Batched Differentially Private Information Retrieval

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

Abstract

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

In this work, we present an efficient PIR protocol that combines all of the above techniques to achieve *constant* amortized communication and computation complexity in the size of the database, and is the first to scale to more than 10^5 queries per second deployed on an AWS micro instance. Our protocol also builds upon a new secret sharing scheme that is both incremental and non-malleable, which may be of interest to a wider audience. We leverage differentially private leakage in order to provide better trade-offs between privacy and efficiency. Our protocol provides security up to abort against malicious adversaries that can corrupt all but one party.

1 Introduction

Private Information Retrieval (PIR) [1,2] 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 is the foundation of many privacy preserving systems including private presence discovery [3, 4], anonymous communication and messaging [5–8], private media and advertisement consumption [9,10], certificate transparency [11], and privacy preserving route recommendation [12], among others.

PIR protocols have been developed for a variety of settings, including information theoretic PIR where the database is replicated across several servers [1], and computational PIR that can be performed with a single server [2]. The different settings of PIR are limited by various lower bounds on their computation or communication complexity. In essence, a server must "touch" every entry in the database when responding to a query, otherwise the server learns information about the query, namely what the query is not! Relying on cryptographic primitives to obfuscate the identity of these entries or the access patterns over them does not yield substantial improvements beyond the naive linear scan [13]. These limitations on earlier PIR protocols led researchers to believe that PIR had limited practical benefits due to its prohibitive performance costs [14].

Recent PIR protocols [15, 16] demonstrate that these limitations may be mitigated using novel techniques, including shifting the bulk of the computational work into off-peak hours and amortizing

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

costs over several queries. Other lines of work provide relaxed security definitions for PIR [17] and Oblivious RAM [18]. These new ideas demonstrate that PIR protocols can have either low enough latency or high enough throughput to use practically in the right scenario [19].

This work We construct a novel PIR protocol suitable for batch processing a very large number of queries with high throughput *and* low latency. While the details of our protocol are different from earlier PIR protocols, at a high level our construction combines:

- The fast (sub-linear, no public key crypto) online runtime of PSIR [15] and Corrigan-Gibbs and Kogan [16], and
- The high throughput batched shuffling of Vuvuzela and related works [20–23] achieving security up to differentially private leakage [18].

Our information-theoretic online design may be beneficial to the design of future mixnets and ORAMs. Also, our weaker notion of security still provides a rigorous guarantee: each individual client retains "plausible deniability" to equivocate on her own query, even if aggregate patterns are revealed.

In both the online and offline stages, our protocol amortizes work over all queries processed within a batch, including those made by different clients who are unaware of each other and need not perform any coordination. When the number of queries approaches or exceeds the size of the underlying database, the servers' computation and communication costs amortizes down to a constant. Additionally, the client work and storage is also constant relative to the size of the database. Our protocol supports two or more servers, and is secure against a dishonest malicious majority, up to selective aborts, relying on only standard cryptographic assumptions. Additionally, our protocol is embarrassingly parallel, and exhibits linear improvements in latency and throughput when parallelized over additional machines and resources. As a result, our protocol is most useful in high throughput and high demand applications where the number of queries made across all clients approaches or exceeds the size of the table, in order to make use of our amortization to its fullest extent.

A critical building block in the online protocol is a new information-theoretic secret sharing scheme that is both incremental and non-malleable: servers incrementally reconstruct a shared secret sequentially one server at a time, each combining one secret share with a running tally revealed to the next server, while adversaries perturbing any subset of secret share cannot cause reconstruction to output any specific related secret. This primitive yields a latency boost of over $25\times$ when compared to onion-encryption based techniques since it consists exclusively of cheap arithmetic operations. Additionally, we believe that this secret sharing scheme is of independent interest and may be an efficient alternative to heavier public and symmetric key authenticated onion encryption schemes in other applications like mixnets, moving target defense, verifiable secret sharing, and custom secure multiparty computation protocols.

Our Contribution We make three contributions in this work.

1. We introduce a novel PIR protocol that is the first to achieve constant amortized communication and computation complexity in the size of the database over queries made by different clients, in both its offline and online stages. Our offline stage performs public key operations linear in the size of the database and number of queries, and our online stage consists exclusively of cheap arithmetic operations.

2. To achieve a crypto-free online stage, we construct an efficient secret sharing scheme that is both incremental and non-malleable, based only on modular arithmetic for both sharing and its reconstruction. We believe this construction yields the simplest (and thus fastest) information-theoretic scheme that exhibits both properties combined.

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

Organization The rest of this paper is organized as follows: Section 2 discusses related work, Section 3 summarizes our protocol setting, thread model, and security guarantees. We describe our novel incremental non-malleable secret sharing scheme, and formalize its guarantees in Section 4. Our protocol, in both its online and offline stages, is described in detail in Section 5. We discuss scaling over several parallel machines in Section 6. We show our experimental results in Section 7 and discuss an application case study in section 8. Finally, we discuss future work and extensions in Section 9. The appendices contain various constructions and proofs demonstrating the security of our protocol.

2 Related Work

Private Information Retrieval Private Information Retrieval (PIR) has been been extensively studied since its inception [1, 2] in a variety of settings. Information theoretic PIR replicates the database over several non-colluding servers [24], while computational PIR traditionally uses a single database and relies on cryptographic hardness assumptions [25–27]. Naive PIR protocols require 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 their computation or communication complexities [13]. This led earlier work to conclude that the practical benefits of PIR are limited [14].

Modern PIR protocols attempt to sidestep or mitigate the impact of lower bounds by using the following techniques:

- Adding a linear-time pre-processing stage, and designing a protocol with sublinear online computation [28–30].
- Encoding the database and queries, either via replication or coding theory [15, 16, 29, 31–34].
- Using custom homomorphic primitives [34–36].
- Relaxing security by allowing some leakage [17].
- Batching query handling [11, 15, 16, 29, 34].

These techniques were demonstrated to improve the practicality of PIR protocols [19].

PIR protocols that support batching queries amortize the costs of processing over the queries in the batch. Certain protocols only allow queries made by a single client to be batched [15, 16, 34]. Others allow batching of queries from different clients, commonly by a transformation to matrix multiplication [11, 29], to achieve sub-linear (but non-constant) amortized computation per query. As Table 1 shows, our work provides even lower amortized complexity.

Protocol	# Servers	Offline			Online			Extra Storage	
		Comm.	Computation		Comm.	Computation			
			Server	Client		Server	Client	Server	Client
BIM04 [29]	≥ 2	0	0	0	n^d	n^d	n^d	poly(n)	0
PSIR [15]	1	n/q_c	n/q_c	n/q_c	polylog(n)	n †	n †	0	$\sqrt{n}$
CK20 [16]‡	1	$\sqrt{n}$	n	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	0	$\sqrt{n}$
CK20 [16]	2	$\sqrt{n}/q_c$	n/q_c	$\sqrt{n}/q_c$	$\sqrt{n}$	$\sqrt{n}$	$\sqrt{n}$	0	$\sqrt{n}$
This Work	≥ 2	q_c §	n/Q	n/Q	q_c §	n/Q ¶	n/Q ¶	$n+Q$	q_c

Table 1: Asymptotic communication and computation complexity of PIR protocols *per query*. Here, n is the table size, q_c is the number of queries made by a single client, Q is the total number of queries from all clients, and $d < 1$ is a constant. Remarks:
† Contains only a sublinear number of public key operations. ¶ No public key operations. ‡ Relies on FHE.
§ The table shows the client-server communication complexity, intra-server communication is in $n + Q$.

Mixnets Traditional mixnets [37] 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 [38, 39] add *cover* traffic to obfuscate various traffic patterns. However, ad-hoc cover traffic is shown to leak information over time [40].

Recent work mitigates this by relying on secure multiparty computation [41] or differential privacy. Vuvuzela [20] adds noise traffic from a suitable distribution to achieve formal differential privacy guarantees over leaked traffic patterns, and Stadium [22] improves on its performance by allowing parallel noise generation and permutation. Similar techniques have been used in private messaging systems [23], and in differential privacy models that utilize shuffling for privacy amplification [42] or for introducing a *shuffled* model that lies in between the central and the local model [43].

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

Differential Privacy and Access Patterns Using differential privacy to efficiently hide access patterns of various protocols has seen increasing interest in the literature. Earlier work relaxes the security guarantees of PIR to be differentially private [17] in the semi-honest setting, by putting the burden on clients to hide their actual query among a number of random queries selected from an appropriate distribution. This assumes that clients are all honest, and requires the database to be replicated over a large number of servers.

A larger body of work relaxes the security guarantees of Oblivious RAM (ORAM), a primitive where a single client obliviously reads and writes to a private remote database [44, 45], to be differentially private. Extensions of ORAM address multi-client settings [46], where the obliviousness of access patterns must be guaranteed over all users. Differentially private ORAM relaxations [47] guarantee that neighboring access patterns (those that differ in the location of a single access) occur with similar probability.

Differentially private access patterns have been studied for searchable encryption [48] and generic secure computation [18] where the ideal functionality outputs a leakage function over the input

data that must itself be differentially private. The security notions between these various works are tightly connected. DP-PIR can be instantiated from DP-ORAM protocols [47] with similar differential privacy security relaxation. In both cases, privacy is provided at the level of a user query/access, and the same noise distribution can be used to hide access patterns similarly to the classic differentially private histogram release example [49].

Secret Sharing Shamir Secret Sharing [50] 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 [51,52] additionally protect against an adversary that can tamper with shares, and guarantees that tampered shares either reconstruct to the original message or to some random value. Recent work [53] shows generic transformations that construct non-malleable schemes from input secret sharing schemes over the same access structure. In this work, we design a secret sharing scheme that is both incremental and non-malleable.

3 System Overview

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

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

Finally, our offline protocol uses *onion encryption* to pass secrets through several servers, where an onion encryption of anonymous secrets is defined as:

$$OEnc(i) = Enc(a_0^i \ :: \ ... \ :: \ Enc(a_m^i, \ sk_m) \ ..., \ sk_0)$$

3.1 Setting

Separating the computation into an offline and an online stage, and speeding up the online stage by performing expensive cryptographic operations in the offline stage, [15, 16] are standard PIR techniques. Our protocol allows the costs of both offline and online stage to be amortized over many queries, even ones made by different non-collaborating clients. Unlike previous work, this amortization allows us to drive the computational cost down to a constant per query, when the number of queries is in the order or in excess of $|T|$.

We describe our protocol in terms of a single epoch consisting of an input-independent offline stage followed by an online stage. Due to input independence, our protocol alllows for executing executing a larger offline stage in one shot, which can then support several online stages executed afterwards. Practical deployments may benefit from this feature to run the offline stage when computing power is cheaper, after which the clients have flexibility to choose when to make their

queries. Note that the converse is not true: secrets resulting from several offline stages cannot be used in any single online computation, or else an adversary could identify which offline stage each query belongs to.

The client state, created by 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. While corresponding stages operate over batches of the same size, the offline stage 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, whether via having larger machines with increased CPU power, or more distributed machines. Thus, it may reasonable to run the offline stage with more resources, if these additional resources are cheaper to acquire overnight (e.g., Amazon EC2 spot instances).

While the requirements on the offline stage are primarily throughput oriented, the online stage must exhibit good latency as well. Our offline stage is similar to the Vuvuzela protocol [20], which exhibits good throughput. However, our experimentation demonstrates that using techniques similar to Vuvuzela incur large latency over any single query, and is thus unsuited for use in our online stage, since a response may only be given to a client after the entire batch has been handled, to avoid leaking information about the secret permutation used to guarantee privacy. This requires all the work be included in the latency, including the linear number of public key operations.

3.2 Threat Model

Our threat model is the same as Vuvuzela [20]. We tolerate up to $m - 1$ malicious servers and $n - 1$ malicious clients. Our protocol leaks noisy access patterns over the honest clients' queries, in the form of a noisy histogram $\mathcal{H}(Q) = H_{\text{honest}}(Q) + \chi(\epsilon, \delta)$, such that the histogram release is differentially private.

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

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

3.3 Interpreting Security

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

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

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

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

This definition is identical to that of Mazloom and Gordon [18]. Whereas traditional differential privacy mechanisms trade privacy for accuracy, by introducing noise into the output, schemes meeting this definition trade privacy for performance, such that increasing privacy (lowering ϵ and δ) results in additional noise queries and thus increased load on the system.

We have chosen to define security in terms of a leakage function, rather than using a game formulation [17], because it interacts better with simulation-based security proofs, which we use to show the security of our protocol. Simulation-based security holds when the protocol is sequentially composed, which extends our security to cases where several online stages are pooled together into a single offline stage. Furthermore, sequential composition adds up nicely with differential privacy composition guarantees, when several instances of the protocol are run over (potentially) correlated queries.

Note that our leakage is defined over plaintext values in the database. In the ORAM domain, it is common to define leakage over ciphers, since the database is encrypted or garbled. In our setting, these two definitions are equivalent. We experimented with alternative formulations, where the underlying database is encrypted in the offline stage, and queries and responses are correspondingly encrypted and decrypted during the online stage. However, this did not improve privacy: the adversary can transform the histogram over ciphers to an equivalent one over plaintexts.

The adversary can choose its noise query according to an alternate distribution that differentiates entries in the database (e.g. make all the noise query against a single key). The leakage information contains this noise, in addition to the honest noise inserted by honest servers. While this leakage remains differentially private, the adversary can use its alternate distribution to correlate ciphers (over which the leakage is produced) with plaintexts (over which the malicious noise queries are made). The adversary can correlate these entries with more certainty the more noise it is allowed to inject into the system. This remains possible to do even if a server's ability to introduce noise is limited (e.g., with noise verification and ZK proofs), if the adversary can corrupt (or register its own) clients and use their actual queries to skew the histogram distribution.

This seems to be an inherent limitation of protocols that allow the adversary control over a portion of noise or input queries, which applies beyond our specific setting and beyond PIR. While traditional client-server ORAM does not have this limitation, since all queries are made by the client, other extensions of ORAM, such as mutli-client or multi-server ORAM may exhibit this equivalence of leakage over ciphers and plaintexts.

4 Incremental Non-malleable Secret Sharing

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

The secret sharing scheme must provide 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. *Independence:* An adversary cannot link any partially reconstructed output from a set of outputs to any shared input in the corresponding input set.

3. *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 intelligently to enforce that corresponding reconstruction yields a specific value (e.g., 0), or a specific function of the original secret (e.g., adding a fixed offset).

Non-malleability is critical for preserving security for cases where the last (backend) server is corrupted. The backend can observe the final reconstructed values of all queries, and can 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 corresponding client.

The first two properties can be easily achieved with any appropriately-thresholded secret sharing scheme. Secrecy is guaranteed since reconstruction is impossible without knowing all the shares. Independence can be enforced by having every query use an independent random set of shares. However, non-malleability excludes several such schemes, such as additive or XOR-based sharing. Furthermore, our protocol is *incremental*: a server processing a set of input messages consumes its allocated shares for these queries to produce the output messages, until the final server finishes processing, and the original queries are reconstructed.

While several non-malleable secret sharing schemes exist [51, 52], they don't satisfy our incremental design. It would have been possible to use different primitives that satisfy these properties, such as authenticated onion symmetric-key encryption. However, these operations remain more expensive than simple information theoretic secret sharing schemes that can be implemented with a handful of arithmetic operations.

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

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

Note that all shares except the first one can be selected prior to knowing q.

The secret q can be constructed incrementally, by consuming a running tally l_{j-1} and one secret share (x_j, y_j) at a time and producing a new running tally l_j of the reconstruction, such that $l_m = q$:

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

Secrecy It is obvious that knowing only m shares or less provides no information about the secret, since there is an equal number of combinations of values in the remaining share pair that produce any desired output when reconstructed, regardless of the initial secret.

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

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

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

The game demonstrates that regardless of what tampering the adversary performs (it has the freedom to select any tallies), the next tampered partial tally is indistinguishable from a random one, even knowing what the non-partial tally should have been. This is a strong guarantee that implies that the actual tally cannot be extracted from the tampered one, since the existence of any such extraction mechanism provides a way to distinguish the real and ideal world. Theorem 1 demonstrates our construction satisfies this security game. We provide a proof of the theorem in Appendix D.

Theorem 1 (Incremental Non-malleability of Our Scheme)**.** *The real and ideal dealers from the incremental non-malleability security game are statistically indistinguishable, when the game is instantiated with our secret sharing scheme.*

5 Protocol

5.1 Intuition

With differential privacy, the amount of noise needed to make a histogram release private only depends on the number of cells in the histogram, but not its actual values. In our setup, this translates to having noise queries induced by the system independently of the number of queries. This differs from how one might achieve perfect secrecy using the strategy of adding cover noise, where the servers would need to pad all entries to the largest value.

The independence of the DP noise from the number of input queries is desirable. It allows for scaling to a very large number of queries, which amortizes the cost of the noise queries. However, this only works if the noise and real queries produced by honest servers and clients are indistinguishable

Algorithm 1 Client Offline Stage

Input: Nothing.

Output: Shared anonymous secrets $a_0, ..., a_m$, one per each of the m servers.

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

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

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

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

from each other. This means that any query (fake or real) in the processed output set of an honest server should not be correlated with any query in the input set of that server, even when the adversary has complete view of the inputs and outputs, but not the honest server's secret state. We achieve this by ensuring that honest servers both shuffle the input queries and re-randomize their values. We rely on onion decryption to perform this randomization in the offline stage, while relying on our incremental non-malleable reconstruction during the online stage, which uses anonymous secrets produced during the offline stage.

As an optimization, we do not need to use a non-malleable secret sharing scheme for response handling. This is because the adversary cannot observe the final response output, which is only revealed to the corresponding client, and thus cannot observe any useful information from the effects of the perturbation. Our protocol uses additive secret sharing, since every share in it consists of a single group element, while shares of our incremental non-malleable scheme consists of two field elements.

5.2 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 deadline passes, or after a certain number of messages from clients is reached. All incoming messages at that point constitutes the input set for that server, which get processed by that server into an output set of a larger size, since the server also injects its own messages into the set, consisting of its own random secrets that she needs to perform the noise addition in the online stage later on. Algorithms 1 and 2 show the client and server portions of the offline stage respectively.

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

Algorithm 2 Server s_j Offline Stage

Input: The underlying database $T = K; V$, and privacy parameters ϵ, δ.
Output: A mapping M of unique tag $t_j i$ to its corresponding shared anonymous secrets a_j^i, one per incoming request i. A list of shared anonymous secrets L, each entry containing a secret $a_{j'}$ per server $s_{j'} > s_j$. Histogram $\mathcal{N}$ of noise queries.

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 installs entry $(t_{j+1}^i, x_j^i, y_j^i, e_j^i)$ mapped by key t_j^i in its local mapping M, for later user in the online stage.

If $j < m$:

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

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

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

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 possible (sub-components of) their plaintexts (by CPA-security), and since the output set is uniformly shuffled.

The number of shared anonymous secrets injected by a server and stored at subsequent servers must suffice to handle all noise queries that the server needs to inject in the online stage. As a result, our protocol requires the server to pre-sample the histogram of noise during the offline stage to determine the total count of noise queries that will be used during the online stage, and inject precisely the needed amount of shared anonymous secrets into the output set. While the sampling of a histogram is cheap, the construction of the anonymous secrets is expensive, since each secret needs to be onion encrypted with the keys of the subsequent servers. Note that this sampling and construction does not depend on the input messages from clients or previous servers.

Finally, note that no entry in the output set of a server can be sent over to the next server until the first server finishes processing all of its input set. Otherwise, an adversary with view of both input and output sets can learn information about the shuffling order, either directly by observing that an output message cannot correspond to an input message that the server did not receive yet, or more indirectly by observing timing information. To avoid wasting server resources subsequent to the server currently processing its input set, we have these subsequent servers spend their resources pre-sampling the noise, and onion-encrypting the shared anonymous secrets they need to inject, so that onion encryption operations are no longer on the critical path of the computation. Note that

Algorithm 3 Client Online Stage

Input: A query q, and a shared anonymous secrets a_j per server s_j stored from the offline stage.
Output: A response r, corresponding to the value associated to q by table T.

1. **Compute Final Incremental Secret Share**: Client computes $l = x_0$, so that $(x_0, 0)$ combined with $(x_1, y_1), ..., (x_m, y_m)$ is a valid sharing of q, per our incremental secret sharing scheme.
2. **Query Submission**: Client sends (t_1, l) to the first server on the chain s_1.
3. **Response Reconstruction**: Client receives a response r_1 from s_1. The client reconstructs the response $r' = r_1 - \sum e_j \ mod \ z$.
4. **Response verification**: The client ensures that the response $r' = (q, r)$, where q is the original query. Furthermore, it verifies that it is signed against $pk = (pk_1, ..., pk_m)$.

the first server can perform this onion encyption ahead of time or while client queries are being batched.

5.3 Online Stage

Our online stage is structured similarly to the offline stage. However, it requires going through the chain of servers twice. The first phase moves from the clients to the backend server, where every server injects its noise queries into the running set of queries, and incrementally reconstructs the values of received queries. The second phase moves in the opposite direction, with every server removing responses to their noise queries, and incrementally reconstructing the values of received responses, until the final value of a response is reconstructed by its corresponding client. The details of the client and server portions of the online protocol are shown in Algorithms 3 and 4, respectively.

The last server in our chain, which we call the backend server, operates differently than the rest of the servers. The backend gets to see the reconstructed query set, and find their corresponding responses in the underlying database. The backend need not add any noise queries, which alleviates the need for shuffling or batching at the backend. On the other hand, all other servers need to inject noise and ensure that they only start sending their output queries and responses sets after they are done processing their corresponding input set.

When a client makes a real query, or a server makes a noise query, they attach an ephemeral, anonymous tag to it that identifies which anonymous secret installed at the very next server should be used when handling this query. That installed secret itself contains a corresponding tag that in turn identifies which secret at the next server should be used then, and so on. This onioning of tags mirrors the onion encryption technique, and relies on it for anonymity. Corrupted servers preceding the first honest server can identify any client query, either by revealing each of their shuffling orders to each other, or by tracking the offline shuffling orders of the tag corresponding to that query. However, corrupted servers succeeding the honest server cannot: they cannot link tags in the offline stage to their owners, since the honest server onion decrypts and shuffles during that stage, and they cannot link the queries during the online stage, since their values are modified by incremental reconstruction and their onioned-tags are determined by the honest server.

Algorithm 4 Server s_j Online Stage

Input: The underlying database $T = K; V$, the mapping M, list L, and noise histogram $\mathcal{N}$ stored from the offline stage.
Output: Nothing.

1. **Anonymous Secret Lookup**: For every received query (t_i^j, l_j^i), the server finds the corresponding anonymous secret from the offline stage at $M[t_i^j] = (t_i^{j+1}, x_i^j, y_i^j, e_i^j)$.

2. **Query Handling**: For every received query, the server computes the output query $(t_i^{j+1}, R(l_j^i, (x_i^j, y_i^j)))$ using the corresponding anonymous secret, where R is our scheme's incremental reconstruction function. The server also stores the corresponding pre-additive secret share e_i^j.

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. Note that there are exactly as many secrets in L as overall queries in $\mathcal{N}$ by construction.

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}, when the server receive the corresponding responses from s_{j+1}, she de-shuffles them using the inverse order used by shuffling.

5. **Response Handling:** All received responses corresponding to noise queries generated by this server are discarded. For every remaining received response r_{j+1}^i, the server computes the output response $r_j^i = r_{j+1}^i + e_i^j \ mod \ z$. Responses and pre-additive shares can be associated because the responses are deshuffled.

6. **Response Forwarding:** The server send out all computed output responses to the previous server on the chain, without any further shuffling, so that the previous server can deshuffle them accordingly. If this server is the very first server, the responses are instead sent to their respective clients, which can be identified since the responses are de-shuffled and match the received queries' order.

If $j = m$:

7. **Response Lookup**: The backend server does not need to inject any noise or shuffle. Instead, after an input query is handled, $R(l_m^i, (x_i^m, y_i^m))$ reconstructs to the original query value, given that the query was handled honestly up to this point. The backend server finds the corresponding response value $r_m^{'i}$ by looking the query up in table T. If the query was not found, the server sets r' to be any random value (or a value that corresponds to error), so that malicious clients cannot deny service to other clients by making incorrect queries.

8. **Response Handling:** The server computes the corresponding output response $r_m^i = r_m^{'i} + e_i^m \ mod \ z$. The server sends the output response to the previous server on the chain, without any shuffling.

A similar approach is used during the second phase when responses are handled. The response values themselves are incrementally reconstructed using additive pre-shares installed during the offline stage. Furthermore, we do not need to send or keep track of any tags, instead a server can link a response to a query in her input sets via their order, which can be used to identify the appropriate additive pre-share to use. Our protocol achieves this by requiring that every server deshuffles the responses it receives inverting the shuffling it performed on its input queries.

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. Additionally, a backend server may choose to provide incorrect responses to queries by ignoring the underlying database T.

Each of these deviations has the same effect: queries that are affected or *mishandled* by the deviation will end up reconstructing to a random value, by non-malleability. Note that mishandling in the response round will result in these responses reconstructing to an unverifiable value at the client side, unless the malicious servers can forge signatures. In either case, this implies that 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 by mishandling their queries (or responses), a malicious backend can deny service to any number of client who queried a particular entry in the database, by incorrectly responding to such queries, and any server can deny service to random clients by mishandling random queries or responses. However, note that the backend and frontend capability cannot be combined even when colluding: because at least one honest server exists between the frontend and backend, the backend server cannot link queries it received to those the frontend server produced, and therefore cannot link them to client identities. These guarantees are similar to those of Vuvuzela [20] and many other mixnet systems.

5.4 Formal Security

We rigorously specify our security guarantee in Theorem 2, which refers to the ideal functionality defined in Algorithm 5. The ideal functionality formalizes our notion of "selective" abort. In particular, it formalizes capabilities of the adversary to deny service to certain clients based on client identity, query value, or neither (depending on which servers are corrupted).

Theorem 2 (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 $q^1, ..., q^n$, we have:*

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

A construction of the simulator and proof for Theorem 2 are shown in Appendices A and B.

5.5 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

Algorithm 5 Ideal Functionality $\mathcal{F}$

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

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

being differentially private, which in turn depend on the underlying noise distribution. Several differentially private mechanisms exist in the literature for a variety of functions and statistics. Our application requires that the noise we add be non-negative (we cannot add a negative amount of queries) and integer (e.g., we cannot add half a query), plus it is desirable to have an upper bound on the amount of queries. Algorithm 6 details how we modify the standard Laplace mechanism to satisfy these properties. Theorem 3 proves that this distribution indeed provides differential privacy. The proof can be found in Appendix C.

Theorem 3 (Leakage is Diferentially Private)**.** $\mathcal{H} = H_{honest} + \chi(\epsilon, \delta)$ *is* (ϵ, δ)*-Differentially Private.*

Standard unmodified Laplace noise traditionally provides the best noise to privacy ratio, and we believe our modified Laplace still exhibits good performance. However, other distributions exist in the literature and some have been used for hiding access patterns in oblivious protocols. Some of these distributions may provide different trade-offs, particularly distributions that are discrete (e.g., two-sided geometric [18] or Poisson [22]), are non-negative (e.g., geometric) or have finite support (e.g., binomial). All of these distributions will require some amount of modification to be used in our context, and it is well understood that using one-sided noise (e.g., non-negative) induces a δ loss.

In addition to optimizing for the mean of the distribution, which estimates the mean latency and through put of the system, the variance can be interesting for applications or setups that benefit from increased predictability between epochs. The variance also influences how δ behaves as a function of the bounds on the amount of noise added, such that the higher the variance is the larger the range of the allowed noise should be to achieve the same δ. We note that the Laplace mechanism has well known exploits when implemented using floating point arithmetic [54], although

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

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

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

For every $i \in |T|$:

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

careful reformulation of our modified Laplace distribution is sufficient to overcome them, since our distribution is over integers.

The lower bound on noise values in our modified distribution is 0. This means that with some non-zero probability, an entry in the table may have no noise queries made against it. Imagine that the set of client queries also did not refer to this entry, this information is then leaked by the noised histogram. A symmetric observation can be made for the upper bound on the noise. This leakage is configurable and modeled via δ, such that the smaller the δ the less likely such an event occurs, at the cost of a larger mean and upper bound on the amount of noise queries. This behavior remains identical even if the lower bound is set to a different non-zero value by shifting.

6 Scaling and Parallelization

Our protocol benefits greatly from parallelizing it over multiple machines with increased networking and computing resources. This is particularly important for our offline stage, which is CPU-bound because of its public key operations.

In a non-parallel setting, the notions of a party and a server are identical. For scaling, we allow parties to operate multiple single or multi-core machines. These machines form a single trust domain because they are owned by the same party. In other words, we assume that if one of these machines is corrupted, all other machines belonging to the same party are also corrupted. This maintains our security guarantees at the level of a party, particularly, the protocol remains secure even if only one party (and all its machines) is honest. This setup allows machines owned by the same party to share all their secret state, including anonymous secrets installed at any of them during the offline stage, as well as the noise query they select.

Our overall design is simple: any machine m_i^j belonging to party j only communicates with a single machine m_i^{j-1} and m_i^{j+1} from the preceding and succeeding parties. m_i^j communicates with these two machines to receive inputs and send outputs respectively. A machine also communicates with all other machines belonging to the same party for shuffling. This design is shown in Figure 1.

It is important that shuffling be uniform globally over all messages received by all machines of the same party, so that the identity of the machine that sends the output message to the next party, as well as the relative order of that message, leaks no information about its source.

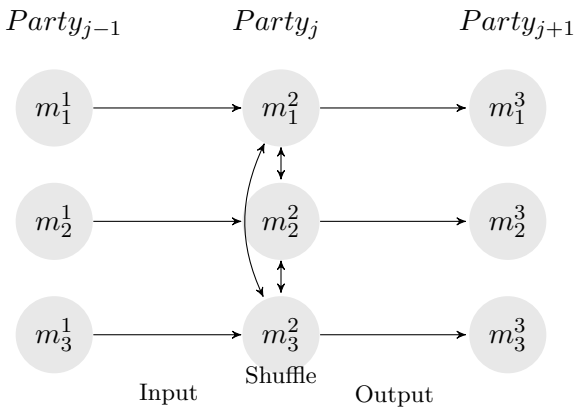


Figure 1: Parallel Design

Distributing Noise Generation Since we are generating noise independently for each entry in the table T, we can parallelize the generation by assigning each machine an equal subset of table entries to generate noise for, e.g. m_i^j is responsible for generation all noise queries corresponding to $\{T[k] \mid k \ \% \ j = 0\}$. This distribution is limited by the number of entries in the table. If parallelizing noise generation beyond this limit is required, an alternate additive noise distribution (e.g. Poisson [22]) can be used instead, which allows several machines to sample noise for the same table entry from proportionally smaller distribution, while maintaining the desired global distribution.

Distributed Shuffling There are several candidate coordination-free distributed shuffling algorithms that may be used in our setting, the simplest is to have each machine determine a target machine for every query uniformly at random, and have that machine shuffle all messages it receives locally. This simple algorithm ensures that the expected number of messages targeted at each machine is the same, but with some non-zero variance. This variance is undesirable, our protocol is extremely synchronized between parties, with one party only sending out messages to the next one after all its messages have been processed, which means that a party (and our protocol) is as fast as the slowest machine. Therefore, we prioritize maintaining an identical load of queries per machine.

Since machines belonging to the same party are in the same trust domain, they can agree on a single secret seed, which is used to simulate a global shuffle over all the inputs to that party at every machine locally. This can be done knowing only the size of the input set, rather than all the elements in it. For any given machine, this global shuffle indicates which machine is the target for each of its input messages, it also indicates what is the local ordering of messages within a target machine. This ensures that the load of every machine is identical, without the need for coordinating over the network. However, it does require that machines perform this simulated shuffle over the entire number of inputs, rather than only their own. We mitigate this by having machines perform this shuffling simulation prior to receiving their inputs, such as during the time when previous parties are still processing. Therefore, the shuffling simulation is not executed on the critical path of our protocol, and the protocol exhibits perfect parallelism.

Distributing Offline Anonymous Secrets At any given party, both noise and real queries are unlikely to be processed by the same machine that processed their corresponding anonymous secrets during the offline stage, particularly because the shuffling order is independent in both stages, but also since in practical deployments, machines may be removed and new machines may be added by the party between the two stages to satisfy changing demand, especially in scenarios where multiple online stages are pooled over a single larger offline stage.

While it is possible to construct a mapping identifying which machine has the secrets for each query based on the tag attached to the query, this mapping is asymptotically as large as the secrets themselves, and adds additional networking burden, since the query must first be sent from the machine that initially received it to the machine that can process it, which in turn must send it to the target machine for shuffling. Instead, it is more desirable for all the machines within a party to exchange and replicate the secrets after the offline stage, so that any machine is capable of processing any given query.

Our experiments and implementation assume that the entire set of secrets can fit on any single machine's main memory. In practice, this can become a limitation when the number of queries becomes too large, and distributing the storage becomes critical. While it is possible to use hard drives or SSDs to store these secrets, we believe that the overhead of searching for and reading the secrets from slower storage mediums may introduce significant overheads. Instead, a better alternative is to use a remote distributed in-memory file system or key value storage, especially ones that support Remote direct memory access (RDMA), offload filtering (e.g. by tag) to the remote machine, and allow batching of multiple requests to amortize network round-trip time over multiple queries. These techniques are gaining in popularity in modern distributed systems, and have been implemented was success in various research [55, 56] and production systems [57, 58].

7 Experimental Results

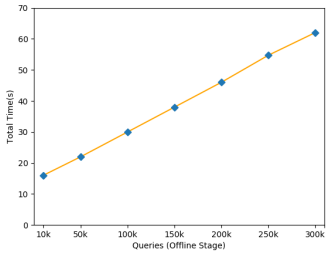
7.1 Setup

All our experiments are run on Amazon EC2 t2.micro instances. These are free tier eligible with a single 2.5GHz core, 1 GB of RAM, and low network bandwidth. Our measured time, displayed on the y-axis, represents the end-to-end runtime of our entire protocol. For the offline stage, this is the total runtime from the start of the stage until all secrets are installed on all servers. For online stage experiments, this is the wall time of the client whose query was the longest to satisfy. Because our system is highly synchronized, i.e. all the queries have to be processed by each party before they are sent to the next party (or back to the clients), the end-to-end time represents the longest latency any one client experiences, as well as the total time it takes to process all queries.

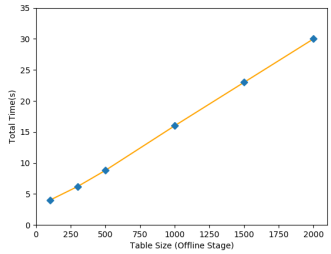
All online stage experiments are run against a table of size 10k, 2 parties, and 1M queries. All offline stage experiments are run against a table of size 1k, 2 parties, and 100k queries, except when otherwise noted in the x-axis. We use privacy parameters $\epsilon = 0.1$ and $\delta = 0.0001$ in all our experiments. Our implementation is open source, and available at `https://github.com/multiparty/DP-PIR`.

7.2 Results

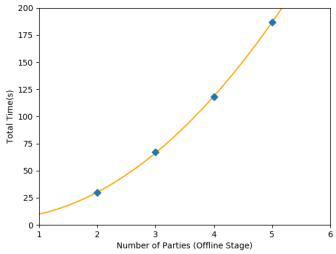
Our experiments demonstrate that the primary factor affecting the performance of our protocol is the ratio $P = \frac{|Q|}{|\mathcal{N}|}$ of real to noise queries, where the expected number of noise queries is $\mu *$



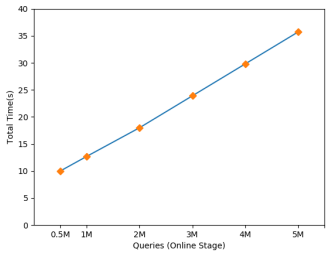
(a) End-to-end offline time over varying query loads



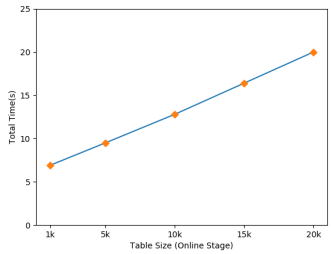
(b) End-to-end offline time over varying database sizes



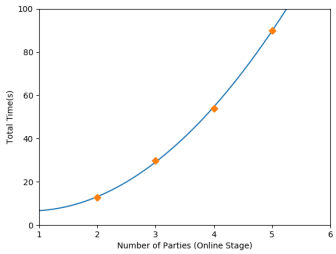
(c) End-to-end offline time over varying number of parties



(d) End-to-end online time over varying query loads



(e) End-to-end online time over varying database sizes



(f) End-to-end online time over varying number of parties

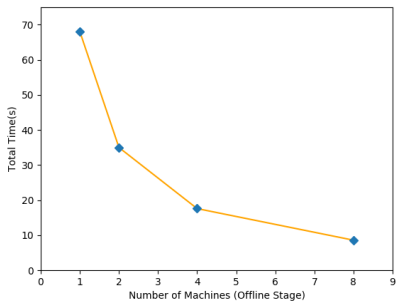
Figure 2: End-to-end runtime for the offline (top) and online (bottom) stages as a function of query load (left), database size (middle), and number of parties (right). The exact setup is described in Section 7.1.

$|T| * |parties|$. Table 2 shows hard baselines exhibited by our system, both in terms of throughput and total dollar cost. These baselines are defined over the overall queries in the system, both real and noise. Thus, the system is not guaranteed to meet these baselines. Instead, our experiments demonstrate that the system approaches these baselines as the ratio P grows larger. Furthermore, the baselines do not include speedups from parallelization and deployment using large machines.

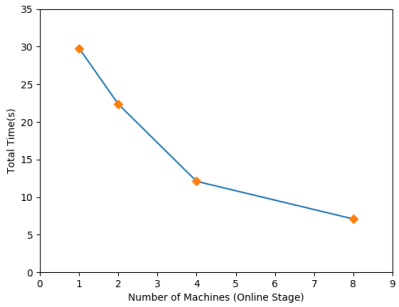
Figures 2a and 2d demonstrate how this ratio affects the system. While we expect and show that our protocol scales linearly with the number of queries both in the offline and online stages, we observe that the slope of this linear growth is governed by the ratio P. When the number of real queries dominates the performance and exceeds the amount of noise injected into the system, this slope approaches 1. As P keeps on growing with the number of queries, the cost of processing noise queries remains fixed and quickly becomes negligible. As a result, our protocol amortizes costs per query down to a constant.

Figures 2b and 2e display a similar story to the previously discussed figures. Increasing the table size results in a linear increase in the number of noise queries. When P remains large, these increases do not observably affect the overall performance of the system. As the number of noise queries dominates P, their increase overwhelms the overall performance cost, causing the slopes in these two subfigures to approach 1.

Finally, our online and offline stages both exhibit perfect parallelism, as seen in Figure 3. This is particularly true for the offline stage, which is heavily CPU-bound because of the large number of public key operations performed in it. Our online stage on the other hand is mostly network



(a) Offline stage scaling



(b) Online stage scaling

Figure 3: Scaling our protocol over parallel machines.

	Offline		**Online**	
Parties	Queries/sec	$/1M Queries	Queries/sec	$/1M Queries
2	6250	0.001	154321	4×10^{-5}
3	4137	0.0023	100502	9×10^{-5}
4	3108	0.0041	73260	17×10^{-5}
5	2400	0.0053	57471	28×10^{-5}

Table 2: Baseline throughput and cost over real and noise queries.

bound. However, it still benefits from parallelizing over machines with independent network resources, since that enables the network bandwidth to be utilized more effectively. Increasing the number of machines from 1 to 2 exhibits a less than perfect speed up, since the introduction of distributed shuffling, relative to a single machine's local shuffling, causes machines to perform additional network communication, which eats away from the reduction in bandwidth from parallelism. However, the speed up becomes perfect when scaling beyond 2 machines.

7.3 Limitations

While our system performs well when the ratio P is high, the opposite is true when that ratio is low. When noise queries dominate real ones, it causes our protocol to spend a majority of the time processing noise queries, which do not have value in of themselves beyond covering the underlying access patterns. We foresee two scenarios where this may happen that limit the performance of our protocol. First, when the number of queries is small relative to the size of the underlying table, the protocol is unlikely to perform well, since that case benefits very little from batching and amortization. Second, when the privacy parameters are amplified, such that they overwhelm the ratio P, the protocol exhibits too much load from noise queries, reducing the overall amortization. Both of these limitations are expected, query loads that have little opportunity for batching are a better fit for streaming protocols, that process incoming queries on the fly, and privacy setups that approach having no leakage are better suited for PIR protocols that guarantee perfect security by default. Finally, increasing the number of parties implicitly increases the amount of noise, since each party adds noise independently. This compounds with the overhead of having an additional party in the protocol chain, as shown in Figures 2c and 2f.

8 Case Study - Route Recommendation

Unlike traditional PIR, differentially private PIR offers security guarantees that are sensitive to sequential composition. Furthermore, our protocol has several parameters and deployment details that can be tweaked per application for enhanced performance. We demonstrate how our protocol can be effectively instantiated for the use case of route recommendation, and how certain lessons learned from this use case can provide insight into adoption to other applications.

Route recommendation services, such as Google Maps, are extremely popular services with many millions of clients using them on a daily basis. Existing work attempts to modify graph algorithms used for route recommendation to provide privacy guarantees, via the use of various cryptographic constructions, such as secure multi party computation (MPC) [59] and Oblivious RAM [60]. A different line of work demonstrates that privacy-preserving route recommendation may be implemented using PIR [12]. This is an attractive alternative: it does not require transforming existing route recommendation algorithms into secure ones, many of which are complex and proprietary with considerable computational complexity, and thus avoid adding considerable design burdens and performance overheads. Existing route recommendation algorithms can be used as-is to generate a routing table, where the keys are pairs of source and destination nodes, and values are (partial) recommended paths between them. Route recommendation in this scenario becomes a direct instance of PIR, where clients query this routing table privately to retrieve recommended routes.

Iterative Querying Storing the entire shortest path in the value column in the table is undesirable. It introduces significant storage overheads, and causes responses to be much larger in size. Instead, only a few (traditionally one) next hops are stored and returned whenever a client makes a query. The client can then make follow up queries to reconstruct the remaining path between the last returned next hop and the original destination. The exact number of hops to store and return has performance implications. On the one hand, more hops mean fewer iterative queries, on the other hand, each of these queries, including their corresponding offline stage secrets, becomes larger. The optimal number of hops to store is likely to depend on the particular deployment environment, incoming request profiles, financial costs, and latency requirements, and is probably best found empirically after fixing these constrains.

Composition The number of queries a client consecutively makes may reveal the length of the recommended path. It is common to require clients to pad that count to the diameter of the map, by making additional random queries. Let us denote the total number of required queries to reach the diameter by g. In the worse case, all g of these queries are correlated, the next one picks up where the previous one left. Furthermore, while these queries are physically separated over several online stages, they logically constitute a single query. In most applications, it makes sense to provide privacy guarantees at the level of such logical client query, rather than at the level of implementation-specific partial queries. This is particularly true in route recommendation, where learning some information about few of these queries may reveal a lot about the overarching query, including the destination, bounds on the path length, and partial portions of the shortest path.

Therefore, the noise parameters must be configured to provide $(\epsilon, \delta)-$differntial privacy at the level of g correlated queries, rather than only a single one. One way to analyze this is to use the composition theorem, where the noise added per online stage provides more than the required

total guarantees locally, such that when g such guarantees are composed, the end total boils down to (ϵ, δ). Alternatively, we can model this by considering neighboring histograms that differ in g queries, rather than one, when computing the sensitivity.

Table Size Unlike other more direct applications of PIR, route recommendation experiences a penalty when expressed as a table: while the underlying map may be of size n, the routing table has size n^2. This additionally implies that our protocol adds noise proportional to $O(n^2)$ rather than $O(n)$. Concretely, this means that our protocol is attractive to use for route recommendation when the number of client queries is large relative to the table size, not the map size. There exists work in the literature that attempts to mitigate this quadratic overhead, by compressing the table [12], relying on low-degree geometric shapes to encode regions of the map, or utilizing contracting hierarchies, where routing is first done at less granular levels (e.g. cities or neighborhood) and then moved to increased granular levels (e.g zip codes and then street addresses) [61].

Offline and Online Load Balancing Intuitively, route recommendation services experience most of their load during peak hours, when more people are commuting, rather than during off-peak hours, when traffic is low. It is best to spend the computing resources on online computation during peak hours, while spending them on offline computation during off-peak hours. Thus, it is reasonable to pool-in many online computations, about as many as we expect peak hours to have for the given batching time epoch, into a single offline stage that runs when load is low. Front-end applications of the service (e.g. the Android Google Maps app) can be given a deadline (e.g. until midnight every day) to submit anonymous secrets to the service to be used online later that day. After the deadline passes, the offline stage kicks in while computing on a majority of the service's computing resources, with only a reduced amount of resources preserved to process online queries coming in during those hours. Finally, it makes sense to adjust the batching time interval for the online stage based on load, to keep latency within reasonable bounds as the load changes through out the day.

Replication and Scaling Finally, it makes sense to run several instances of the protocol in parallel, to allow online stages with large batching times to overlap, and provide geo-replication keeping application servers in close proximity to each other and to their clients. Depending on setup, this may require additional design effort to determine how anonymous secrets that were installed during the offline stage of one replica may be used in the online stage of a different replica.

9 Future Work and Extensions

Reducing Noise The applicability of our protocol to any particular use case hinges on the ratio of real to noise queries, which in our protocol is in the order of $\frac{|Q|}{|T|}$. Reducing the overall amount of noise injected into the system can make our protocol more attractive to applications where this ratio is low. An interesting recent line of work [42, 43] suggests that adding an oblivious shuffling mechanism to any local-model differentially private protocol provides better differentially private guarantees than an identically-configured one without. Since we assume that at least one server is honest and hence acts as an oblivious shuffler, then a tighter analysis of our system will yield better privacy parameters than the ones we suggest, over the same amount of noise. Alternatively, we might achieve the same privacy with fewer noise queries.

Leveraging Honest Clients Another direction for reducing the noise is to consider different weaker thresholds. Imagine a setup where 2-out-of-m parties are trusted to be honest; it is likely possible that the same privacy parameters can be met in that setup with each party injecting a reduced amount of noise. However, this does not immediately follow from the composition theorem of differential privacy: these 2 honest parties may be separated in the chain by adversarial parties, whose actions would affect query vectors that consists of a fraction of the total noise needed for achieving differential privacy. It is unclear to us whether this can be exploited by the adversary, or whether it is a limitation in the privacy analysis techniques used to reason about such scenarios. This direction of work may have implications on differential privacy research beyond PIR. Techniques such as noise verification [23] may be useful here to ensure that noise generated by an honest server is not tampered with prior to reaching the next honest server. It is worth noting that in certain setups, our protocol can be demonstrated to be safe while using a fraction of the total noise per party. For example, if an honest majority of parties is assumed, it is guaranteed that there will be at least two consecutive honest parties, and thus parties are able to inject half the total amount of noise each.

Streaming One of the limitations of our system involves needing to batch real queries with fake ones. A possible extension of the protocol could provide ways to stream queries as they come without stalling the system. The crux of this problem lies in finding appropriate mechanisms to streamline noise queries with real ones, while handling frequency and traffic analysis leakage over time. There is some amount of work in differential privacy literature that explores such streaming mechanisms [62–65]. This may be complemented with other Mixnet-based techniques, such as randomly introducing delays to messages as a way to mimic shuffling [66]. This line of work requires rethinking about time intervals and time delays as additional dimension that needs to be protected by differential privacy via randomization.

Fairness Another dimension for improvement is providing security up to abort with fairness guarantees. While there may be inherent limitations to our design, e.g., a server refusing to accept any queries and thus denying everyone service, it may be possible to improve the criteria upon which such a server may refuse service, for example, not allowing backend servers to deny service based on the query values. We foresee that this could be achieved by using techniques from verifiable mixnets [22], in order to prove that no intermediate server drops messages along the way, and constructing appropriate zero-knowledge proofs to ensure that the front-end and back-end servers do not drop messages either.

Acknowledgments

The authors are grateful to Frederick Jansen, Jens Schmuedderich, and Malte Schwarzkopf for their valuable feedback on earlier versions of this work. This material is supported by the National Science Foundation under Grants No. 1414119, 1718135, and 1931714, by DARPA under Agreement No. HR00112020021, and by Honda Research Institutes.

References

[1] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, no. 6, p. 965–981, Nov. 1998. [Online]. Available: https://doi.org/10.1145/293347.293350

[2] 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*, ser. FOCS '97. USA: IEEE Computer Society, 1997, p. 364.

[3] N. Borisov, G. Danezis, and I. Goldberg, "Dp5: A private presence service," *Proceedings on Privacy Enhancing Technologies*, vol. 2015, no. 2, pp. 4–24, 2015.

[4] R. Parhi, M. Schliep, and N. Hopper, "Mp3: A more efficient private presence protocol," in *Financial Cryptography and Data Security*, S. Meiklejohn and K. Sako, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 38–57.

[5] A. Kwon, D. Lazar, S. Devadas, and B. Ford, "Riffle: An efficient communication system with strong anonymity," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 2, pp. 115–134, 2016.

[6] S. Angel and S. Setty, "Unobservable communication over fully untrusted infrastructure," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 551–569. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/angel

[7] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, "Pir-tor: Scalable anonymous communication using private information retrieval," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. USA: USENIX Association, 2011, p. 31.

[8] R. Cheng, W. Scott, E. Masserova, I. Zhang, V. Goyal, T. Anderson, A. Krishnamurthy, and B. Parno, "Talek: Private group messaging with hidden access patterns," 2020.

[9] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish, "Scalable and private media consumption with popcorn," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 91–107. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/gupta-trinabh

[10] M. Green, W. Ladd, and I. Miers, "A protocol for privately reporting ad impressions at scale," in *CCS*. ACM, 2016, pp. 1591–1601.

[11] W. Lueks and I. Goldberg, "Sublinear scaling for multi-client private information retrieval," in *Financial Cryptography and Data Security*, R. Böhme and T. Okamoto, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 168–186.

[12] D. J. Wu, J. Zimmerman, J. Planul, and J. C. Mitchell, "Privacy-preserving shortest path computation," *arXiv preprint arXiv:1601.02281*, 2016.

[13] H. Lipmaa, "First cpir protocol with data-dependent computation," in *Proceedings of the 12th International Conference on Information Security and Cryptology*, ser. ICISC'09. Berlin, Heidelberg: Springer-Verlag, 2009, p. 193–210.

[14] R. Sion and B. Carbunar, "On the practicality of private information retrieval," in *NDSS*. The Internet Society, 2007.

[15] S. Patel, G. Persiano, and K. Yeo, "Private stateful information retrieval," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1002–1019. [Online]. Available: https://doi.org/10.1145/3243734.3243821

[16] H. Corrigan-Gibbs and D. Kogan, "Private information retrieval with sublinear online time," in *Advances in Cryptology – EUROCRYPT 2020*, A. Canteaut and Y. Ishai, Eds. Cham: Springer International Publishing, 2020, pp. 44–75.

[17] R. R. Toledo, G. Danezis, and I. Goldberg, "Lower-cost ϵ-private information retrieval," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 184–201, 2016.

[18] S. Mazloom and S. D. Gordon, "Secure computation with differentially private access patterns," in *CCS*. ACM, 2018, pp. 490–507.

[19] F. Olumofin and I. Goldberg, "Revisiting the computational practicality of private information retrieval," in *Financial Cryptography and Data Security*, G. Danezis, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 158–172.

[20] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich, "Vuvuzela: Scalable private messaging resistant to traffic analysis," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 137–152. [Online]. Available: https://doi.org/10.1145/2815400.2815417

[21] D. Lazar and N. Zeldovich, "Alpenhorn: Bootstrapping secure communication without leaking metadata," in *OSDI*. USENIX Association, 2016, pp. 571–586.

[22] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich, "Stadium: A distributed metadata-private messaging system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 423–440.

[23] D. Lazar, Y. Gilad, and N. Zeldovich, "Karaoke: Distributed private messaging immune to passive traffic analysis," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 711–725.

[24] A. Beimel and Y. Ishai, "Information-theoretic private information retrieval: A unified construction," in *Automata, Languages and Programming*, F. Orejas, P. G. Spirakis, and J. van Leeuwen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 912–926.

[25] B. Chor and N. Gilboa, "Computationally private information retrieval," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 304–313.

[26] C. Cachin, S. Micali, and M. Stadler, "Computationally private information retrieval with polylogarithmic communication," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 1592. Springer, 1999, pp. 402–414.

[27] H. Lipmaa, "An oblivious transfer protocol with log-squared communication," in *Information Security*, J. Zhou, J. Lopez, R. H. Deng, and F. Bao, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 314–328.

[28] G. D. Crescenzo, Y. Ishai, and R. Ostrovsky, "Universal service-providers for private information retrieval," *J. Cryptol.*, vol. 14, no. 1, pp. 37–74, 2001.

[29] A. Beimel, Y. Ishai, and T. Malkin, "Reducing the servers' computation in private information retrieval: Pir with preprocessing," *Journal of Cryptology*, vol. 17, no. 2, pp. 125–151, 2004.

[30] D. Boneh, K. Lewi, and D. J. Wu, "Constraining pseudorandom functions privately," in *Public Key Cryptography (2)*, ser. Lecture Notes in Computer Science, vol. 10175. Springer, 2017, pp. 494–524.

[31] R. Canetti, J. Holmgren, and S. Richelson, "Towards doubly efficient private information retrieval," in *Theory of Cryptography*, Y. Kalai and L. Reyzin, Eds. Cham: Springer International Publishing, 2017, pp. 694–726.

[32] E. Boyle, Y. Ishai, R. Pass, and M. Wootters, "Can we access a database both locally and privately?" in *Theory of Cryptography Conference*. Springer, 2017, pp. 662–693.

[33] A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs, "Private anonymous data access," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Cham: Springer International Publishing, 2019, pp. 244–273.

[34] S. Angel, H. Chen, K. Laine, and S. T. V. Setty, "PIR with compressed queries and amortized query processing," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 962–979.

[35] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian, "Xpir: Private information retrieval for everyone," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 2, pp. 155–174, 2016.

[36] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia, "Splinter: Practical private queries on public data," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 299–313. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wang-frank

[37] D. L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Commun. ACM*, vol. 24, no. 2, p. 84–90, Feb. 1981. [Online]. Available: https://doi.org/10.1145/358549.358563

[38] M. J. Freedman and R. Morris, "Tarzan: A peer-to-peer anonymizing network layer," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 193–206. [Online]. Available: https://doi.org/10.1145/586110.586137

[39] O. Berthold and H. Langos, "Dummy traffic against long term intersection attacks," in *Privacy Enhancing Technologies*, R. Dingledine and P. Syverson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 110–128.

[40] N. Mathewson and R. Dingledine, "Practical traffic analysis: Extending and resisting statistical disclosure," in *Privacy Enhancing Technologies*, D. Martin and A. Serjantov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 17–34.

[41] N. Alexopoulos, A. Kiayias, R. Talviste, and T. Zacharias, "MCMix: Anonymous messaging via secure multiparty computation," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1217–1234.

[42] Ú. Erlingsson, V. Feldman, I. Mironov, A. Raghunathan, K. Talwar, and A. Thakurta, "Amplification by shuffling: From local to central differential privacy via anonymity," in *SODA*. SIAM, 2019, pp. 2468–2479.

[43] A. Cheu, A. Smith, J. Ullman, D. Zeber, and M. Zhilyaev, "Distributed differential privacy via shuffling," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Cham: Springer International Publishing, 2019, pp. 375–403.

[44] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *STOC*. ACM, 1987, pp. 182–194.

[45] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, p. 431–473, May 1996. [Online]. Available: https://doi.org/10.1145/233551.233553

[46] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder, "Maliciously secure multi-client oram," in *International Conference on Applied Cryptography and Network Security*. Springer, 2017, pp. 645–664.

[47] S. Wagh, P. Cuff, and P. Mittal, "Differentially private oblivious ram," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 4, pp. 64–84, 2018.

[48] 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*, 2018, pp. 810–818.

[49] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Found. Trends Theor. Comput. Sci.*, vol. 9, no. 3-4, pp. 211–407, 2014.

[50] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[51] V. Goyal and A. Kumar, "Non-malleable secret sharing," in *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, 2018, pp. 685–698.

[52] S. Badrinarayanan and A. Srinivasan, "Revisiting non-malleable secret sharing," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2019, pp. 593–622.

[53] D. Aggarwal, I. Damgr^{a}rd, J. B. Nielsen, M. Obremski, E. Purwanto, J. Ribeiro, and M. Simkin, "Stronger leakage-resilient and non-malleable secret sharing schemes for general access structures," in *Advances in Cryptology – CRYPTO 2019*, A. Boldyreva and D. Micciancio, Eds. Cham: Springer International Publishing, 2019, pp. 510–539.

[54] I. Mironov, "On significance of the least significant bits for differential privacy," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 650–661. [Online]. Available: https://doi.org/10.1145/2382196.2382264

[55] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, cpu-efficient key-value store," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 103–114.

[56] J. Yang, J. Izraelevitz, and S. 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)*, 2019, pp. 221–234.

[57] "Apache ignite," https://github.com/apache/ignite, accessed: 2020-12-01.

[58] "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.

[59] J. Brickell and V. Shmatikov, "Privacy-preserving graph algorithms in the semi-honest model," in *Advances in Cryptology - ASIACRYPT 2005*, B. Roy, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 236–252.

[60] M. Keller and P. Scholl, "Efficient, oblivious data structures for mpc," in *Advances in Cryptology – ASIACRYPT 2014*, P. Sarkar and T. Iwata, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 506–525.

[61] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," *Transp. Sci.*, vol. 46, no. 3, pp. 388–404, 2012.

[62] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum, "Differential privacy under continual observation," in *Proceedings of the forty-second ACM symposium on Theory of computing*, 2010, pp. 715–724.

[63] C. Dwork, M. Naor, T. Pitassi, G. N. Rothblum, and S. Yekhanin, "Pan-private streaming algorithms." in *ICS*, 2010, pp. 66–80.

[64] G. Kellaris, S. Papadopoulos, X. Xiao, and D. Papadias, "Differentially private event sequences over infinite streams," *Proc. VLDB Endow.*, vol. 7, no. 12, p. 1155–1166, Aug. 2014. [Online]. Available: https://doi.org/10.14778/2732977.2732989

[65] A. Hassidim, H. Kaplan, Y. Mansour, Y. Matias, and U. Stemmer, "Adversarially robust streaming algorithms via differential privacy," *arXiv preprint arXiv:2004.05975*, 2020.

[66] D. Kesdogan, J. Egner, and R. Büschkes, "Stop- and- go-mixes providing probabilistic anonymity in an open system," in *Information Hiding*, D. Aucsmith, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 83–98.

A Simulator Construction

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

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

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

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

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

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

   - **Neither s_1 nor the backend are corrupted**: The simulator executes step 3 below.
   - **If s_1 is corrupted**: The simulator begins by identifying any mishandled honest client queries in the current query vector. For each honest client query, the simulator looks for it by its tag, which the simulator knows because she simulated the offline stage of that client. The simulator validates that the associated tally has the expected value, furthermore, it checks that the anonymous secret installed at s_i during the offline stage match the ones the simulator generated when simulating the client portion of that offline stage. All of these checks depend on the honest client and honest server s_i offline state, which the simulator knows.
     If any of tags, tallies, or shares do not match their expected value, or are missing, then the simulator knows that the adversary has mishandled this client's query (or corresponding offline stage) prior to server s_i. The simulator sends the identities of all such clients to the ideal functionality (step 1 in $\mathcal{F}$).
   - **If backend is corrupted**: The simulator receives a noised histogram H_{honest} from the ideal functionality. The simulator identifies all honest queries that have not been mishandled so far. Say there are k such queries. As part of simulating s_i, the simulator will replace the tallies of these queries with new tallies, such that the tally of honest query $w \leq k$ would reconstruct to the value of the w-th entry in H_{honest}, when combined with the remaining shares that the simulator generated for that client during its offline stage.
     Furthermore, the simulator needs to inject noise queries for s_i. The simulator chooses the tallies for these queries so they reconstruct to the remaining values in H_{honest}. This guarantees that all correctly handled honest client queries combined with this server's

noise have the distribution H_{honest}.
The simulator shuffles the updated query vector and uses it as the output query vector for this server.

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

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

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

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

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

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

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

B Proof of Theorem 2

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

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

We argue about the online stage in two cases:

Case 1: The backend server s_m is honest.

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

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

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

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

5. The backend executes the honest protocol in both worlds. While the backend sees different distributions in either worlds, since honest clients make random queries when simulated, the honest protocol is not dependent on that distribution, and only output responses in the form of secret shares. These secret shares are selected at random during the offline stage by the client, without knowing the response or the query. Therefore, the output of the backend is indistinguishable in both worlds.

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

Case 2: The backend server s_m is corrupted.

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

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

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

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

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

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

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

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

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

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

On the other hand, a response can be mishandled by (1) corrupting its tally/value (2) corrupting its relative order within a response vector. The first case arises when an adversary sets the tally value to one different than the sum of its previous value and additive pre-share from its corresponding anonymous secret, as well as when a backend server disregards the underlying database, and assigns a different initial value to a given response. The second case happens when the adversary does not deshuffle responses with the inverse order of the corresponding shuffle. The simulator can check

these two cases as well: if a deshuffle was performed correctly, then every response and query at the same index must correspond to one another, and the simulator can compute the expected value of that response from its query value, database T, and additive pre-shares. If the response and query did not match, then either the shuffling or tally computation was corrupted.

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

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

Finally, intermediate servers (those surrounded by honest servers on both ends) see only query and response vectors that have been shuffled honestly by at least one server, and have a random share applied to their tally by that server as well. So their inputs are indistinguishable from random, and thus they can only mishandle randomly. The first server (and its adjacent servers) see query and response vectors whose tallies are random (because at least one share corresponding to them is unknown), but have a fixed order, since no shuffling has yet occurred, therefore they mishandle queries based on the order (i.e. client identity) as well as randomly. Lastly, the backend (and its adjacent servers) see queries and responses that have been shuffled by at least one honest server, but whose values are revealed, since no shares of these values are unknown. The backend can mishandle queries based on their known value, but not based on their client identity, since mishandling based on index/order is identical to mishandling randomly, because the order is random.

□

C Proof of Theorem 3

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

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

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

by post-processing. □

D Proof of the Incremental Non-Malleability Theorem1

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

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

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

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

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

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

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

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