Batched Differentially Private Information Retrieval

Kinan Dak Albab* *Brown University*
Rawane Issa* *Boston University*
Mayank Varia *Boston University*
Kalman Graffi *Honda Research Institute Europe*

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 an offline 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 differentially private leakage in order to provide better trade-offs between privacy and efficiency. Our protocol achieves speedups up to and exceeding 10x in practical settings compared to state of the art PIR protocols, and can scale to batches with hundreds of millions of queries 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) [29,49] 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 [29], and computational PIR using single server [49]. 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, or else the server learns information about the query, namely what the query is not!

Recent PIR protocols [31, 47, 57, 68] achieve sub-linear computation and communication by relying on a preprocessing/offline stage that shifts the bulk of computation into off-peak hours [11], relaxing security to allow limited leakage [72], or batching queries, mostly in the case when they originate from the **same** client. These advances allowed PIR to be used in a variety of applications including private presence discovery [17, 67], anonymous communication and messaging [7, 26, 50, 63], private media and advertisement consumption [41, 42], certificate transparency [57], and privacy preserving route recommendation [80].

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 [47]) of handling each query quickly adds up when the number of queries approaches 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 assumes the number of queries is much smaller than the database size [57], burdens clients with making noise queries [72], or requires clients to closely coordinate and share secrets when preprocessing is used [11]. 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:

*K. Dak Albab and R. Issa contributed equally to this work.

1. Offloading public key operations to an offline stage so that the online stage consists only of cheap operations [31,68].

2. High throughput batched shuffling of messages by mixnets and secure messaging systems [53,54,75,77].

3. Relaxing the security of oblivious data structures and protocols to differentially private leakage [61].

DP-PIR Overview Our protocol is a batched 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 $O(n+q)$ 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) [35]. 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 generated noise queries similar to Vuvuzela [77]. 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 distribution can be configured to provide privacy at the level of a single query or all queries made by the same client in a single batch or over a period of time. The number of these noise queries is linear in n and $\frac{1}{\varepsilon}$ 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 provides an interpretation of what this differentially oblivious [23] access pattern privacy means (as compared to traditional PIR).

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 that this secret sharing scheme, which we define in section 4, is 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 a novel 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 to 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 hundreds of millions queries, while achieving throughput several fold higher than existing state of the art protocols. 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 [7,41,66,80]. However, they are impractical for a large class of applications with a large number of queries.

Motivating example One example that we consider throughout this work is checking for software updates on mobile app stores. The Google Play and iOS app stores contain an estimated 2.56 and 1.85 million applications each [44], and the number of active Android and iOS devices exceed 3 and 1.65 billion, respectively [32]. These devices perform periodic background checks to ensure that their installed applications are up to date. Currently, these checks are done without privacy: the 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. If the installed application is out of date, the device can then download the updated application via some anonymous channel, such as Tor. However, unlike DP-PIR, existing PIR protocols cannot scale to such loads, where the number of devices is about 1000x larger than the size of the database, each with tens of applications installed, given how quickly the sub-linear overheads per query add up. We demonstrate this empirically with three state of the art PIR protocols: Checklist [47], DPF [18], and SealPIR [6].

Additional Applications We believe that a large class of applications demonstrate similar properties ideal for DP-PIR. In privacy-preserving automated exposure notification for contact tracing [22, 73, 74], 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, identifying misinformation in end-to-end encrypted messaging systems [48] usually involves a denylist far smaller than the total number of messages exchanged in the system within a reasonable batching time window.

Our protocol relies on having two or more non-colluding parties that together constitute the service provider. This is a common assumption used by many other PIR protocols. Secure multiparty computation (MPC) has been applied in many real world applications over the last decade. This includes services federated over somewhat-independent subdivisions within the same large organization [1, 71], or additional parties that volunteer to participate to promote common social good [30, 69]. A third category, which we believe is most suited for the app store example, involves providers actively seeking out third parties to federate their services [13, 52] under contractual agreements for privacy or compliance reasons, usually in exchange for financial or reputation incentives. This has spurred various startups [64, 65] that provide their participation in secure multiparty computations as a service.

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. Such as applications where 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]. In practice, we emphasize that our relaxed DP 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.

Comparison to 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 [10], while computational PIR traditionally uses a single database and relies on cryptographic hardness assumptions [20, 28, 55].

Protocol	Computation		Communication	
	Online	Offline	Online	Offline
BIM04 [11]	$n^{0.55}$	–	$n^{0.55}$	–
CK20 [31]	$\sqrt{n}$	n	$\lambda^2 \log n$	$\sqrt{n}$
Checklist [47]	$\sqrt{n}$	n	$\lambda \log n$	$\sqrt{n}$
Naive †	n	-	n/q^*	-
PSIR [68] †	$q^* n$	n	$\log^c n$	n/q
CK20 [31] †	$q^*\sqrt{n}$	n	$\sqrt{n}$	$\sqrt{n}/q^*$
BIM04 [11] ‡§	$qn^{\frac{w}{3}}$	–	$n^{\frac{1}{3}}/q$	–
LG15 [57] ‡¶	$q^{0.8}n$	–	$\sqrt{n}$	–
This work ‡‖	$c_{\varepsilon,\delta}n+q$	$c_{\varepsilon,\delta}n+q$	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.
¶: up to $q = \sqrt{n}$.
‖: 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

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 [56].

Modern PIR protocols commonly introduce an offline preprocessing stage, which either encodes the database for faster online processing using replication [11, 15, 31, 47] and coding theory [19, 21, 43, 68], performs a linear amount of offline work per client to make the online stage sub-linear [21, 31, 47, 47], or performs expensive public key operations so that the online stage only consists of cheaper ones [31, 47, 68]. Other protocols rely on homomorphic primitives during online processing [3, 6, 79].

Finally, some protocols allow batching queries to amortize costs. When combined with preprocessing, batching is only supported for queries originating from the same client [31, 47, 68], or ones that share secret state [11]. Batching queries from different clients without preprocessing is possible [45] but has limitations. Earlier work induces a sublinear (but non constant) amortized computation complexity [11, 57]. Our work amortizes the computation costs of queries made by different queries down to a constant, while also requiring constant client work. In section 8, we discuss ε-PIR [72] which also amortizes such queries but burdens clients with generating the noise queries required for differential privacy.

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

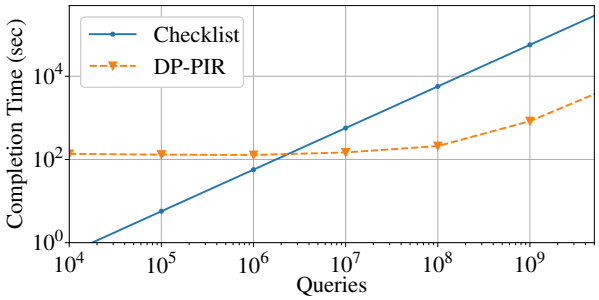


Figure 1: Checklist and DP-PIR Total completion time (y-axis, logscale) for varying number of queries (x-axis, logscale) against a 2.5M database

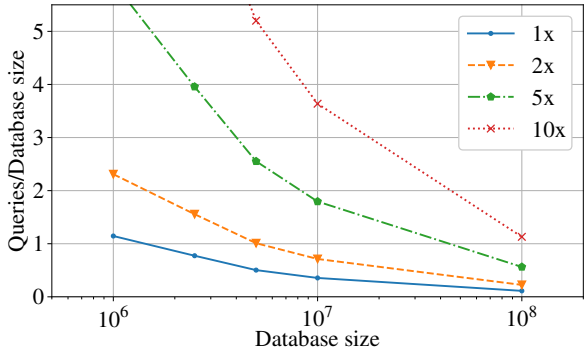


Figure 2: The ratios of queries/database (y-axis) after which DP-PIR outperforms Checklist by the indicated x factor for different database sizes (x-axis, logscale)

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 $n = 2.5M$ elements. Our protocol has constant performance initially, which starts to increase with the number of queries q as they exceed 10M. In more detail, the computation time of DP-PIR is proportional to the total count of noise and real queries $c_{\varepsilon,\delta}n + q$, where $c_{0.1,10^{-6}} = 276$. Therefore, the cost induced by q is negligible compared to $c_{\varepsilon,\delta}n$ 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 $q = 1.9M$ (slightly below $\frac{4}{5}$ the size of the database), 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 ∞. For a database with 2.5M elements, our experiments demonstrate that we outperform Checklist by at least 2x, 5x, and 10x after the ratio exceeds $1.5, 3.9$ and 8.1 respectively. We note that the largest data-point in the two figures are extrapolated.

The ratio required for achieving a particular speedup is not identical for all database sizes. As shown in Figure 2, 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 $q \to \infty$.

We extrapolate from our empirical results to three possible scenarios for our Google Play store example, where the database contains roughly $2.5M$ elements with $3B$ active users, with the same setup and parameters as above. First, we assume each user makes exactly a single query (corresponding to a single app on their phone) resulting in a batch of size $q = 3B$, and $\frac{q}{n} = 1200$. In the second scenario, we assume each user checks the updates for all apps on their phone (e.g. say at most 100 apps), but only configure our system to provide DP guarantees only at the level of a single query (i.e. event-DP). In the last scenario, each user similarly makes 100 queries, but we configure our system to provide user-level DP guarantees protecting all the queries of the same user (i.e. user-DP), which results in adding 100 times the amount of noise. Our estimates indicate that our protocol will exhibit speedups of 161x, 180x, and 161x over checklist in these scenarios respectively. We discuss the different DP configurations in section 3. We exhibit similar trends with larger speedups given even less queries over SealPIR [6] and DPF [18], as shown in appendix A.

3 Protocol Overview

Our protocol consists of $c_1, \ldots, c_d$ clients and $s_1, \ldots, s_m$ servers. We designate s_1 and s_m as a special *frontend* and *backend* server respectively. We assume that every server s_i has a public encryption key pk_i known to all servers and clients, with associated secret key sk_i. Every server has a copy of the underlying database $T = K \to (V, \Sigma)$ mapping keys to values and signatures, such that $T[k] = (v, \sigma)$, where σ is a $(m,m)-$threshold signature over (k,v) by the m servers. The signatures are only needed for integrity and do not affect the privacy of clients; they allow clients to verify that the responses they received correspond to the correct T agreed upon by the servers, and can be omitted when the backend is assumed to be semi-honest. We refer to the query made by client c_i by q^i, and its associated response by $r^i = (v^i, \sigma^i)$.

3.1 Setting

Our protocol is easiest to understand in the case of a single epoch consisting of an input-independent offline stage followed by an online stage. 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). Our offline stage is similar to Vuvuzela [77], which exhibits good throughput. However, 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.

In practice, services using DP-PIR alternate between collecting a batch of queries submitted from clients within some configurable time window, and processing that batch using our online protocol. In section 7, we discuss the effects of this batching window on our performance. Each batch requires corresponding offline processing. Our protocol allows multiple offline stages (e.g. the ones corresponding to an entire day's worth of batches) to be pooled together into a proportionally larger stage executed in one shot during off-peak hours when resources are cheaper (e.g. the night before). The clients can choose to make their queries at any time after preprocessing, but client states from several uncombined offline stages should not be used in a single online batch, to avoid allowing the adversary to identify the origin of the query by diffing out clients that participated in different stages.

Our protocol assumes that T and its signatures are provided as input. Thus, the servers must agree on T and produce signatures for it ahead of time. The same T and signatures can be reused by many offline/online stages; servers need only compute new signatures when the underlying database changes, and may rely on timestamps to enable clients to reject expired responses. The servers never sign or verify any signatures during either the offline or online stages, and each client needs to verify one signature per received response. Therefore, the efficiency of signing/verification is secondary. Instead, our protocol prefers signature schemes that produce shorter signatures for lower bandwidth.

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.

In terms of *confidentiality*, our protocol differs from traditional perfectly-private PIR protocols in that it leaks noisy access patterns over the honest clients' queries, in the form of a differentially private noisy histogram $\mathcal{H}(Q) = H_{\text{honest}}(Q) + \chi(\varepsilon, \delta, \phi)$.

As for *integrity*, 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: any server 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 Privacy

Our protocol can be configured to provide different levels of (ε, δ)-differential privacy by selecting the parameters of the underlying distribution used to sample noise queries. The most efficient (and easiest to understand) configuration is often called *event-DP*, which provides guarantees at the level of any **single** honest query. Another DP configuration, commonly termed *user-DP*, provides guarantees at the level of **all** queries made by any honest client. We use event-DP throughout the paper except when otherwise noted.

We provide either guarantee at the level of a single isolated batch. In particular, we consider two batches of queries Q and Q' over the honest clients' queries to be ϕ-neighboring batches when they consists of identical queries except for ϕ queries. In event-DP, it is enough to consider $\phi = 1$. While in user-DP, we set ϕ to the number of queries a client can make within a batch (or an upper bound of it). In either case, the sensitivity is 2ϕ, which means that for the same ε, δ the expected number of noise queries we add grows linearly in ϕ.

Definition 1 (Differentially Private PIR Access Patterns)**.** *For any privacy parameters ε, δ, and every two $\phi-$neighboring batches of queries Q, Q', the probabilities of our protocol producing identical access pattern histograms are (ε, δ)-similar when run on either set:*

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

Our definition uses the substitution formulation of DP, rather than the more common addition/removal; see [76, §1.6] for details. Substitution is commonly used in secure computation protocols involving DP leakage [61]. 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. Substitution is more conservative adding twice the expected amount of noise queries, since its sensitivity is 2ϕ compared to ϕ in the other.

So far, we only discussed guarantees within a single online stage. In any long running DP system where clients can make unbounded queries, it is impossible to achieve user-DP globally. Instead, practical systems [58] often rely on the user-time-DP model, where the guarantees extend over all queries made by a client over a set moving time window (e.g. a week). We can achieve this by setting ϕ to the number of queries that a client may make over a time window, regardless of how the client distributes the queries over the batches in that window. This follows from DP's composition theorem.

One way to interpret our DP guarantees (aka "differential obliviousness" [23]) is that they provide any client with plausible deniability: a client that made queries $q_1, ..., q_\phi$ over some period of time can claim that her true queries were any different $q'_1, ..., q'_\phi$, 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, differential obliviousness trades privacy for performance while always providing accurate outputs. In DP-PIR, increasing privacy (by lowering ε and δ or increasing ϕ) results in additional noise queries, making our protocol proportionally slower, and requiring a proportionally larger batch of queries to achieve the same amortization, and thus speedup, over other protocols. The amount of noise queries scales linearly in ϕ and $\frac{1}{\varepsilon}$ and sub-linearly in δ (see Table 3).

4 Incremental Non-Malleable Secret Sharing

Our protocol relies on shuffling real queries with noise queries by our chain of servers, similar to Vuvuzela and other mixnets where public key onion encryption is used to pass secrets through that chain. 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 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).

Formally, we define a secret sharing scheme with incremental reconstruction with the usual sharing mechanism but a new method to recover the original secret.

Definition 2. *An incremental secret sharing scheme S over a field $\mathbb{F}$ and m parties contains two algorithms.*

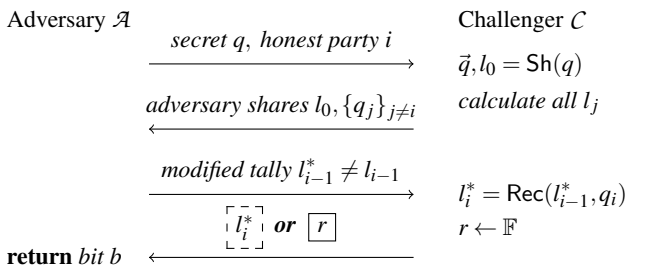
- $\mathsf{Sh}(q)$ *disperses a secret q into a randomly chosen set of shares $\vec{q} = q_1, \ldots, q_m \in \mathbb{F}$ and some initial tally l_0.*
- $\mathsf{Rec}(l_{i-1}, q_i) \to l_i$ *performs party i's partial reconstruction to produce running tally l_i.*

The scheme is correct if for all sharings $(\vec{q}, l_0) \leftarrow \mathsf{Sh}(q)$, the overall reconstruction returns $l_m = q$.

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.

We formally define non-malleability through the following indistinguishability game. It guarantees that if an adversarial set of $m-1$ parties submits a tampered partial tally l^*_{i-1} to the honest party i, then the tally l^*_i returned by the honest party is uniformly random. As a result, l^*_i is independent of (and therefore hides) the secret q, and it only completes to a reconstruction of q with probability $1/|\mathbb{F}|$.

Definition 3. *Consider the following two games that only differ in the final step. Call them* Left *and* Right *respectively.*



*We say that an **incremental** secret sharing scheme $S = (\mathsf{Sh}, \mathsf{Rec})$ is **non-malleable** if for all adversaries $\mathcal{A}$, the* Left *and* Right *games are (perfectly) indistinguishable.*

Several non-malleable secret sharing schemes exist [9, 40]. However, they are not incremental: their reconstruction is a one-shot operation over all shares. Conversely, known incremental schemes, such as additive or XOR-based sharing, are vulnerable to malleability. It would have been possible to use different primitives in our protocol that satisfy our desired 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 Incremental Sharing Construction Given a secret q, a prime modulus z, and an integer m, our scheme produces $m+$

Algorithm 1 Client i Offline Stage

Input: Nothing.

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

Output to s_1: Onion encryption of $a_1^i, ..., a_m^i$.

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^i. (2) Two incremental pre-shares $x_j^i \in \mathbb{Z}_z$ and $y_j^i \in \mathbb{Z}_z^*$. (3) An additive pre-share $e_j^i \in [0, 2^b)$. We define $e^i = \Sigma\, e_j^i \mod 2^b$ which the client uses to reconstruct the response online.

2. **Build Shared Anonymous secrets**: The client builds $a_j^i = (t_j^i, t_{j+1}^i, x_j^i, y_j^i, e_j^i)$, for every server $1 \le j \le m$, using the generated random values above, with $t_{m+1}^i = \perp$. 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 $OEnc_m^i = Enc(sk_m,\ a_m^i)$ and $OEnc_j^i = Enc(sk_j,\ a_j^i :: OEnc_{j+1}^i)$.

4. **Secrets Submission**: The client sends the onion cipher $OEnc_1^i$ to server s_1. The client can leave the protocol as soon as receipt of this message is acknowledged.

Algorithm 2 Server s_j Offline Stage

Configuration: The underlying database $T : K \to (V, \Sigma)$, and privacy parameters $\varepsilon, \delta, \phi$.

Input from s_{j-1} or clients if $j = 1$: A set of onion ciphers of anonymous secrets, one per each incoming request.

Output state at 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 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 $\mathcal{N} \leftarrow \chi(\varepsilon, \delta, \phi)$, 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 $s_{j'}$ with $j' > j$, using the same algorithm as the client. The server stores these secrets in L.

5. **Shuffling and Forwarding**: The server shuffles all onion ciphers, including all $OEnc_{j+1}^i$ decrypted in step (1) or generated by step (4), and sends them over to the next server s_{j+1}.

1 pairs $q_0 = (x_0, y_0), q_1 = (x_1, y_1), ..., q_m = (x_m, y_m)$, where each pair represents a single share of q. All x and y values 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:

$$x_0 = \langle [(q - x_m) \times y_m^{-1}] \ldots - x_1 \rangle \times y_1^{-1} \ mod\ z, \qquad y_0 = 0.$$

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 (32 bits in our experiments). To reconstruct the secret q, we show below the incremental reconstruction operations $\mathsf{Rec}(l_{i-1}, q_i)$ to construct the first partial tally and all subsequent ones:

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

Correctness (i.e., $l_m = q$) stems from our choice of (x_0, y_0). Refer to the full version of this work on ePrint [4] for the proof showing that our construction is non-malleable.

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. The server outputs a larger set. This set contains both the processed input messages, as well as new messages inserted by the server.

The client-side protocol is shown in algorithm 1. Concretely, for each server j, client i generates secret $a_j^i = (t_j^i, t_{j+1}^i, x_j^i, y_j^i, e_j^i)$, where t_j^i and t_{j+1}^i are random tags chosen from a sufficiently large domain that the client uses online to point each server to its secret without revealing its identity, x_j^i and y_j^i are secret shares from our incremental secret sharing

Algorithm 3 Client i Online Stage

Input: A query q^i.
Input state at the client: a shared anonymous secrets $a^i_j = (t^i_j, t^i_{j+1}, x^i_j, y^i_j, e^i_j)$ per server s_j generated by the offline stage.
Output: A value v^i corresponding to $T[q^i]$.

1. **Compute Final Incremental Secret Share**: Client computes $l^i_1 = x^i_0$, so that $(x^i_0, 0)$ combined with $(x^i_1, y^i_1), ..., (x^i_m, y^i_m)$ is a valid sharing of q^i, per our incremental secret sharing scheme.
2. **Query Submission**: Client sends (t^i_1, l^i_1) to server s_1.
3. **Response Reconstruction**: Client receives response r^i_1 from s_1 and reconstructs $(v^i, \sigma^i) = r^i_1 - e^i \mod 2^b$.
4. **Response verification**: The client ensures that σ^i is a valid signature over (q^i, r^i) by $s_1, ..., s_{m-1}$.

Algorithm 4 Server s_j Online Stage

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

1. **Anonymous Secret Lookup**: For every received query (t^i_j, l^i_j), 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 $(t^i_{j+1}, \mathsf{Rec}(l^i_j, (x^i_j, y^i_j)))$, where Rec is our scheme's incremental reconstruction function.

If $j < m$:

3. **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 $\mathcal{N}$.
4. **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.
5. **Response Handling:** Received responses corresponding to noise queries generated by this server are discarded. For every remaining received response r^i_{j+1}, the server computes the output response $r^i_j = r^i_{j+1} + e^i_j \mod 2^b$.
6. **Response Forwarding:** The server sends all output responses r^i_j to s_{j-1}, or the corresponding client c_i if $j = 1$.

If $j = m$:

7. **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 finds the corresponding $T[q^i] = (v^i, \sigma^i)$. If q^i was not found in the database (because a malicious party mishandled it), we return an arbitrary random value.
8. **Response Handling:** The backend computes responses $r^i_j = (v^i :: \sigma^i) + e^i_j \mod 2^b$, and sends them to s_{j-1}.

scheme used to reconstruct the query, and $\Sigma\, e^i_j \mod 2^b = e^i$ are additive secret shares used to mask the response. The exponent b corresponds to the bit size of values and signatures (instantiated to $32+384$ in our experiments). Our offline protocol uses *onion encryption* from CCA-secure public key encryption to pass secrets through the servers (here, :: denotes string concatenation):
$OEnc^i_1 = Enc(sk_1, a^i_1 :: Enc(sk_2, a^i_2 :: \ ... \ Enc(sk_m, a^i_m) \ ...))$

In addition to secrets from clients, each server must inject sufficiently many secrets at subsequent servers to handle all noise queries that the server needs to make in the online stage. This corresponds to steps 3 and 4 in algorithm 2, where the server computes the exact noise amount by pre-sampling.

The output set of each server 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 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, and since the output set is uniformly shuffled. This is true even if the adversary perturbs onion ciphers prior to passing them to an honest party (by CCA security), which in-essence denies service to the corresponding query.

Online Stage The client-side online protocol is shown in algorithm 3. The server-side online stage (algorithm 4) is structured similarly to the offline stage. However, it requires going through the chain of servers twice. The first phase (steps 1-4) moves from the clients to the backend server, where every server incrementally reconstructs the values of received queries using the stored secrets (steps 1-2), and injects its noise queries into the running set of queries (step 3). The second phase moves in the opposite direction (steps 5-6), with every server removing responses to their noise queries, and incrementally reconstructing the 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 (steps 7-8). It computes the reconstructed query set, and finds their corresponding responses in the 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.

The honest server shuffles and re-randomizes all its input messages, which guarantees that the adversary cannot link input and output 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 its 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 thus cannot observe the effects of a 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.

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, and 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 [77] and many other mixnet systems.

Algorithm 5 Ideal Functionality $\mathcal{F}$

Input: A set of queries q^i, one per client, the underlying database $T : K \to (V, \Sigma)$, and privacy parameters $\varepsilon, \delta, \phi$.
Output: A set of outputs v^i, one per client, either equal to the correct value 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 $\chi(\varepsilon, \delta, \phi)$.
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 v^i such that $T[q^i] = (v^i, \sigma^i)$ for every client i not excluded by any of the steps above.

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 a specific query based on at most one of its value or its origin client. A construction for the simulator and proof for Theorem 1 are available in the full version of this work [4].

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:*

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

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

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

Input: The size of the database $|T|$, privacy parameters ε, δ, and the number of protected queries ϕ.
Output: A histogram $\mathcal{N}$ over T representing how many noise queries must be issued for each database entry.

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

For every $i \in |T|$:

2. Sample ε-DP Laplace noise: $u_i \leftarrow Laplace(0, \frac{2\phi}{\varepsilon})$.
3. Clamp negative noise: $u'_i := max[0,\ B + min(B, u_i)])$.
4. $\mathcal{N}[i] = \text{floor}(u'_i)$

private, which entails adding noise to that histogram from a suitable distribution. Algorithm 6 shows the mechanism each server uses to sample the noise queries $\mathcal{N}$, and we prove that it indeed achieves (ε, δ)-differential privacy in our full paper [4]. Step (2) is a Laplace substitution $(\varepsilon, 0)$-DP histogram release, which may produce negative values. Step (3) ensures values are non-negative by clamping into $[-B, B]$ and shifting by B, where B is carefully selected in (1) to yield a privacy loss of exactly δ. Table 3 shows the expected number of noise queries per server and database element for different ε and δ.

6 Scaling and Parallelization

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 would need to add an independent set of noise queries, since each 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. 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 offline secret state and the noise queries they select.

A machine m_i^j belonging to party j 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. The machine also communicates with all other machines belonging to the same party j 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 [75]) 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 shuffle 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 shared seed locally to uniformly sample the same global permutation P using Knuth shuffle. Given a total batch of size l, 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 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. Maintaining a copy of all secrets in the main memory of each machine may be suitable for smaller applications. At larger scales, it may be more appropriate to use shared key-value storage or in-memory distributed file system [8, 51, 62, 81].

7 Evaluation

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 performing various computations from the protocol, as well as time spent waiting for network IO as messages get exchanged between servers. Our measurements do not include client processing or round-trip time.

All experiments in the paper use $\varepsilon = 0.1$ and $\delta = 10^{-6}$.

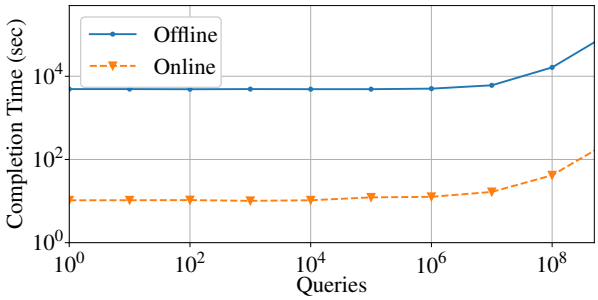


Figure 3: Completion time for varying number of queries against a 100K database (logscale)

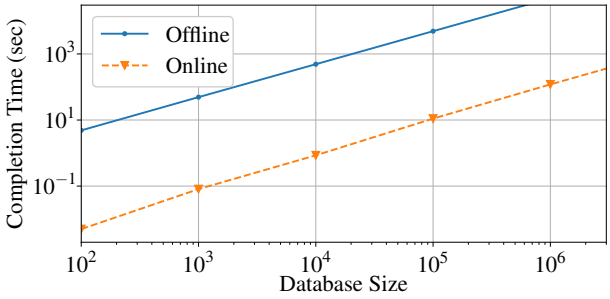


Figure 4: Completion time for a batch consisting only of noise queries against varying database sizes (logscale)

Keys and values in our database are each 4 bytes, with signatures that are 48 bytes long (e.g. BLS [16]). We ran our experiments on AWS r4.xlarge instances that cost around $0.25 per hour, using only one thread. A primary factor in selecting these instances is RAM, since we need sufficient memory to store large query batches. We implemented our protocol using a C++ prototype with about 6.1K lines of code. Our prototype relies on libsodium's `crypto_box_seal` [33] for encryption. Our code is available on GitHub [34].

Scaling Figures 3 and 4 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 a large enough number of queries, our runtime scales linearly as the overhead of noise queries is amortized away over the real queries. Our noise overhead scales linearly with the size of the database. The cost of processing any input query *in isolation (without noise)* 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 500x more expensive than our online stage. This is expected since the offline stage performs a public key operation for each corresponding modular online arithmetic operation.

Figure 5 shows how our protocol scales with the number

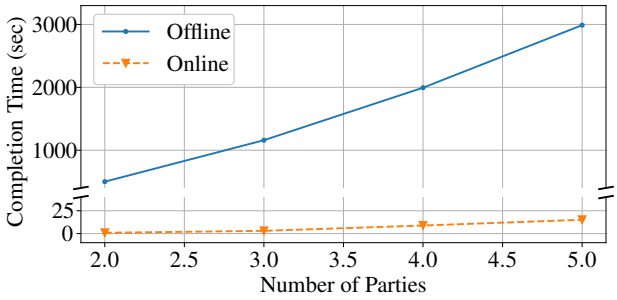


Figure 5: Completion time for varying number of parties with 100K queries against a 10K database

Machines / Party	Server time (seconds)	
	Offline	Online
1	5010	11
2	2560	8.2
4	1296	4.0
8	664	2.2

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

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, causing the offline stage to scale super-linearly in the number of parties. In addition, each server naively adds the full amount of noise queries required to independently tolerate up to $m-1$ corrupted parties. 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, and can have important consequences to mixnets, the DP shuffling model, and DP mechanisms in general. Techniques such as noise verification [53] 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. This means that the amount of noise overhead (and thus the number of queries required for that overhead to amortize effectively) grows linearly with $\frac{1}{\varepsilon}$. Our protocol trades security for performance. It can efficiently amortize the cost

δ \ ε	1	0.1	0.01	0.0001
10^{-5}	23	230	2302	23025
10^{-6}	27	276	2763	27631
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)

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 $\frac{1}{\varepsilon}$ 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 [75] or Geometric [61]), or by adapting recent work on privacy amplification [27, 36] that achieves the same level of privacy using less noise with oblivious shuffling.

Latency Latency in Checklist and similar systems includes the computation cost of a single query *in isolation* (which is low), and any *queuing delays* experienced by the query after its arrival if the computational resources are busy handling previous queries. This delay depends on the rate at which queries come in, and can be significantly larger than the batching overheads in applications with a large query load. In contrast, our protocol is primarily throughput oriented and its latency is a secondary concern determined by two components: the idle waiting time required to collect the batch of queries from different clients, which we call the *batching window*, and the active processing time of that batch after collection. The first component depends on the configuration. The later component is precisely the total computation time measured in the various experiments in earlier parts of the paper. Lowering the batching window beyond a certain point can have a negative impact on latency (and even throughput), since it can result in smaller batches dominated by noise where amortization is not effective. Furthermore, it can introduce queuing delays at the level of batches, where a previous ongoing batch still occupies system resources after the next batch has been collected.

We analyze DP-PIR's latency and the effects of the batching window in our full paper [4]. We summarize three important observations: (1) Queuing delays in existing systems are significant and can cause them to exhibit latency worse than DP-PIR with a large number of queries. (2) Both DP-PIR and existing systems can be scaled horizontally to exhibit lower latency. Traditional PIR protocols can achieve sub-second latencies if given enough resources, but this can be prohibitively expensive when the query rate is high. (3) For our target large query loads, DP-PIR can be configured to exhibit decent latency with a much lower budget than existing systems.

The Offline Stage PIR protocols with an offline stage typically do so to improve their online latency, which is less critical in our target applications. It is possible to combine both DP-PIR stages into a single stage that performs onion-encryption of the query directly, without the need to install anonymous secrets. This combined protocol would exhibit similar trends to our current design, but will be around two orders of magnitude slower than our online protocol on its own. A fair comparison here must also account for the offline cost of existing protocols, which can be significantly larger than our offline cost. For example, Checklist relies on an expensive per-client offline stage linear in the size of the database, which we observe takes up to 7 seconds per client in our experiments. In DP-PIR, the offline cost for a single query amortizes to a few milliseconds. One key difference is that a client can reuse the hint produced by Checklist's offline stage to make many following queries, rather than a fixed number of queries in DP-PIR. However, the hint becomes invalid whenever the database is updated. Checklist provides an updatable offline construction, where a single update to the database can be carried over to a previous offline computation in cost logarithmic in the database size.

We believe the that the offline-online design provides better deployment cost and performance, and allows DP-PIR to meet the availability and liveness requirements of many applications, including our App store example. Concretely, the offline-online design allows greater control over the batching window, which governs the effectiveness of amortization, client latency, and the duration needed for updates to the database to become visible to clients at the next batch. For example, it may be desirable to allow clients to query the App store multiple times a day, e.g. every hour, in order to discover important app updates earlier. A natural way to achieve this is to use a batching window of one hour or less. However, this is only effective if this window includes sufficient queries for amortization, and has sufficient time to complete processing before the next batch. The offline setup lowers both requirements, making smaller windows practical (or alternatively, cutting the online cost of the same window by 500x).

The offline stages for multiple online stages can be pooled together and executed ahead of time. Clients can choose to issue less queries than they signed up for in the pooled offline stage without privacy loss. Service providers can use this to execute the combined offline stages during off-peak hours when resources are cheaper (e.g. overnight). Furthermore, providers can use different setups for each stage to optimize the effectiveness of their budget. The offline stage is CPU-intensive due to its public key operations, while the online stage is entirely network bound.

8 Related Work

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

Mixnets Traditional mixnets [24] 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 [12, 37] add *cover traffic* to obfuscate various traffic patterns. However, ad-hoc cover traffic is shown to leak information over time [60].

Recent work mitigates this by relying on secure multiparty computation [5] or differential privacy. Vuvuzela [77] adds noise traffic from a suitable distribution to achieve formal differential privacy guarantees over leaked traffic patterns, and Stadium [75] improves on its performance by allowing parallel noise generation and permutation. Similar techniques have been used in private messaging systems [53], and in differential privacy models that utilize shuffling for privacy amplification [36] or for introducing a *shuffled* model that lies in between the central and the local models [27].

Differential Privacy and Access Patterns Using differential privacy to efficiently hide access patterns of various protocols has seen increasing interest in the literature. $\varepsilon-$PIR relaxes the security guarantees of PIR to be differentially private [72] in the semi-honest setting. Their two AS schemes are closest to our protocol: they require clients (rather than servers) to generate noise queries along with their real queries, and send all of them through an anonymous network for mixing. When the number of clients is large enough, this can amortize the number of queries any of them have to generate to a constant. However, this approach generates far more total load on the system. For example, in our app store example with 2 servers, a 2.5M database, and 3B clients, each client needs to generate 282 noise queries to hide a *single* query with $\varepsilon = 0.1$, which results in close to $850B$ queries to the system in total, compared to the $< 4B$ total load on our system (but with $\delta = 10^{-6} \neq 0$). These constructions do not provide integrity guarantees, and will require further noise queries to protect against potential malicious or unavailable clients.

Others relax the security of Oblivious RAM (ORAM), a primitive where a single client obliviously reads and writes to a private remote database [38, 39], to be differentially private. Extensions of ORAM address multi-client settings [59]. Differentially oblivious RAM [23, 78] guarantees that neighboring access patterns (those that differ in the location of a single access, i.e. event-DP) occur with similar probability. DP access patterns have been studied for searchable encryption [25] and generic secure computation [61].

Secret Sharing Shamir Secret Sharing [70] 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 [9, 40] 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. Aggarwal et al. [2] show generic transformations to build non-malleable schemes from secret sharing schemes over the same access structure.

9 Conclusion

This paper introduces a novel PIR protocol targeted exclusively at applications with high query rates relative to the database. This focus is intentional and necessary: DP-PIR handles large batches so well specifically because it handles small ones poorly. Our construction makes PIR usable in scenarios that were previously impractical or unexplored. DP-PIR is primarily geared towards amortizing total server work (i.e. throughput), but not for sub-second client latency, and only provides relaxed differential privacy guarantees.

The performance of DP-PIR is closely tied to its configurations, which determine the number of noise queries generated by our system, and thus the number of queries required to amortize their overheads effectively. Our experiments meet or extend beyond standard configurations suggested by existing work. Checklist [47] supports exactly two parties, and PIR schemes are rarely instantiated with more than three. For small databases (e.g. $n < 100K$), the naive solution of sending the entire DB to the client may be desirable. Vuvuzela [77] recommends $\varepsilon \in [0.1, \ln(3)]$ and sets $\delta = 10^{-4}$, and other work [61, 72] also mostly focuses on $\varepsilon \geq 0.1$.

The ratio of queries to database size $\frac{q}{n}$ is the primary performance criteria that governs how effective DP-PIR is compared to existing protocols. Within the space of *typical configurations* outlined above, our experiments demonstrate that applications with $\frac{q}{n} < \frac{1}{10}$ are unsuited for DP-PIR, while applications with $\frac{q}{n} > 10$ are almost always guaranteed to exhibit speedups of several folds when using DP-PIR. Applications with ratios in $[\frac{1}{10}, 10]$ may or may not be suited to DP-PIR, depending on their exact configurations. For example, we can achieve better performance than existing work for a ratio of 0.8 when the database size is $2.5M$, but not when it is of size $1M$ (section 2). Thus, such applications require individual analysis to determine the best way to realize them.

Our protocol shifts expensive public key operations to an offline stage. This allows for more flexibility over the batching window to meet application requirements, and a more efficient allocation of computational resources. However, applications were these factors are not a concern may elect to combine the two stages into a single one, that still exhibits similar trends to our online stage, but is about two orders of magnitude more expensive. Finally, these ratios, and the number of noise queries, also depend on the level (and duration) of protection offered to users (e.g. event-DP vs user-time-DP) as expressed by ϕ. DP-PIR intentionally relaxes its guarantees for increased performance. This relaxation becomes less meaningful as ε and ϕ approach perfect security.

Acknowledgments The authors are grateful to our helpful shepherd, Wouter Lueks, and the anoynmous reviewers. We are grateful to Andrei Lapets, Frederick Jansen, Jens Schmuedderich, Malte Schwarzkopf, Ran Canetti, and Adam Smith for their valuable feedback on various versions of this work. This material is supported by Honda Research Institutes, by the National Science Foundation under Grants No. 1414119, 1718135, and 1931714, by DARPA under Agreement No. HR00112020021, and by DARPA and the Naval Information Warfare Center (NIWC) under contract No. N66001-15-C-4071.

References

[1] ACDEB. Advisory committee on data for evidence building: Year 1 report. https://www.bea.gov/system/files/2021-10/acdeb-year-1-report.pdf, 2021.

[2] Divesh Aggarwal, Ivan Damgård, 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.

[3] 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.

[4] Kinan Dak Albab, Rawane Issa, Mayank Varia, and Kalman Graffi. Batched differentially private information retrieval. Cryptology ePrint Archive, Paper 2020/1596, 2020. https://eprint.iacr.org/2020/1596.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security*, pages 325–343, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[14] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *IEEE Symposium on Security and Privacy*, pages 762–776. IEEE, 2021.

[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] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[17] Nikita Borisov, George Danezis, and Ian Goldberg. DP5: a private presence service. *Proceedings on Privacy Enhancing Technologies*, 2015(2):4–24, 2015.

[18] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page

1292–1303, New York, NY, USA, 2016. Association for Computing Machinery.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In *SODA*, pages 2448–2467. SIAM, 2019.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] Henry Corrigan-Gibbs. Privacy-preserving telemetry in Firefox. Real World Crypto (RWC), 2020. https://rwc.iacr.org/2020/slides/Gibbs.pdf.

[31] 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.

[32] Alex Cranz. 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.

[33] Frank Denis. The sodium cryptography library, Jun 2013.

[34] DP-PIR GitHub repository. https://github.com/multiparty/DP-PIR/tree/usenix2022, 2022.

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

[36] 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.

[37] 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.

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

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

[40] 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.

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

[42] 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.

[43] 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.

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

[45] 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.

[46] Daniel Kales. Golang DPF library. https://github.com/dkales/dpf-go, 2021.

[47] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *USENIX Security Symposium*, pages 875–892. USENIX Association, 2021.

[48] 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.

[49] 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.

[50] 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.

[51] Michael Labib. 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.

[52] Andrei Lapets, Frederick Jansen, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, and Azer Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, COMPASS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

[58] Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. Privacy budget scheduling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 55–74, 2021.

[59] 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.

[60] 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.

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

[62] 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.

[63] 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.

[64] MPC Alliance. https://www.mpcalliance.org/, 2021.

[65] Nth Party. https://www.nthparty.com/, 2021.

[66] 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.

[67] 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.

[68] 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.

[69] Anjana Rajan, Lucy Qin, David W Archer, Dan Boneh, Tancrede Lepoint, and Mayank Varia. Callisto: A cryptographic approach to detecting serial perpetrators of sexual misconduct. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, pages 1–4, 2018.

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

[71] Stephanie Straus. A federal government privacy preserving technology demonstration, 2021. https://mccourt.georgetown.edu/news/a-federal-government-privacy-preserving-technology-demonstration/.

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

[73] 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.

[74] 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.

[75] 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.

[76] Salil Vadhan. The complexity of differential privacy. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 347–450. Springer International Publishing, Cham, 2017.

[77] 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.

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

[79] 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.

[80] David J. Wu, Joe Zimmerman, Jérémy Planul, and John C. Mitchell. Privacy-preserving shortest path computation. In *NDSS*. The Internet Society, 2016.

[81] 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.

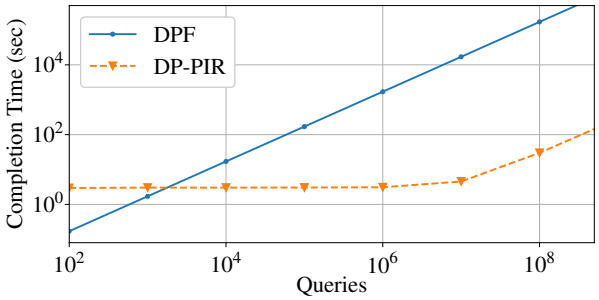


Figure 6: DPF and DP-PIR Total completion time (y-axis, logscale) for varying number of queries (x-axis, logscale) against a 2.5M database

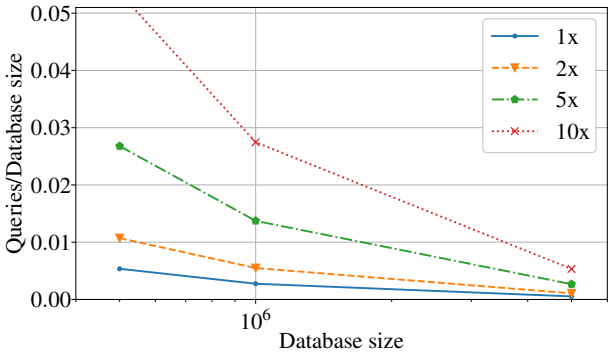


Figure 7: The ratios of queries/database (y-axis) after which DP-PIR outperforms DPF for different database sizes (x-axis, logscale)

A DPF and SealPIR

The setup and parameters in both comparisons below is identical to section 2.

DPF Boyle, Gilboa, and Ishai [18] propose a PIR protocol based on distributed point functions (DPF). Unlike the offline-online protocol introduced in Checklist that uses punctured pseudorandom sets, DPF requires linear work in the database size to handle user queries. However, DPF requires no offline preprocessing and significantly lower client computation and communication than checklist. We compare our system to the DPF implementation provided as an alternative backend for checklist based on the optimized implementation of Kales [46]. Figures 6 and 7 show our results. For a database with 100K elements, DPF outperforms DP-PIR when the number of queries is small relative to the size of the database. When the number of queries q approaches $31K$, with $\frac{n}{q} = 0.0124$, the two systems exhibit identical completion time, with DP-PIR significantly outperforming DPF as the number of queries grow beyond that.

SealPIR Figures 8 and 9 show similar results for SealPIR.

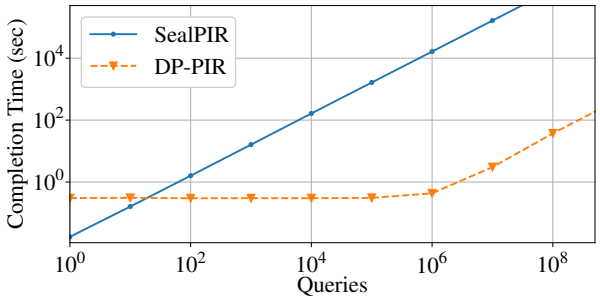


Figure 8: SealPIR and DP-PIR Total completion time (y-axis, logscale) for varying number of queries (x-axis, logscale) against a 10K database

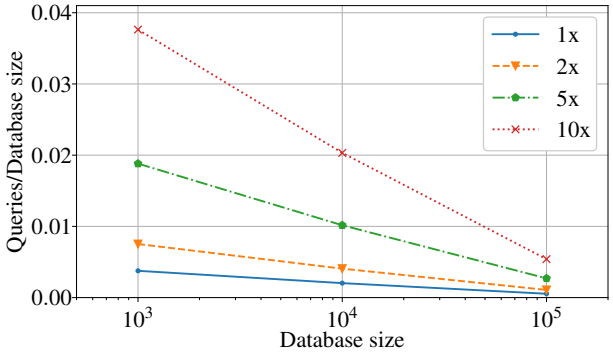


Figure 9: The ratios of queries/database (y-axis) after which DP-PIR outperforms SealPIR for different database sizes (x-axis, logscale)

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 32) with a ratio $\frac{q}{n}$ of just 0.003. Similarly, we achieve 2x, 5x, and 10x speedups for modest ratios all below 0.02. These ratios decrease as the database size grows, similar to our experiment with Checklist. We outperform SealPIR with far fewer queries than we do Checklist and DPF, in large part because SealPIR's uses expensive homomorphic operations during its online stage, while checklist offloads expensive linear work to an offline stage. Our protocol goes even further, only executing a couple of modular arithmetic operations per query online.