of the database) is far smaller than the total population of that area (i.e. the number of queries). Similarly, blocking misinformation in end-to-end encrypted messaging systems [44] usually involves a denylist far smaller than the total number of messages exchanged in the system within a reasonable batching time window. Our empirical results demonstrate that DP-PIR achieves significant concrete speedups over state of the art protocols in applications where the ratio of queries to database size exceeds 10. On the other hand, our protocol is unsuited for applications with a ratio smaller than $\frac{1}{10}$.

We believe that the differential privacy guarantees of DP-PIR suffice for applications where the primary focus is protecting the privacy of any given client, but not overall trends or patterns. For many applications where PIR could be employed, it is also desirable for the (approximate) overall query distribution to be publicly revealed, e.g. an app store that displays download counts or a private exposure notification service that also identifies infection hotspots. DP-PIR is ideal for such applications, since it reveals a noised version of this distribution, without having to use an additional private heavy hitters protocol [14]. Furthermore, our DP guarantees are more meaningful in the settings that DP-PIR target, where the number of queries and clients is large, and any particular client's query is intermixed with a large set of indistinguishable noise queries as well as queries from other clients. In practice, we emphasize that our relaxed guarantees should be viewed as an improvement over the insecure status-quo, rather than a replacement for PIR protocols that have stronger guarantees but impractical overheads in our target settings.

Existing PIR Protocols Private Information Retrieval (PIR) has been been extensively studied in a variety of settings. Information theoretic PIR replicates the database over several non-colluding servers [11], while computational PIR traditionally uses a single database and relies on cryptographic hardness assumptions [18, 25, 49].

Naive PIR protocols require a linear amount of computation and communication (e.g. sending the entire database over to the client), and several settings have close-to-linear lower bounds on either computation or communication [50].

Modern PIR protocols commonly introduce an offline preprocessing stage, which either encodes

Protocol	Computation		Communication	
	Online	Offline	Online	Offline
BIM04 [12]	$n^{0.55}$	–	$n^{0.55}$	–
CK20 [27]	$\sqrt{n}$	n	$\lambda^2 \log n$	$\sqrt{n}$
Checklist [43]	$\sqrt{n}$	n	$\lambda \log n$	$\sqrt{n}$
Naive †	n	-	n/q^*	-
PSIR [59] †	$q^* n$	n	$\log^c n$	n/q
CK20 [27] †	$q^*\sqrt{n}$	n	$\sqrt{n}$	$\sqrt{n}/q^*$
BIM04 [12] ‡§	$qn^{\frac{w}{3}}$	–	$n^{\frac{1}{3}}/q$	–
LG15 [51] ‡¶	$q^{0.8}n$	–	$\sqrt{n}$	–
This work ‡‖	$c_{\epsilon,\delta}n + q$	$c_{\epsilon,\delta}n + q$	1	1

†: support batching of queries made by the **same** client.
‡: supports batching of queries made by **different** clients.
§: amortizes to $n^{\frac{w}{3}}$, $w \geq 2$ is the matrix mult. exponent.
¶: amortizes to a constant when $q \sim n^5$.
‖: amortizes to a constant when $q \sim n$.

Table 1: Computation and communication complexity of various existing PIR protocols. Here, n is the database size, q^* and q are the number of queries made by a single or different clients. For protocols that support batching, computation complexity represents the **total** complexity to handle a batch. Communication is always per query

the database for faster online processing using replication [12, 15, 27, 43] and coding theory [17, 19, 38, 59], performs a linear amount of offline work per client to make the online stage sub-linear [19, 27, 43, 43], or performs expensive public key operations so that the online stage only consists of cheaper ones [27, 43, 59]. Other PIR protocols improve performance by allowing some leakage [61], or relying on homomorphic primitives [6, 8, 68].

Finally, some protocols allow batching queries to amortize costs. When combined with preprocessing, batching is only supported for queries originating from the same client [27, 43, 59], or ones that share secret state [12]. Batching queries from different clients without preprocessing is possible [40] but has limitations. Earlier work induces a sublinear (but non constant) amortized computation complexity [12], while the state of the art [51] only amortizes to a constant when the number of queries approach n^5, at the cost of $\sqrt{n}$ client computation and communication. Our work is the first to amortize computation costs down to a constant quickly, while also requiring constant client computation and communication.

Experiment Setup Our experiments measure the server(s) time needed to process a complete set of queries with $\epsilon = 0.1$ and $\delta = 10^{-6}$. While the trends shown in these results are intrinsic properties of our protocol design, the exact numbers depend on the setup and protocol parameters. Section 7 discusses our setup and the effects of these parameters in detail.

Checklist Figure 1 shows the server computation time of Checklist and DP-PIR when processing different number of queries against a database with $n = 100K$ elements. Our protocol has constant performance initially, which starts to increase with the number of queries q as it exceeds 1M. In

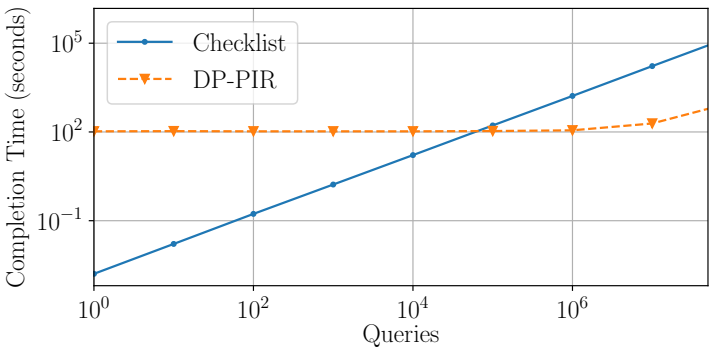

Figure 1: Checklist and DP-PIR Total completion time (y-axis, logscale) for varying number of queries (x-axis, logscale) against a 100K database

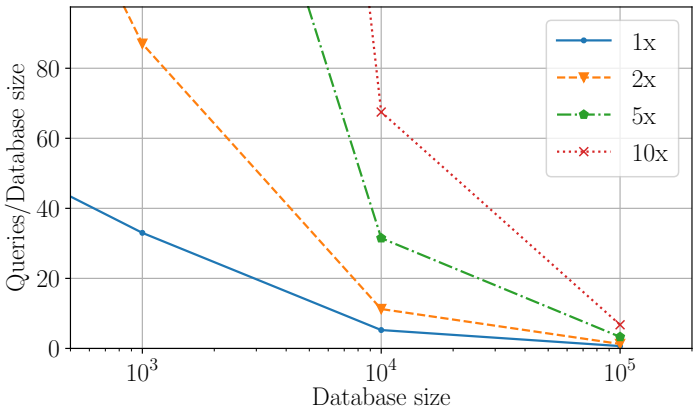

Figure 2: The ratios of queries/database (y-axis) after which DP-PIR outperforms Checklist for different database sizes (x-axis, logscale)

more detail, the computation time of DP-PIR is proportional to $c_{\epsilon,\delta}n + q = 184 \times 100,000 + q$. Therefore, the cost induced by q is negligible until q becomes relatively significant.

On the other hand, Checklist scales linearly with the number of queries throughout, as its computation time is proportional to $q\sqrt{n}$. When the number of queries is small, this cost is far smaller than the initial overhead of our system. As q approaches n, both systems start getting comparable performance. DP-PIR achieves identical performance to Checklist at $q = 67K$ (two thirds the database size) and outperforms Checklist for more queries. Our speedup over Checklist grows with the ratio $\frac{q}{n}$, approaching a maximum speedup determined by $\sqrt{n}$ when the ratio approaches ∞. Our experiments demonstrate that we outperform Checklist by at least 2x, 5x, and 10x after the ratio exceeds $1.27, 3.3$, and 6.7 respectively.

The ratio required for achieving a particular speedup is not identical for all database sizes. Figure 2 shows how these ratios decrease as the database size grows. DP-PIR prefers larger databases: The larger the database the smaller the ratio required by DP-PIR to achieve a particular speedup, and the larger the maximum speedup that DP-PIR can achieve as $q \to \infty$. In our motivating scenario involving the Google Play store where the ratio exceeds 1000, our protocol can achieve a

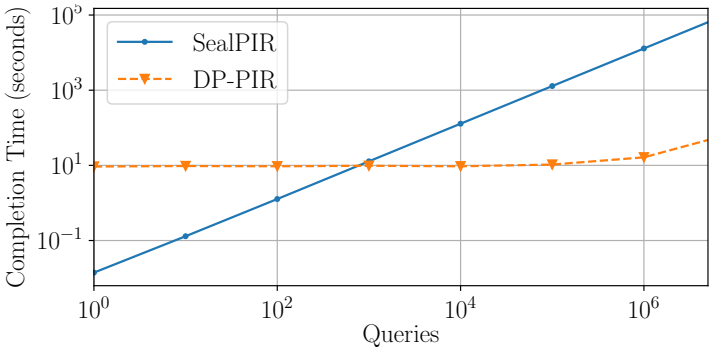

Figure 3: SealPIR and DP-PIR Total completion time (y-axis, logscale) for varying number of queries (x-axis, logscale) against a 10K database

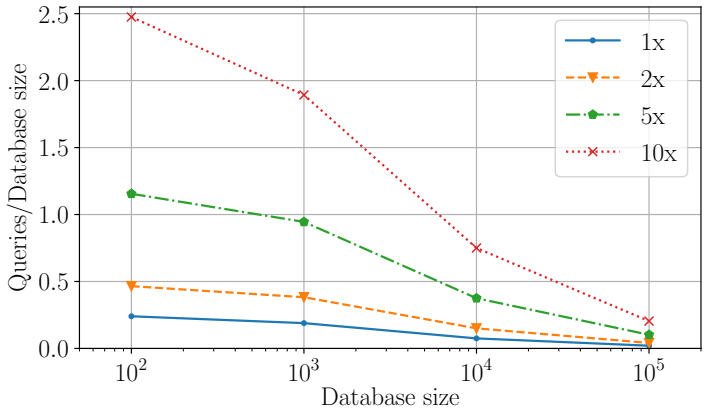

Figure 4: The ratios of queries/database (y-axis) after which DP-PIR outperforms SealPIR for different database sizes (x-axis, logscale)

speedup over Checklist far in excess of 10x.

SealPIR Figures 3 and 4 show similar results for SealPIR. In the first experiment, we use a database size of only 10K elements, and find that DP-PIR outperforms SealPIR at relatively few queries (around 600) with a ratio $\frac{q}{n}$ of just 0.06. Similarly, we achieve 2x, 5x, and 10x speedups for modest ratios of $\frac{15}{100}$, $\frac{3}{8}$, and $\frac{3}{4}$ respectively. These ratios decrease as the database size grows, similar to our experiment with Checklist.

While SealPIR has a sub-linear overhead per query similar to Checklist, we outperform SealPIR with far fewer queries than we do Checklist. In fact, our experiments show that Checklist is 20x faster than SealPIR. Checklist's online phase contains only symmetric key operations, since it offloads all expensive public key operations to an offline stage. SealPIR on the other hand does not rely on such an offline stage, and instead performs expensive homomorphic operations during its online stage. Our protocol goes even further, only executing a couple of modular arithmetic operations per query online.

3 Protocol Overview

Notation Our protocol consists of d clients and m servers labeled $c_1, ..., c_d$ and $s_1, ..., s_m$ respectively. We designate s_1 and s_m as a special *frontend* and *backend* server respectively. We assume that every server s_i has public encryption key pk_i known to all servers and clients, with associated secret key sk_i. Finally, we assume that every server has a copy of the underlying database, represented as a table $T = K : V$ consisting of a key and a value column.

We refer to the query made by client c_i by q^i, and its associated response as r^i. We use $(x_0^i, y_0^i), ..., (x_m^i, y_m^i)$ and $e_0^i, ..., e_m^i$ to denote the two sets of secret shares created by client i for her query and response. We denote the shared *anonymous secrets* installed from client c_i at server s_j by $a_j^i = (t_j^i, t_{j+1}^i, x_j^i, y_j^i, e_j^i)$. We drop the i superscript when the context is clear.

Our offline protocol uses *onion encryption* to pass secrets through the servers as defined below, where :: denotes string concatenation:

$$OEnc(i) = Enc(sk_1,\ a_1^i :: Enc(sk_2,\ a_2^i ::\ ...\ Enc(sk_m,\ a_m^i)\ ...\ ))$$

3.1 Setting

We describe our protocol in terms of a single epoch consisting of an input-independent offline stage followed by an online stage. Our protocol allows for executing a (slower) offline stage in one shot, which can then support several (faster) online stages executed afterward. Practical deployments may benefit from this feature to run the offline stage when computing power is cheaper, after which the clients have flexibility to choose when to make their queries. The converse is not true: secrets resulting from several offline stages cannot be used in a single online computation, or else an adversary could identify which offline stage each query belongs to.

The client state, created in the offline stage and consumed in the online one, consists exclusively of random elements. Clients can store the seed used to produce these elements to achieve constant storage relative to the number of queries and number of servers. A client need only submit her secrets to the service during the offline stage, and can immediately leave the protocol afterwards. The client can reconnect at any later time to make a query without any further coordination.

The offline stage is more computationally expensive than the online one, since it performs a linear number of public key operations overall. We suggest that the offline stage be carried out during off-peak hours (e.g. overnight), when utilization is low. Furthermore, both our stages are embarrassingly parallel in the resources of each party. It may be reasonable to run the offline stage with more resources, if these resources are cheaper to acquire overnight (e.g. spot instances).

While the requirements on the offline stage are primarily throughput oriented, the online stage must exhibit reasonable latency as well. Our offline stage is similar to Vuvuzela [66], which exhibits good throughput. The linear number of public key operations performed by Vuvuzela makes it impractical for our online stage. Indeed, our online stage is crypto-free using only a handful of arithmetic operations per query.

3.2 Threat Model

Our construction operates in the 'anytrust' model up to *selective* abort. Specifically, we tolerate up to $m-1$ malicious servers and $d-1$ malicious clients. Our protocol leaks noisy access patterns over the honest clients' queries, in the form of a differentially private noisy histogram $\mathcal{H}(Q) = H_{\text{honest}}(Q) + \chi(\epsilon, \delta)$.

Our protocol is secure up to *selective* abort, and does not guarantee fairness. Adversarial servers may elect to stop responding to queries, effectively aborting the entire protocol. Furthermore, they can do so selectively for certain queries. All servers can decide to drop queries at random, the frontend server can drop queries based on the identity of their clients, and the backend server can drop queries based on their value.

We stress that an adversary cannot drop a query based on the conjunction of the client's identity and the value, regardless of which subset of servers gets corrupted. Also, an adversary can only drop a query, but cannot convince a client to accept an incorrect response, since clients can validate the correctness of received responses locally.

3.3 Interpreting Security

We protect the privacy of any particular query made by an honest client with the guarantees of differential privacy. In particular, we consider two neighboring batches of queries Q and Q' over d clients. The two query sets consists of identical queries, except for a single client c_i, who makes two different queries $q \neq q'$ in the two batches respectively. We formalize this in definition 1.

Definition 1 (Differentially Private PIR Access Patterns)**.** *For any privacy parameters ϵ, δ, and every two query sets Q, Q' that differ on a single query i (i.e. $\forall j \neq i, Q[j] = Q'[j]$), the probabilities of our protocol producing identical access pattern histograms are (ϵ, δ)-similar when run on either set:*

$$Pr[\mathcal{H}(Q) = H] \leq e^{\epsilon} Pr[\mathcal{H}(Q') = H] + \delta$$

Our definition uses the substitution formulation of DP, rather than the more common addition/removal; see [65, §1.6] for details. Substitution is commonly used in secure computation protocols involving DP leakage [54]. We use this variant since our protocol does not hide whether a client made a query in a batch or not: the adversary already knows this e.g. by observing IP addresses associated to queries. Instead, we hide the value of the query itself. Our formulation is more conservative adding twice the expected amount of noise queries, since the sensitivity is 2 in our formulation and 1 in the other.

One way to interpret our differentially private guarantees is to note that it provides any client with plausible deniability: a client that made query q can claim that her true query was any $q' \neq q$, and external distinguishers cannot falsify this claim since the probability of either case inducing any same observed histogram of access patterns is similar.

Whereas traditional differential privacy mechanisms trade privacy for accuracy, our scheme trades privacy for performance, such that increasing privacy (i.e. lowering ϵ and δ) results in additional noise queries and load on the system.

We show the security of our protocol using a simulation-based proof, where the ideal functionality reveals a leakage function to the simulator. We then argue separately that this leakage function is (ϵ, δ)-DP. Simulation-based security holds when the protocol is sequentially composed, which extends our security to cases where several online stages are pooled together into a single offline stage. Furthermore, sequential composition combines nicely with the DP composition theorem, when several instances of the protocol are run over (potentially) correlated queries.

Note that our leakage is defined over plaintext values in the database. In the ORAM domain, it is common to define leakage over ciphers, since the database is encrypted or garbled. In our setting, these two definitions are equivalent. The adversary can carefully select the portion of noise

queries it controls, and correlate that with the observed histogram over ciphers, to map ciphers back to their original plaintexts.

4 Incremental Non-malleable Secret Sharing

Mixnets traditionally organize computation sequentially in a chain over servers, and rely on public key onion encryption in order to pass secrets meant for subsequent servers through previous servers. However, this induces a large number of public key operations, proportional to $m \times |batch|$. We use a novel cheaper arithmetic-based secret sharing scheme instead of onion encryption during our own online stage.

The secret sharing scheme provides similar security guarantees to onion encryption, to ensure that input and output queries are untraceable by external adversaries:

1. *Secrecy:* As long as one of the shares is unknown, reconstruction cannot be carried out by an adversary.

2. *Incremental Reconstruction:* A server that only knows a single secret share and a running tally must be able to combine them to produce a new tally. The new tally must produce the original secret when combined with the remaining shares.

3. *Independence:* An adversary cannot link any partially reconstructed output from a set of outputs to any shared input in the corresponding input set.

4. *Non-Malleability:* An adversary who perturbs any given share cannot guarantee that the output of reconstruction with that perturbed share satisfies any desired relationship. In particular, the adversary cannot perturb shares such that reconstruction yields a specific value (e.g., 0), or a specific function of the original secret (e.g., adding a fixed offset).

Non-malleability is critical for preserving security when the last (backend) server is corrupted. The backend can observe the final reconstructed values of all queries to identify queries perturbed by earlier colluding servers. If the perturbation can be undone (e.g. by removing a fixed offset), then the backend can learn the value of the query, and link it to information known by other servers, such as the identity of its client.

Secrecy and independence can be easily achieved with any appropriately-thresholded secret sharing scheme. However, non-malleability excludes several incremental schemes, such as additive or XOR-based sharing. While several non-malleable secret sharing schemes exist [10, 35], they don't satisfy our incremental design. It would have been possible to use different primitives that satisfy these properties, such as authenticated onion symmetric-key encryption. However, these operations remain more expensive than simple information theoretic secret sharing schemes that can be implemented with a handful of arithmetic operations.

Our Sharing Scheme Given a secret q, a prime modulus z, and an integer m, our scheme produces $m+1$ pairs $(x_0, y_0), (x_1, y_1), ..., (x_m, y_m)$, each representing a single share of q. All x and y shares are chosen independently at random from $\mathbb{F}_z$ and $\mathbb{F}_z^*$ respectively, except for the very first pair x_0, y_0, whose values are set to:

$$x_0 = ((q - x_m) \times y_m^{-1} \; ... - x_1) \times y_1^{-1} \; mod \; z, \quad y_0 = 0$$

All shares except the first one can be selected prior to knowing q. This is important for our offline stage. The modulus z must be as big as the key size in the underlying database (64 bits in our experiments). The secret q can be reconstructed *incrementally* from a running tally l_{j-1} and one share (x_j, y_j) to produce a new tally l_j, such that $l_m = q$:

$$l_0 = y_0 \times 1 + x_0 \ mod \ z$$
$$l_j = y_j \times l_{j-1} + x_j \ mod \ z$$

Incremental Non-malleability Security Game We describe the following security game between an adversary and an incremental sharing dealer. The dealer possesses a fixed secret share h, while the adversary produces two partial tallies l, l' of its own choosing, such that $l \neq l'$.

In the real world, the dealer computes the next two tallies $R(l, h)$, $R(l', h)$ by applying our scheme's incremental reconstruction function to l and l' with its fixed share. In the ideal world, the dealer computes $R(l, h)$ correctly, and produces a random value for the second tally r. The aim of the adversary is to distinguish between the real and ideal world.

The adversary here represents all parties in an incremental reconstruction protocol up to and excluding the last honest party, the first tally produced by the dealer represents the non-tampered tally resulting from all of these parties following the reconstruction protocol correctly, while the second tally represents the tampered tally resulting from some of these parties using tampered shares (or deviating arbitrarily from the protocol). The dealer represents the last honest party.

The game demonstrates that regardless of what tampering the adversary performs (it has the freedom to select any tallies), the next tampered partial tally is indistinguishable from a random one, even knowing what the non-partial tally should have been. This is a strong guarantee that implies that the actual tally cannot be extracted from the tampered one, since the existence of any such extraction mechanism provides a way to distinguish the real and ideal world. In Appendix D, we prove that our construction satisfies this security game.

5 Our DP-PIR Protocol

Offline Stage Our offline stage consists of a single sequential pass over the m servers. Clients generate random secrets locally, and submit them after onion encryption to the first server in the chain. The first server receives all such incoming messages from clients, until a configurable granularity is reached, e.g. after a certain time window passes or a number of messages is received. All incoming messages at that point constitutes the input set for that server, which get processed by that server into an output set of a larger size, since the server also injects its own messages into the set, consisting of its anonymous random secrets that she needs to perform the noise addition in the online stage later on.

The number of shared anonymous secrets injected by a server and stored at subsequent servers must suffice to handle all noise queries that the server needs to inject in the online stage. As a result, our protocol requires the server to pre-sample the histogram of noise during the offline stage to determine the total count of noise queries that will be used during the online stage, and inject precisely the needed amount of shared anonymous secrets into the output set.

The output set contains onion ciphers, encrypted under the keys of the subsequent servers in the chain. None of the plaintexts decrypted by the current server survives, they are all consumed and stored in the server's local mapping for use during the online stage. No linkage between

Algorithm 1 Client Offline Stage

Input: Nothing.

Output state at the client: a list of anonymous secrets $[a_0, ..., a_m]$, one per each of the m servers. The client uses this in the online stage.

Output to s_1: Onion encryption of $a_0, ..., a_m$.

1. **Generate Random Values**: For each Server s_j, the client generates 4 values all sampled uniformly at random: (1) A globally unique identifier t_j. (2) Two incremental pre-shares $x_j \in \{0, ..., z\}$ and $y_j \in \{1, ...z\}$. (3) An additive pre-share $e_j \in \{0, ..., z\}$.

2. **Build Shared Anonymous secrets**: The client builds $a_j = (t_j, t_{j+1}, x_j, y_j, e_j)$, for every server $1 \leq j \leq m$, using the generated random values above, with $t_{m+1} = \perp$. These secrets are stored by the client for later use in the online stage.

3. **Onion Encryption**: The client onion encrypts the secrets using the correspond server's public key, such that $OEnc(m) = Enc(sk_m,\ a_m)$ and $OEnc(j) = Enc(sk_j,\ a_j :: OEnc(j{+}1))$.

4. **Secrets Submission**: The client sends the onion cipher $OEnc(1)$ to server s_1. The client can leave the protocol as soon as receipt of this message is acknowledged.

messages in the input and output sets is possible without knowing the server's secret key, since the ciphers in the input cannot be used to distinguish between (sub-components of) their plaintexts (by CPA-security), and since the output set is uniformly shuffled.

Online Stage Our online stage is structured similarly to the offline stage. However, it requires going through the chain of servers twice. The first phase moves from the clients to the backend server, where every server injects its noise queries into the running set of queries, and incrementally reconstructs the values of received queries. The second phase moves in the opposite direction, with every server removing responses to their noise queries, and incrementally reconstructing the values of received responses, until the final value of a response is reconstructed by its corresponding client.

The backend operates differently than the rest of the servers. The backend gets to see the reconstructed query set, and find their corresponding responses in the underlying database via direct look-ups. The backend need not add any noise queries, which alleviates the need for shuffling at the backend.

Discussion The security of both offline and online stages rely on the same intuition. First, an adversary that observes the input and output sets of an honest server should not be able to link any output message to its input. Second, the adversary must not be able to distinguish outputs corresponding to real queries from noise injected by that server.

An adversary cannot link a message in the output set to its corresponding input message, since the honest server re-randomizes these messages. In the offline stage this re-randomization is performed with onion-decryption, while the online stage performs it using our non-malleable incremental reconstruction and additive secret sharing for the two phases respectively. We do not need to use a non-malleable secret sharing scheme for response handling, since the adversary cannot observe the final response output, which is only revealed to the corresponding client, and

Algorithm 2 Server s_j Offline Stage

Configuration: The underlying database $T = K; V$, and privacy parameters ϵ, δ.
Input from $\mathbf{s_{j-1}}$ or clients if $\mathbf{j = 1}$: A set of onion ciphers of anonymous secrets, one per each incoming request.
Output state at $\mathbf{s_j}$: A mapping M of unique tag t_j^i to its corresponding shared anonymous secrets a_j^i used to handle incoming queries during the online stage. A list of generated anonymous secrets L used to create noise queries during the online stage. A sampled histogram $\mathcal{N}$ of noise queries to use in the online stage.
Output to server $\mathbf{s_{j+1}}$: A set output onion ciphers corresponding to input onion ciphers and noise generated by s_j.

1. **Onion Decryption**: For every received onion cipher $OEnc(j)_i$, the server decrypts the cipher with its secret key sk_j, producing a_j^i and $OEnc(j+1)_i$.

2. **Anonymous Secret Installation**: For every decrypted secret $a_j^i = (t_j^i, t_{j+1}^i, x_j^i, y_j^i, e_j^i)$, the server stores entry $(t_{j+1}^i, x_j^i, y_j^i, e_j^i)$ at $M[t_j^i]$ for later use in the online stage.

If $j < m$:

3. **Noise Pre-Sampling**: The server samples a histogram representing counts of noisy queries to add for every key in the database $\mathcal{N} \leftarrow \chi(\epsilon, \delta)$, and computes the total count of this noise $S = \sum \mathcal{N}$.

4. **Build shared anonymous secrets for noise**: The server generates S many anonymous secrets and onion encrypts them for all servers $s_{j'} > s_j$, using the same algorithm as the client. The server stores these secrets in L.

5. **Shuffling and Forwarding**: The server shuffles all onion ciphers, either decrypted from incoming messages, or generated by the previous step, and sends them over to the next server s_{j+1}.

thus cannot observe the effects of the perturbation.

Shuffling in the noise with the re-randomized messages ensure that they are indistinguishable. A consequence of this is that a server cannot send out any output message until it receives the entirety of its input set from the previous server to avoid leaking information about the permutation used. Idle servers further along the chain can use this time to perform input independent components of the protocol, such as sampling the noise, building and encrypting their anonymous secrets, or sampling a shuffling order.

A malicious server may deviate from this protocol in a variety of ways: it may de-shuffle responses incorrectly (by using a different order), attach a different tag to a query than the one the offline stage dictates, or set the output value corresponding to a query or response arbitrarily (including via the use of an incorrect pre-share). The offline stage does not provide a malicious server with additional deviation capability: any deviation in the offline stage can be reformulated as a deviation in the online stage, after carrying out the offline stage honestly, with both deviations achieving identical effects. Finally, a backend server may choose to provide incorrect responses to queries by ignoring the underlying database.

Algorithm 3 Client Online Stage

Input: A query q.

Input state at the client: a shared anonymous secrets $a_j = (t_j, t_{j+1}, x_j, y_j, e_j)$ per server s_j generated by the offline stage.

Output: A response r, corresponding to the value associated to q in the database.

1. **Compute Final Incremental Secret Share**: Client computes $l = x_0$, so that $(x_0, 0)$ combined with $(x_1, y_1), ..., (x_m, y_m)$ is a valid sharing of q, per our incremental secret sharing scheme.

2. **Query Submission**: Client sends (t_1, l) to server s_1.

3. **Response Reconstruction**: Client receives response r_1 from s_1. Client reconstructs $r' = r_1 - \sum e_j \ mod \ z$.

4. **Response verification**: The client ensures that the response $r' = (q, r)$, where q is the original query. Furthermore, it verifies that it is signed by $pk = (pk_1, ..., pk_m)$.

Each of these deviations has the same effect: The non-malleability of both our sharing scheme and onion encryption ensures that mishandled messages reconstruct to random values. While mishandled responses will not pass client-side verification, unless the adversary can forge signatures. In either case, the affected clients will identify that the output they received is incorrect and reject it. Ergo, servers can only use this approach to selectively deny service to some clients or queries. A malicious frontend can deny service to any desired subset of clients since it knows which queries correspond to which clients, a malicious backend can deny service to any number of client who queried a particular entry in the database, and any server can deny service to random clients. The backend and frontend capabilities cannot be combined even when colluding since at least one honest server exists between the frontend and backend. These guarantees are similar to those of Vuvuzela [66] and many other mixnet systems.

Formal Security We rigorously specify our security guarantee in Theorem 1, which refers to the ideal functionality defined in Algorithm 5. The ideal functionality formalizes our notion of "selective" abort. In particular, it formalizes capabilities of the adversary to deny service to certain clients based on client identity, query value, or neither (depending on which servers are corrupted).

Theorem 1 (Security of our protocol Π)**.** *For any set A of adversarial colluding servers and clients, including no more than $m-1$ servers, there exists a simulator S, such that for client inputs $q^1, ..., q^d$, we have:*

$$View_{Real}(\Pi, A, (q^1, ..., q^d)) \approx View_{Ideal}(\mathcal{F}, \mathcal{S}, (q^1, ..., q^d))$$

A construction of the simulator and proof for Theorem 1 are shown in Appendices A and B.

Differential Privacy Our security theorem contains leakage revealed to the backend server in the form of a histogram over queries made by honest clients and honest servers. Our privacy guarantees hinge on this leakage being differentially private, which in turn depend on the underlying

Algorithm 4 Server s_j Online Stage

State at $\mathbf{s_j}$: The mapping M, list L, and noise histogram $\mathcal{N}$ stored from the offline stage.
Input from $\mathbf{s_{j-1}}$ or clients if $\mathbf{j = 1}$: A list of queries (t_i^j, l_j^i).
Output to $\mathbf{s_{j-1}}$ or clients: A list of responses r_j^i corresponding to each query i.

1. **Anonymous Secret Lookup**: For every received query (t_i^j, l_j^i), the server finds $M[t_i^j] = (t_i^{j+1}, x_i^j, y_i^j, e_i^j)$.
2. **Query Handling**: For every received query, the server computes output query $(t_i^{j+1}, R(l_j^i, (x_i^j, y_i^j)))$, where R is our scheme's incremental reconstruction function.

If $j < m$:

4. **Noise injection**: The server makes output queries per stored noise histogram $\mathcal{N}$, using the stored list of anonymous secrets L and the client's online protocol. By construction, there are exactly as many secrets in L as overall queries in $\mathcal{N}$.
5. **Shuffling and Forwarding:** The server shuffles all output queries, both real and noise, and sends them over to the next server s_{j+1}. The server waits until she receives the corresponding responses from s_{j+1}, and de-shuffles them using the inverse permutation.
6. **Response Handling:** Received responses corresponding to noise queries generated by this server are discarded. For every remaining received response r_{j+1}^i, the server computes the output response $r_j^i = r_{j+1}^i + e_i^j \bmod z$.
7. **Response Forwarding:** The server sends all output responses r_j^i to s_{j-1}, or the corresponding client c_i if $j = 1$.

If $j = m$:

4. **Response Lookup**: The backend server does not need to inject any noise or shuffle. By construction, step (2) computes $(\perp, q_i)$ for each received query. The backend find the corresponding response r_i' by looking up q_i in the database. If q_i was not found in the database (because a malicious party mishandled it), we set r_i' to a random value.
5. **Response Handling:** The backend computes the output responses $r_j^i = r_i' + e_i^j \bmod z$, and sends them to s_{j-1}.

noise distribution. Several differentially private mechanisms exist in the literature for a variety of functions and statistics. Our application requires that the noise we add be non-negative (we cannot add a negative amount of queries) and integer (e.g., we cannot add half a query). It is desirable to have an upper bound on the amount of noise queries. Algorithm 6 details how we modify the standard Laplace mechanism to satisfy these properties. Appendix C proves that this mechanism indeed achieves (ϵ, δ)-differential privacy.

Algorithm 5 Ideal Functionality $\mathcal{F}$

Input: A set of queries q^i, one per client, the underlying database $T = K; V$, and privacy parameters ϵ, δ.
Output: A set of responses r^i, one per client, either equal to the correct response or $\perp$.
Leakage: A noisy histogram $\mathcal{H}$ revealed to s_m.

1. if s_1 is corrupted, $\mathcal{F}$ receives a list of client identities from the adversary. These clients are excluded from the next steps, and receive $\perp$ outputs.
2. $\mathcal{F}$ reveals the noised histogram $\mathcal{H} = H_{\text{honest}} + \mathcal{N}$ to the backend server s_m, where H_{honest} is the histogram of queries made by *honest* clients not excluded by the previous step, and $\mathcal{N}$ is sampled at random from the distribution of noise $\chi(\epsilon, \delta)$.
3. if the backend is corrupted, $\mathcal{F}$ receives a list of counts c_i for every entry in the database k_i, and outputs $\perp$ to c_i-many clients, randomly chosen among the remaining clients that queried k_i.
4. if any server, other than s_m and s_1, is corrupted, $\mathcal{F}$ receives a number c, and outputs $\perp$ to c-many clients, randomly chosen among the remaining clients.
5. if s_1 is corrupted, $\mathcal{F}$ receives an additional list of client identities to receive $\perp$.
6. $\mathcal{F}$ outputs $T[q^i]$ for every client i not excluded by any of the steps above.

Algorithm 6 Noise Query Sampling Mechanism $\chi(\epsilon, \delta)$

Input: The size of the database $|T|$, and privacy parameters ϵ, δ.
Output: A histogram $\mathcal{N}$ over T representing how many noise queries to add per database entry.

1. $B := |CDF^{-1}_{Laplace(0,2/\epsilon)}(\frac{\delta}{2})|$.

For every $i \in |T|$:

2. $l_i \leftarrow Laplace(0, \frac{2}{\epsilon})$.
3. $l'_i := max(-B, min(B, l_i)) + B$.
4. $\mathcal{N}_i := \text{floor}(l'_i)$.

6 Scaling and Parallelization

Many existing PIR protocols can be trivially scaled over additional resources, by running completely independent parallel instances of them on different machines. This approach is not ideal for our protocol: each instance needs to add an independent set of noise queries, since each instance reveals an independent histogram of its queries. Instead, our protocol is more suited for parallelizing a single instance over additional resources, such that only a single histogram is revealed without needing to add ancillary noise queries.

In a non-parallel setting, the notions of a party and a server are identical. For scaling, we allow

parties to operate multiple machines. These machines form a single trust domain. In other words, we assume that if one of these machines is corrupted, all other machines belonging to the same party are also corrupted. This maintains our security guarantees at the level of a party. Particularly, the protocol remains secure if one party (and all its machines) is honest. Machines owned by the same party share all their secret state, including anonymous secrets installed at any of them during the offline stage, as well as the noise query they select.

Any machine m_i^j belonging to party j only communicates with a single machine m_i^{j-1} and m_i^{j+1} from the preceding and succeeding parties, in order to receive inputs and send outputs respectively. A machine also communicates with all other machines belonging to the same party for shuffling.

Distributing Noise Generation Our protocol generates noise independently for each entry in the database, we can parallelize the generation by assigning each machine a subset of database entries to generate noise for, e.g. m_i^j is responsible for generating all noise queries corresponding to keys $\{k|\ k\ \%\ j = 0\}$. This distribution is limited by the size of the database. If parallelizing the noise generation beyond this limit is required, an alternate additive noise distribution (e.g. Poisson [64]) can be used instead, which allows several machines to sample noise for the same database entry from a proportionally smaller distribution.

Distributed Shuffling Machines belonging to the same party must have identical probability of outputting any input query after shuffling, regardless of which server it was initially sent to. An ideal algorithm guarantees that the number of queries remains uniformly distributed among machines after shuffling. We choose one that requires no online coordination to ensure it maintains perfect scaling. Machines belonging to the same party agree on a single secret seed ahead of time. They use this seed to simulate a non-distributed global shuffling locally. Assuming that the total number of input queries is l, each machine m_i^j samples l random numbers using the shared seed locally to uniformly sample the same global permutation P of size l using Knuth shuffle. Each machine m_i^j need only retain $P[\frac{i\times l}{m} : \frac{(i+1)l}{m}]$ which determines the new indices of each of its input queries. The target machine that each query q should be sent to can be computed by $P[q]\%\frac{l}{m}$. This algorithm performs optimal communication $\frac{l}{m}$ per machine but requires each machine to perform CPU work linear in the overall number of queries to sample the overall permutation. This work is independent of the actual queries, and can be done ahead of time (e.g. while queries are being batched or processed by previous parties).

Distributing Offline Anonymous Secrets Since the offline and online stages perform independent shuffling, it is unlikely that an online query be processed by the same machine as its associated offline secret. We require all machines belonging to the same party to share all secrets they installed during the offline stage, so that any of them can quickly retrieve the needed ones during the online stage. For small scale applications, it may be suitable for each machine to maintain a copy of all these secrets in its Main memory. For larger scale applications, it may be more appropriate to store that copy on a local SSD, or utilize an appropriate key-value storage or in-memory distributed file system, especially ones that support Remote direct memory access (RDMA) [55, 70], offloaded lookups to the remote machine [1], and batching of multiple requests to minimze network costs [4].

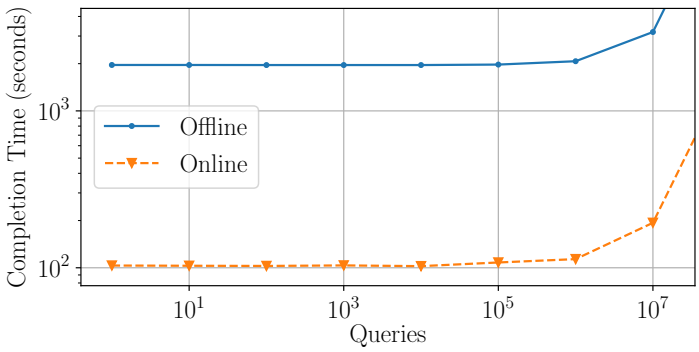

Figure 5: Completion time for varying number of queries against a 100K database (logscale)

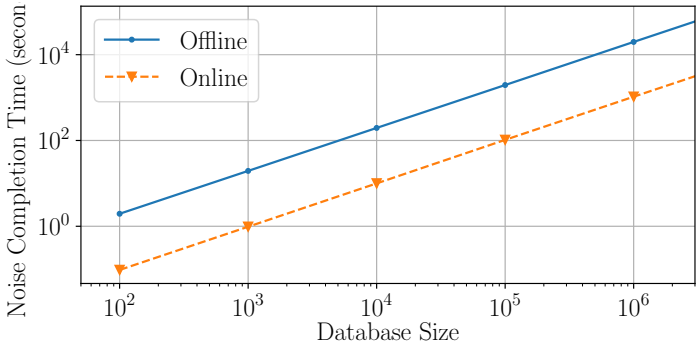

Figure 6: Completion time for a batch consisting only of noise queries against varying database sizes (logscale)

7 Evaluation

We evaluate DP-PIR and its applicability via several experiments. Our evaluation focuses on the following questions:

1. How does DP-PIR scale with application parameters?
2. How does DP-PIR compare to other PIR protocols?
3. What is the latency of DP-PIR?
4. When is DP-PIR ideal and when is it unsuitable?

Experiment Setup Our various experiments measure the server completion time for a batch of queries. For the online stage, this is the total wall time taken from the moment the first server receives a complete batch ready for processing, until that batch is completely processed by the entire protocol, and its outputs are ready to be sent to clients. For the offline stage, the measurements start when the complete batch is received by the first server, and ends when all servers finished processing and installing the secrets. Measurements include the time spent in CPU

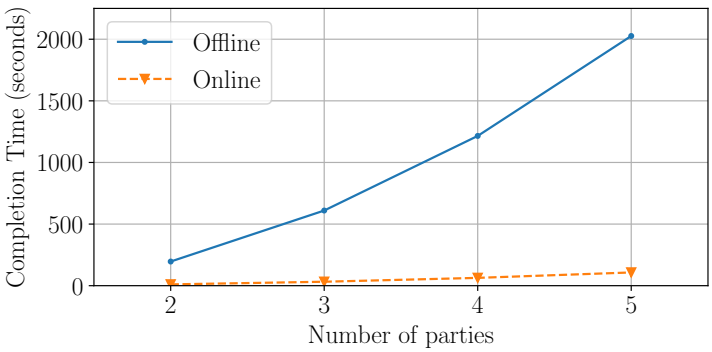

Figure 7: Completion time for varying number of parties with 100K queries against a 10K database (yaxis logscale)

performing various computations from the protocol, as well as time spent waiting for network IO as messages get exchanged between servers. Our measurements does not include client processing or round-trip time, we do not have the capacity to faithfully deploy hundreds of thousands of clients concurrently. Instead, we feed the servers pre-generated queries from a trace file. This does not affect our measurements, since our protocol and measurements start after a complete batch of queries is received.

We ran these experiments with $\epsilon = 0.1$ and $\delta = 10^{-}6$ on general purpose AWS t2.large instances with two 3GHz cores, 8 GB RAM, and low (around 100Mbps) network bandwidth. These are relatively small machines costing less than \$0.1 per hour. We chose these machines to demonstrate that our protocol can be effectively deployed on cheap commodity hardware, and to ensure our protocol does not gain an implicit setup advantage against Checklist and SealPIR (more on this later). We implemented our protocol using a C++ prototype with about 6.5K lines of code. Our prototype relies on libsodium for public key operations and TCP sockets. We use a database with keys and values that are 8 bytes each.

Scaling Figures 5 and 6 show how our protocol scales with the number of queries and database size respectively. Our runtime is dominated by noise queries when the number of queries is smaller than the size of the database, and begins to increase with the number of queries as they exceed it. For large enough number of queries, our runtime scales linearly as the overhead of noise queries is amortized away over the real queries. Our protocol scales linearly with the size of the database. The cost of processing any input query *in isolation* is constant and does not depend on the database size, which only affects the number of noise queries added by our protocol. The offline stage is about 20x more expensive than our online stage. This is expected since the offline stage performs a public key operation for each corresponding modular arithmetic operation in the online stage.

Figure 7 shows that our protocol scales super-linearly in the number of parties. Our protocol is most efficient when only two parties are involved. When the number of parties increases, a query has to pass through more servers as it crosses the chain. This is more pronounced in the offline stage, as it additionally increases the size and layers of each onion cipher. All this is compounded by each server naively adding in the full required amount of noise queries independently, since only one of them may be honest. Adding less noise by relying on additional assumptions (e.g. honest majority) is an open problem, which can help improve our scaling with the number of parties,

Machines / Party	Completion time (seconds)	
	Offline	Online
1	2021	113.5
2	1048	70.63
4	520	35.88
8	261	18.2

Table 2: Horizontal scaling with 1M queries and a 100K DB

δ \ ϵ	**1**	**0.1**	**0.01**	**0.0001**
$\mathbf{10^{-5}}$	23	230	2302	23025
$\mathbf{10^{-6}}$	27	276	2763	27631
$\mathbf{10^{-7}}$	32	322	3223	32236

Table 3: Expected number of noise queries B per database element as a function of different ϵ (columns) and δ (rows)

and can have important consequences to mixnets, the DP shuffling model, and DP mechanisms in general. Techniques such as noise verification [47] may be useful to ensure that (partial) noise generated by an honest server is not tampered with by future malicious servers.

Table 2 demonstrates how our protocol scales horizontally. Parallelizing the online stage primarily parallelizes communication. However, parallel shuffling introduces an additional round of communication per party. As a result, our online stage speed up when using 2 machines is not $2x$. We exhibit linear speedups as the number of machines exceeds 2.

Finally, the expected number of noise queries added per database element is a function of ϵ and δ. Table 3 lists this expected number for various combinations of ϵ and δ. The expectation increases linearly as ϵ decreases but scales better with δ. Our protocol trades security for performance. It can efficiently amortize the cost of independent queries due to its relaxed DP security guarantees. As ϵ becomes smaller, this relaxation becomes less meaningful, as the DP security guarantees approach those of computational security. While linear scaling with $\frac{1}{\epsilon}$ appears to be intrinsic to our protocol, we believe it may be possible to reduce the scaling constant, by using different basis distributions that are inherently non-negative or discrete (e.g. Poisson [64] or Geometric [54]), or by adapting recent work on privacy amplification [24, 31] that achieves the same level of privacy in various DP mechanisms using less noise with oblivious shuffling.

Comparison to Other Protocols Section 2 demonstrates that the performance of DP-PIR versus other state of the art protocols is governed by the ratio of queries to database size. Our results show that our protocol can achieve significant speedups (over 2x, 5x, 10x and beyond) for various modest ratios, especially for large databases. While the overall trends shown in section 2 are an intrinsic benefit of our design, the exact ratio and runtime values shown depend on the application parameters and deployment setting.

We consciously chose to use t2.large instances in our experiments to avoid inflating our performance because of the setup. These instances provide much better CPU and memory capabilities than network bandwidth. This benefits Checklist and SealPIR, which are both CPU-bound protocols that perform no communication during server computation. Our online network-hungry

protocol is bottle-necked by the low network resources, which are unable to match the throughput of the CPU, causing CPU utilization to remain low. This is less drastic for our offline protocol, which performs frequent CPU-heavy public key operations. Like most MPC applications, our protocol prefers having high network bandwidth, such as when the parties are co-located within the same placement group, or are connected via high-speed LANs. These setups can commonly provide bandwidth exceeding 10 Gbps. Any such increase in network bandwidth translates to a direct performance boost for our protocol.

Therefore, the exact speedup we achieve for a given ratio highly depends on the setup. Our protocol will perform proportionally better than our experiments in more network optimized setups, while the other protocols benefit more from more compute-optimized setups. Similarly, we can deduce from tables 2 and 3 that this ratio linearly increases or decreases with the privacy budget and number of machines.

Throughput and Latency The completion times reported are a product of the throughput and number of queries. Furthermore, they constitute the majority of the tail latency, since a query can only be sent back to a client after the entire batch has been completely processed, so that no information about the secret permutation is leaked. Specifically, the tail latency is the sum of the round trip cost between the client and server, the batching frequency (i.e the window during which incoming queries constitute the same batch), and the aforementioned completion time. We can always improve latency by adding more machines or resources per party, as shown in table 2. The latency of our offline stage is irrelevant, as clients disconnect as soon as they submit their secrets to the first server.

For reasonable ratios of queries to database size and small batching frequencies, this is comparable to other existing protocols. Existing protocols incur large overheads per query and process queries sequentially. When a large number of queries is received in a small window, these queries get queued up as the system goes through them for processing, and the tail latency includes the time to process all previous queries. In fact, when the ratio of queries to database size is sufficient for our protocol to achieve a certain speedup in completion time, it follows that our tail latency exhibits a similar speedup. This introduces an important consideration: it is ideal to configure our protocol with a smaller batching window to minimize latency. At the same time, this window must be large enough to ensure that enough queries are batched in to reach an effective ratio of queries to database size.

The latency of DP-PIR and other shuffling-oriented systems can be radically improved by alternative mechanisms that generate a stream of noise queries intermixed with real ones without strict batching. Some work on streaming DP mechanisms exists [28,29,39,41], and may be complemented with adding random delays to messages to mimic shuffling [42].

When should you use DP-PIR? Our protocol offers relaxed differentially private security guarantees, and should only be used for applications where this is suitable. The primary performance factor that determines the applicability of our protocol is the ratio of queries to database size. Our experiments provide evidence that our protocol is almost always preferred when this ratio is above 10, as that guarantees significant speedups in most practical settings. On the other hand, our protocol is almost always less efficient than its traditional counterparts when that ratio is below $\frac{1}{10}$, since this ratio does not provide enough queries to effectively amortize our fixed noise overheads. Applications with ratios in between may be suitable for our protocol, depending on how

close their ratio is to either extremes relative to how much other application parameters suit our protocol. These parameters include the size of the database, desired privacy budget (especially ϵ), and client resources, as well as whether the deployment environment is more network or compute optimized.

8 Related Work

Section 2 discusses existing work on Private Information Retrieval. Here, we discuss related work from other areas.

Mixnets Traditional mixnets [21] consist of various parties that *sequentially* process a batch of onion-ciphers, and output a uniformly random permutation of their corresponding plaintexts. Various Mixnet systems [13, 32] add *cover traffic* to obfuscate various traffic patterns. However, ad-hoc cover traffic is shown to leak information over time [53].

Recent work mitigates this by relying on secure multiparty computation [7] or differential privacy. Vuvuzela [66] adds noise traffic from a suitable distribution to achieve formal differential privacy guarantees over leaked traffic patterns, and Stadium [64] improves on its performance by allowing parallel noise generation and permutation. Similar techniques have been used in private messaging systems [47], and in differential privacy models that utilize shuffling for privacy amplification [31] or for introducing a *shuffled* model that lies in between the central and the local models [24].

While our work has similarities to Vuvuzela, and Mixnets overall, it provides better latency as it makes the (online) processing of every query significantly cheaper, by eliminating *all* crypto operations, relying only on cheap modular arithmetic. This is critical for the success of adapting such Mixnets techniques to PIR and other applications, where the latency of handling one query in a batch is limited by the overall processing time for the entire batch (and not just the query).

Differential Privacy and Access Patterns Using differential privacy to efficiently hide access patterns of various protocols has seen increasing interest in the literature. Earlier work relaxes the security guarantees of PIR to be differentially private [61] in the semi-honest setting, by putting the burden on clients to hide their actual query among a number of random queries selected from an appropriate distribution. This assumes that clients are all honest, and requires the database to be replicated over a large number of servers.

A larger body of work relaxes the security guarantees of Oblivious RAM (ORAM), a primitive where a single client obliviously reads and writes to a private remote database [33, 34], to be differentially private. Extensions of ORAM address multi-client settings [52], where the obliviousness of access patterns must be guaranteed over all users. Differentially private ORAM relaxations [67] guarantee that neighboring access patterns (those that differ in the location of a single access) occur with similar probability.

Differentially private access patterns have been studied for searchable encryption [22] and generic secure computation [54] where the ideal functionality outputs a leakage function over the input data that must itself be differentially private. The security notions from these various works are tightly connected. DP-PIR can be instantiated from DP-ORAM protocols [67] with similar differential privacy security relaxation. In both cases, privacy is provided at the level of a user query/access, and

the same noise distribution can be used to hide access patterns similarly to the classic differentially private histogram release example [30].

Secret Sharing Shamir Secret Sharing [60] allows a user to split her data among n parties such that any t of them can reconstruct the secret. Secret sharing schemes with additional properties have been studied for use in various applications. Some schemes, such as additive secret sharing, allow the secret to be reconstructed *incrementally* by combining a subset of shares of size k into a single share that can recover the original secret when combined with the remaining $n - k$ shares. *Non-malleable* secret sharing schemes [10,35] additionally protect against an adversary that can tamper with shares, and guarantees that tampered shares either reconstruct to the original message or to some random value. Recent work [5] shows generic transformations that construct non-malleable schemes from secret sharing schemes over the same access structure.

9 Conclusion

This paper introduces DP-PIR, a novel multi-server differntially private PIR protocol, specifically geared towards applications with a large number of queries relative to the size of the underlying database. Unlike existing work, DP-PIR amortizes the server work per query down to a constant when the number of queries approach the size of the database. DP-PIR exhibits concrete speedups up to and exceeding 10x compared to state of the art protocols. DP-PIR separates the space of PIR applications into 3 regions, governed by the ratio of the number of queries within a batch q over the size of the database n:

1. $\frac{\mathbf{q}}{\mathbf{n}} \geq \mathbf{10}$: These applications are ideal for DP-PIR, as their scale allows our amortization to truly shine. Our experiments demonstrate that DP-PIR can achieve significant speedups beyond 10x when compared to state of the art PIR protocols in this setting. Numerous applications fall in this region including software update checks, contact tracing, and content moderation.

2. $\frac{\mathbf{q}}{\mathbf{n}} < \frac{\mathbf{1}}{\mathbf{10}}$: These applications are unsuited for our protocol as the amortized overhead per query remains large. DP-PIR is not designed for applications in this region, and we recommend they continue to use traditional PIR protocols. Consider for example a privacy-preserving version of Google docs, where clients privately retrieve documents from an online service. The overall number of documents vastly outnumber how many documents get requested in any reasonable batching window.

3. $\frac{\mathbf{1}}{\mathbf{10}} \leq \frac{\mathbf{q}}{\mathbf{n}} < \mathbf{10}$: Applications in this region may exhibit roughly comparable performance using DP-PIR and other protocols. The optimal protocol to use for such applications depends on how close the ratio is to either extremes, as well as deployment parameters including the privacy budget, database size, server CPU vs bandwidth, and client resources.

Acknowledgments

The authors are grateful to Andrei Lapets, Frederick Jansen, Jens Schmuedderich, and Malte Schwarzkopf for their valuable feedback on earlier versions of this work. This material is supported

by the National Science Foundation under Grants No. 1414119, 1718135, and 1931714, by DARPA under Agreement No. HR00112020021, and by Honda Research Institutes.

References

[1] Apache ignite. https://github.com/apache/ignite. Accessed: 2020-12-01.

[2] App download and usage statistics (2020). https://www.businessofapps.com/data/app-statistics/. Accessed: 2021-06-02.

[3] There are over 3 billion active android devices. https://www.theverge.com/2021/5/18/22440813/android-devices-active-number-smartphones-google-2021. Accessed: 2021-06-02.

[4] Turbocharge amazon s3 with amazon elasticache for redis. https://aws.amazon.com/blogs/storage/turbocharge-amazon-s3-with-amazon-elasticache-for-redis/. Accessed: 2020-12-01.

[5] Divesh Aggarwal, Ivan Damgr^ard, Jesper Buus Nielsen, Maciej Obremski, Erick Purwanto, João Ribeiro, and Mark Simkin. Stronger leakage-resilient and non-malleable secret sharing schemes for general access structures. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 510–539, Cham, 2019. Springer International Publishing.

[6] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, 2016(2):155–174, 2016.

[7] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1217–1234, 2017.

[8] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society, 2018.

[9] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, Savannah, GA, November 2016. USENIX Association.

[10] Saikrishna Badrinarayanan and Akshayaram Srinivasan. Revisiting non-malleable secret sharing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 593–622. Springer, 2019.

[11] Amos Beimel and Yuval Ishai. Information-theoretic private information retrieval: A unified construction. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 912–926, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[12] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers' computation in private information retrieval: Pir with preprocessing. *Journal of Cryptology*, 17(2):125–151, 2004.

[13] Oliver Berthold and Heinrich Langos. Dummy traffic against long term intersection attacks. In Roger Dingledine and Paul Syverson, editors, *Privacy Enhancing Technologies*, pages 110–128, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[14] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. Cryptology ePrint Archive, Report 2021/017, 2021. https://eprint.iacr.org/2021/017.

[15] Dan Boneh, Kevin Lewi, and David J. Wu. Constraining pseudorandom functions privately. In *Public Key Cryptography (2)*, volume 10175 of *Lecture Notes in Computer Science*, pages 494–524. Springer, 2017.

[16] Nikita Borisov, George Danezis, and Ian Goldberg. Dp5: A private presence service. *Proceedings on Privacy Enhancing Technologies*, 2015(2):4–24, 2015.

[17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *Theory of Cryptography Conference*, pages 662–693. Springer, 2017.

[18] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 402–414. Springer, 1999.

[19] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 694–726, Cham, 2017. Springer International Publishing.

[20] Justin Chan, Landon P. Cox, Dean P. Foster, Shyam Gollakota, Eric Horvitz, Joseph Jaeger, Sham M. Kakade, Tadayoshi Kohno, John Langford, Jonathan Larson, Puneet Sharma, Sudheesh Singanamalla, Jacob E. Sunshine, and Stefano Tessaro. PACT: privacy-sensitive protocols and mechanisms for mobile contact tracing. *IEEE Data Eng. Bull.*, 43(2):15–35, 2020.

[21] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, February 1981.

[22] G. Chen, T. Lai, M. K. Reiter, and Y. Zhang. Differentially private access patterns for searchable symmetric encryption. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 810–818, 2018.

[23] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private group messaging with hidden access patterns, 2020.

[24] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 375–403, Cham, 2019. Springer International Publishing.

[25] Benny Chor and Niv Gilboa. Computationally private information retrieval. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 304–313, 1997.

[26] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.

[27] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 44–75, Cham, 2020. Springer International Publishing.

[28] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N Rothblum. Differential privacy under continual observation. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 715–724, 2010.

[29] Cynthia Dwork, Moni Naor, Toniann Pitassi, Guy N Rothblum, and Sergey Yekhanin. Pan-private streaming algorithms. In *ICS*, pages 66–80, 2010.

[30] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

[31] Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Abhradeep Thakurta. Amplification by shuffling: From local to central differential privacy via anonymity. In *SODA*, pages 2468–2479. SIAM, 2019.

[32] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, page 193–206, New York, NY, USA, 2002. Association for Computing Machinery.

[33] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194. ACM, 1987.

[34] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[35] Vipul Goyal and Ashutosh Kumar. Non-malleable secret sharing. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 685–698, 2018.

[36] Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In *CCS*, pages 1591–1601. ACM, 2016.

[37] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 91–107, Santa Clara, CA, March 2016. USENIX Association.

[38] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. Private anonymous data access. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 244–273, Cham, 2019. Springer International Publishing.

[39] Avinatan Hassidim, Haim Kaplan, Yishay Mansour, Yossi Matias, and Uri Stemmer. Adversarially robust streaming algorithms via differential privacy. *arXiv preprint arXiv:2004.05975*, 2020.

[40] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography from anonymity. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 239–248, 2006.

[41] Georgios Kellaris, Stavros Papadopoulos, Xiaokui Xiao, and Dimitris Papadias. Differentially private event sequences over infinite streams. *Proc. VLDB Endow.*, 7(12):1155–1166, August 2014.

[42] Dogan Kesdogan, Jan Egner, and Roland Büschkes. Stop- and- go-mixes providing probabilistic anonymity in an open system. In David Aucsmith, editor, *Information Hiding*, pages 83–98, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[43] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. Cryptology ePrint Archive, Report 2021/345, 2021. https://eprint.iacr.org/2021/345.

[44] Anunay Kulshrestha and Jonathan Mayer. Identifying harmful media in end-to-end encrypted communication: Efficient private membership computation. In *USENIX Security Symposium*. USENIX Association, 2021.

[45] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, page 364, USA, 1997. IEEE Computer Society.

[46] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *Proceedings on Privacy Enhancing Technologies*, 2016(2):115–134, 2016.

[47] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 711–725, 2018.

[48] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *OSDI*, pages 571–586. USENIX Association, 2016.

[49] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao, editors, *Information Security*, pages 314–328, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[50] Helger Lipmaa. First cpir protocol with data-dependent computation. In *Proceedings of the 12th International Conference on Information Security and Cryptology*, ICISC'09, page 193–210, Berlin, Heidelberg, 2009. Springer-Verlag.

[51] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security*, pages 168–186, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[52] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Maliciously secure multi-client oram. In *International Conference on Applied Cryptography and Network Security*, pages 645–664. Springer, 2017.

[53] Nick Mathewson and Roger Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In David Martin and Andrei Serjantov, editors, *Privacy Enhancing Technologies*, pages 17–34, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[54] Sahar Mazloom and S. Dov Gordon. Secure computation with differentially private access patterns. In *CCS*, pages 490–507. ACM, 2018.

[55] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[56] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. Pir-tor: Scalable anonymous communication using private information retrieval. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 31, USA, 2011. USENIX Association.

[57] Femi Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In George Danezis, editor, *Financial Cryptography and Data Security*, pages 158–172, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[58] Rahul Parhi, Michael Schliep, and Nicholas Hopper. Mp3: A more efficient private presence protocol. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security*, pages 38–57, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg.

[59] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1002–1019, New York, NY, USA, 2018. Association for Computing Machinery.

[60] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[61] Raphael R Toledo, George Danezis, and Ian Goldberg. Lower-cost ϵ-private information retrieval. *Proceedings on Privacy Enhancing Technologies*, 2016(4):184–201, 2016.

[62] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *IEEE Data Eng. Bull.*, 43(2):95–107, 2020.

[63] Carmela Troncoso, Mathias Payer, Jean-Pierre Hubaux, Marcel Salathé, James R. Larus, Wouter Lueks, Theresa Stadler, Apostolos Pyrgelis, Daniele Antonioli, Ludovic Barman, Sylvain Chatel, Kenneth G. Paterson, Srdjan Capkun, David A. Basin, Jan Beutel, Dennis Jackson, Marc Roeschlin, Patrick Leu, Bart Preneel, Nigel P. Smart, Aysajan Abidin, Seda Gurses, Michael Veale, Cas Cremers, Michael Backes, Nils Ole Tippenhauer, Reuben Binns, Ciro Cattuto, Alain Barrat, Dario Fiore, Manuel Barbosa, Rui Oliveira, and José Pereira. Decentralized privacy-preserving proximity tracing. *IEEE Data Eng. Bull.*, 43(2):36–66, 2020.

[64] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440, 2017.

[65] Salil Vadhan. *The Complexity of Differential Privacy*, pages 347–450. Springer International Publishing, Cham, 2017.

[66] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 137–152, New York, NY, USA, 2015. Association for Computing Machinery.

[67] Sameer Wagh, Paul Cuff, and Prateek Mittal. Differentially private oblivious ram. *Proceedings on Privacy Enhancing Technologies*, 2018(4):64–84, 2018.

[68] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 299–313, Boston, MA, March 2017. USENIX Association.

[69] David J Wu, Joe Zimmerman, Jérémy Planul, and John C Mitchell. Privacy-preserving shortest path computation. *arXiv preprint arXiv:1601.02281*, 2016.

[70] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and rdma-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, 2019.

A Simulator Construction

Input: $T = K; V$, and ϵ, δ.
Simulating the Offline Stage: The offline stage has no inputs on the client side, and only needs access to T, ϵ, and δ on the server side. The simulator can simulate this stage perfectly by running our protocol when simulating honest parties, and invoking the adversary for corrupted ones.
Simulating Client Online Queries: The simulator uses "junk" queries for this simulation. The actual queries are injected by the simulator later during the query phase.

1. The simulator assigns random query values to each honest client in its head. The simulator then runs our client protocol for these input query values, providing each client with the anonymous secrets the simulator selected when simulating that client's offline phase.

2. The simulator runs the adversary's code to determine the query message of each corrupted client.

Simulating Server Online Protocol - First Pass: The simulator goes through the servers in order, from s_1 to s_{m-1}.

1. **If s_i is corrupted**: The simulator runs the adversary on the query vector constructed by the previous step, which outputs the next query vector.

2. **If s_i is the first non-corrupted server**:

   - **Neither s_1 nor the backend are corrupted**: The simulator executes step 3 below.
   - **If s_1 is corrupted**: The simulator begins by identifying any mishandled honest client queries in the current query vector. For each honest client query, the simulator looks for it by its tag, which the simulator knows because she simulated the offline stage of

that client. The simulator validates that the associated tally has the expected value, furthermore, it checks that the anonymous secret installed at s_i during the offline stage match the ones the simulator generated when simulating the client portion of that offline stage. All of these checks depend on the honest client and honest server s_i offline state, which the simulator knows.
If any of tags, tallies, or shares do not match their expected value, or are missing, then the simulator knows that the adversary has mishandled this client's query (or corresponding offline stage) prior to server s_i. The simulator sends the identities of all such clients to the ideal functionality (step 1 in $\mathcal{F}$).

- **If backend is corrupted**: The simulator receives a noised histogram H_{honest} from the ideal functionality. The simulator identifies all honest queries that have not been mishandled so far. Say there are k such queries. As part of simulating s_i, the simulator will replace the tallies of these queries with new tallies, such that the tally of honest query $w \leq k$ would reconstruct to the value of the w-th entry in H_{honest}, when combined with the remaining shares that the simulator generated for that client during its offline stage.
Furthermore, the simulator needs to inject noise queries for s_i. The simulator chooses the tallies for these queries so they reconstruct to the remaining values in H_{honest}. This guarantees that all correctly handled honest client queries combined with this server's noise have the distribution H_{honest}.
The simulator shuffles the updated query vector and uses it as the output query vector for this server.

3. **If Neither Above Cases are True**: The simulator executes our protocol honestly, including using the same noise queries from the offline stage, to produce the next query vector.

Simulating The Backend: The simulator executes our protocol truthfully, if the backend is not corrupted, or runs the adversary's code if the backend is corrupted, and finds the next response vector.
Simulating Server Online Protocol - Second Pass: The simulator goes through the servers in reverse order, from s_{m-1} to s_1.

1. **If s_i is corrupted**: The simulator runs the adversary on the current response vector, outputting the next response vector.

2. **If s_i is the first encountered non-corrupted server**:

   - **If the backend is corrupted**: The simulator identifies all responses corresponding to honest queries that were misshandled. The responses do not have tags directly embedded in them. However, they should be in the same order as the queries at s_i, which do have these tags. Furthermore, the correct value of the response is know to the simulator, since she can compute it using the T, the value of the corresponding query, and the additive pre-share installed during the offline stage.
   The server sends a histogram over the count of these mishandled responses to the ideal function, grouped by their corresponding query value (step 3 in $\mathcal{F}$).
   - **If the backend is not corrupted**: The simulator executes step 3 below.

3. **If s_i is not corrupted**: The simulator identifies all honest queries that were mishandled, using the same mechansim as above. The simulator ignores mishandled queries that were detected in either of the two cases above (the special cases of the first server or backend being corrupted). The simulator only needs to keep count of such mishandled query.
   If s_i is the last honest server, she sends this count to the ideal functionality (step 4 in $\mathcal{F}$).

Simulating Client Online Responses: For every honest client, the simulator checks that her corresponding response, as outputted by s_1, reconstructs to the expected response value. If the response does not match, then it could have been mishandled by the adversary earlier, and have been already identified by the simulator, such responses are ignored.
The remaining mishandled responses must have been mishandled after the last honest server was simulated. The simulator sends a list of identities of all clients with such responses to the ideal function (step 5 in $\mathcal{F}$).

B Proof of Theorem 1

Proof. The view of the adversary consists of all outgoing and incoming messages from an to adversary corrupted parties. We show that these messages are indistinguishable in the real protocol from their simulator-generated counterparts.

First, note that all output messages from honest clients in the offline stage are cipher of random values. This is true in both the real and ideal world, and thus these messages are statistically indistinguishable. The same is true for messages corresponding to noise anonymous secrets created by an honest server. The adversary only receives such messages in the offline protocol, and therefore behaves identically in both real and ideal worlds.

Case 1: The backend server s_m is honest.

1. The access patterns are not part of the view, and therefore do not need to be simulated.

2. The corrupted clients are simulated perfectly and has identical outgoing message distributions in the real and ideal worlds.

3. The honest clients are choosing their queries randomly in the ideal world. However, their messages only include a tag and a tally. The tag is itself selected randomly during the offline stage, and thus has identical distribution. The tally is indistinguishable from random, regardless of the query it is based on, provided that at least one secret share remains unknown, by secrecy of our incremental sharing scheme. In particular, the honest server share is computationally indistinguishable to the adversary from any other possible share value, by CPA security of the onion encryption scheme.

4. The input messages of the first malicious server have indistinguishable distributions in the real and ideal worlds, and therefore the outgoing messages of that malicious server has indistinguishable distributions, since any honest servers prior to this malicious server are simulated according to the protocol perfectly. Inductively, this shows that all malicious servers have indistinguishable distributions during the first pass of the online stage.

5. The backend executes the honest protocol in both worlds. While the backend sees different distributions in either worlds, since honest clients make random queries when simulated, the

honest protocol is not dependent on that distribution, and only output responses in the form of secret shares. These secret shares are selected at random during the offline stage by the client, without knowing the response or the query. Therefore, the output of the backend is indistinguishable in both worlds.

6. Finally, a similar argument shows that the adversary input and output response vectors are all indistinguishable from random in both worlds, since the last secret share of honest queries remains unknown.

Case 2: The backend server s_m is corrupted.

1. The access patterns are part of the view, the simulator must yield a view consistent with them.

2. The outgoing messages of each corrupted client has identical distributions in the real and ideal worlds.

3. The honest client queries are selected randomly. However, they are secret shared. Their secret share component (tally) is indistinguishable from random in both worlds, given that the honest server share is unknown to the adversary. Therefore, their initial tallies are also indistinguishable (but not the access patterns they induce).

4. The input vector to the first server has identical distributions in both worlds, if that server is malicious, then its output vector will also have identical distributions. This argument can be applied to all malicious servers up to the first honest server.

5. The first honest server retains all queries from malicious servers and clients, and handles them as our honest protocol would. However, the server discards all client noise and injects its own queries into it from the provided $\mathcal{H}$. However, this is indistinguishable to the following server from the case where these queries are handled truthfully: (1) the tag component of the query is handled honestly (2) the tally component of the honest client queries are the result of an incremental reconstruction in our protocol, since the server's share being reconstructed is unknown, the output of this operation is indistinguishable from random even knowing the input. (3) the total count of queries induced by $\mathcal{H}$ is exactly the count of honest client queries that this server discards, plus an amount of noise queries sampled according to the honest noise distribution, this count has the same distribution as the count induced by the honest protocol.

6. The output of the first honest server is indistinguishable, and all the remaining servers are simulated truthfully, therefore their outputs are also indistinguishable., up to the backend.

7. The backend server is corrupted, and can reconstruct the access patterns from the input. However, these access patterns are now indistinguishable between the two worlds, this is because the access patterns of the secret shared queries as outputted by the first honest server in both worlds are indistinguishable: they are both equal to $\mathcal{H}$ + malicious clients and servers queries + mishandled queries. The mishandled queries are guaranteed to reconstruct to random, by our incremental secret sharing non-malleability property, even when their original queries are different (random in the simulated world).

8. The same argument from Case 1 demonstrates that the view from the second pass of the online stage is indistinguishable in both worlds.

The only thing that remains is to show that the interactions of the simulator and adversary with the ideal function $\mathcal{F}$ are indistinguishable. There are at most 4 such interactions. All of these interactions depend on the simulators ability to detect when a query or response has been mishandled.

A query may be mishandled by (1) corrupting its tag (2) corrupting its tally by setting it to a value different than the one determined by the associated offline anonymous secrets. Both of these cases can be checked by the simulator, since she has access to the expected uncorrupted anonymous secret values created by every honest client and server. Either of these cases result in the query reconstructing to random, the second case follows from our non-malleability property, the first induces the following honest server to apply an incorrect share when incrementally reconstructing, and thus follows from our non-malleability property as well.

On the other hand, a response can be mishandled by (1) corrupting its tally/value (2) corrupting its relative order within a response vector. The first case arises when an adversary sets the tally value to one different than the sum of its previous value and additive pre-share from its corresponding anonymous secret, as well as when a backend server disregards the underlying database, and assigns a different initial value to a given response. The second case happens when the adversary does not deshuffle responses with the inverse order of the corresponding shuffle. The simulator can check these two cases as well: if a deshuffle was performed correctly, then every response and query at the same index must correspond to one another, and the simulator can compute the expected value of that response from its query value, database T, and additive pre-shares. If the response and query did not match, then either the shuffling or tally computation was corrupted.

We can consider consecutive servers that are adversarially controlled to be a single logical server, since they can share their state and coordinate without restrictions. For example if the first and second server are corrupted, the second server can identify the identities of clients of corresponding to each of its input queries, because the first server can reveal its shuffling order to the second. Similarly with the backend and previous server. This shows that the correct points to check for mishandling is when an honest server is encountered, rather than after every malicious server, since consecutive servers may perform operations that each appear to be mishandling, but consecutively end up handling queries and responses correctly.

Our simulator does the mishandling checks at the level of an honest server. Furthermore, the simulator assumes that any mishandling was done according to the strongest identification method available to the adversary at that point. For example, it assumes that the first server always mishandles queries based on their clients identities, even though that server may mishandle queries randomly. In either cases these result in indistinguishable distributions. An adversary that mishandles queries randomly has the same distribution as a simulator that copies that random choice and translates it to identities. No server has the capability to mishandle based on both identity and value, since there must be at least one honest server somewhere between the backend and first server (including either of them).

Finally, intermediate servers (those surrounded by honest servers on both ends) see only query and response vectors that have been shuffled honestly by at least one server, and have a random share applied to their tally by that server as well. So their inputs are indistinguishable from random, and thus they can only mishandle randomly. The first server (and its adjacent servers) see query and response vectors whose tallies are random (because at least one share corresponding to them

is unknown), but have a fixed order, since no shuffling has yet occurred, therefore they mishandle queries based on the order (i.e. client identity) as well as randomly. Lastly, the backend (and its adjacent servers) see queries and responses that have been shuffled by at least one honest server, but whose values are revealed, since no shares of these values are unknown. The backend can mishandle queries based on their known value, but not based on their client identity, since mishandling based on index/order is identical to mishandling randomly, because the order is random.

□

C Analysis of the Noised Histogram Release

Theorem 2 (Leakage is Diferentially Private)**.** $\mathcal{H} = H_{honest} + \chi(\epsilon, \delta)$ *is* (ϵ, δ)*-Differentially Private.*

Proof. We define neighboring histograms over access patterns to differ in exactly one query, therefore the sensitivity is 2, since removing a query from a bin in the histogram dictates adding it back to a different bin. Hence adding noise from $Laplace_{0,2/\epsilon}$ constitutes an $(\epsilon, 0)$-differentially private histogram release mechanism, this corresponds to value l_i in our mechanism from algorithm 6.

Our mechanism selects B such that $Prob[l_i \leq -B] = Prob[l_i \geq B] = \frac{\delta}{2}$. Note that $l'_i \neq l_i + B$ iff either of these disjoint cases is true, so $Prob[l'_i \neq l_i + B] = \delta$. This implies that using l'_i constitutes an ϵ, δ-differentially private mechanism.

Finally, taking the floor of l'_i is equivalent to taking the floor of $l'_i + c$, where c is the true count of honest queries, since c is guaranteed to be integer. Therefore, floor maintains differential privacy by post-processing. □

D Proof of the Incremental Non-Malleability Property

Theorem 3 (Incremental Non-malleability of Our Scheme)**.** *The real and ideal dealers from the incremental non-malleability security game are statistically indistinguishable, when the game is instantiated with our secret sharing scheme.*

Proof. The dealer possesses $h = (x, y)$ and the incremental reconstruction function is $R(l, (x, y)) = y \times l + x \ mod \ z$.

Our aim is to show that for all $l,\ l',\ x,\ y$ and random r, the following two quantities are statistically indistinguishable:

$$l,\ l',\ R(l,(x,y)), R(l',(x,y)) \ \equiv \ l,\ l',\ R(l,(x,y)),\ r$$

Let us denote $g = y \times l + x$. We want to find x', y', such that

$$x' \times l + y' = g \ mod \ z$$
$$x' \times l' + y' = r \ mod \ z$$

Since z is prime and $l \neq l'$, we know there exists exactly one solution:

$$x' = (r - g) \times (l' - l)^{-1} \ mod \ z$$
$$y' = g - (r - g) \times (l' - l)^{-1} \times t \ mod \ z$$

Therefore, even with l, l' chosen by the adversary, any execution of the ideal dealer corresponds to a single execution of the real dealer, and vice-versa. Therefore, the adversary cannot distinguish between the real and ideal worlds. □