Talek: a Private Publish-Subscribe Protocol

Raymond Cheng[†], William Scott[†], Bryan Parno[‡], Irene Zhang[†]
Arvind Krishnamurthy[†], Thomas Anderson[†]
†*University of Washington,* ‡*Carnegie Mellon University*

Abstract

Modern applications share user-generated data over the cloud, often exposing sensitive information. Talek is a private publish-subscribe (pub/sub) system that shares user data through potentially untrustworthy servers, while hiding both data content and the communication patterns among its users. Talek is designed with two goals that distinguish it from the prior work in private messaging. First, Talek is designed with the strong security goal of *access sequence indistinguishability*, where clients leak no information to adversarial servers that might help an adversary distinguish between two arbitrary-length client access sequences. Second, our system aims to be practical for general-purpose workloads, from one-to-one messaging to one-to-many news feeds. To achieve these properties, we introduce two novel techniques. *Oblivious logging* is a mechanism for supporting private reads and writes to shared logs stored on servers without coordination between clients. *Private notifications* provide a private and efficient mechanism for subscribers to learn which topics have new messages without polling. We demonstrate a 3-server Talek cluster that achieves throughput of 566,000 messages/minute with 5.57-second end-to-end latency on commodity servers, a 3–4 order of magnitude improvement over related work with similar security goals.

1 Introduction

Many applications depend on cloud servers to send data between users, giving operators full insight into the communication patterns of the application's users. Even if the communication contents are encrypted, network metadata, such as HTTP headers, can be used to infer which users share data, when traffic is sent, where data is sent, and how much is transferred, allowing the network and provider to guess the contents of the communication [37]. When remote hacking, insider threats, and government requests are common, protecting the privacy of communications requires that we guarantee security against a stronger threat model. For some users, such as journalists and activists, protecting communication patterns is critical to their job function and safety.

In this paper, we present Talek, a private publish-subscribe (pub/sub) system. Talek provides user applications a single-writer many-reader log abstraction that is efficient and general-purpose, storing asynchronous messages on untrusted servers without revealing metadata. Pub/sub is a useful communication pattern for a range of applications, including group messaging, news feeds, and data synchronization. Publishers create message logs, which groups of trusted subscribers read at a later time. As long as clients and at least one server are uncompromised and running authentic versions of the software, Talek prevents a cloud operator from learning anything about the communication patterns of the users. Combined with encryption, developers conceal both the *contents* and *metadata* of users' application usage without losing the reliability and availability of the cloud.

Recent research has advanced both one-to-one private messaging [2, 3, 52, 59] and anonymous broadcasting [17–19, 42, 60]. These systems offer security guarantees rooted in k-anonymity [57], plausible deniability [36] or differential privacy [25, 26]. Talek focuses on a stronger security goal based on *access sequence indistinguishability*, where two arbitrary-length client access sequences are indistinguishable to the server, and thus the server learns no information about which users may be communicating. Existing systems guaranteeing indistinguishability are either impractical due to prohibitive network costs [5, 30, 32, 46], or are custom-tailored for specific applications [11, 35], limiting their applicability.

Talek provides a practical design for a general-purpose private pub/sub system with strong security goals based on indistinguishability of access patterns. It is designed to be network bandwidth efficient — usable with mobile clients reading and writing asynchronously to many pub/sub topics, each modeled by a message log.

Talek is based on private information retrieval (PIR) [16, 21, 29], but PIR by itself is not enough to support a a general-purpose private pub/sub system. We combine GPU-based performance improvements with two novel techniques:

- *Oblivious logging* describes a new way to construct a real-time pub/sub message broker that can deliver messages with provable unlinkability between publishers and subscribers.
- *Private notifications* allow subscribers to determine which topics have new messages without polling or revealing anything about their subscription list.

With oblivious logging, all clients issue identically sized random-looking read and write requests to servers at an independent rate. Within a group of clients reading and writing to a shared topic, a shared secret determines the pseudorandom and deterministic sequence of locations for topic messages. The topic writer places new messages in locations that appear random to the adversary. Any subscriber with the topic secret can follow the pseudorandom sequence, reading new messages without coordination with other users. Talek relies on private information retrieval (PIR) to read the message stored at a location without disclosing to the server which location is being read. We apply updates and reads consistently across PIR servers using timestamp ordering [8]. To support message asynchrony, we insert messages into a cuckoo hash table. This data structure densely packs messages and resolves collisions without disclosing information to the adversary. Each server only stores the latest n messages, purging older messages. Choosing a larger value for n means data is stored on the database for longer, at the cost of more expensive reads.

With private notifications, subscribers periodically retrieve a *global interest vector*, which privately and efficiently encodes the set of all topics with new messages. Subscribers use the global interest vector to locally prioritize reads. Servers maintain the global interest vector without leaking any information about its contents.

In our system, the developer chooses l independent servers to host replicas of the data. Talek's security model assumes at least one of the servers is honest. Our guarantees hold for arbitrary behavior by the other servers, who may collude, share secrets, and send faulty responses to clients. An adversary could control the network, all other clients, and $l-1$ servers without impacting the security of the system.

Talek does not guarantee liveness; a single faulty server can deny all use of the system in a way that is detectable to the developer and all clients. We expect developers to choose service providers with reputations for high availability. Because clients connect directly to Talek servers, we also do not hide when users are online. We expect the system to be used for communication among groups of trusted users, such as in messaging, productivity apps and games. If the shared topic secret is exposed to the adversary, subscriber anonymity is preserved, but publisher anonymity is not. Applications that require wide broadcasts to many untrusted subscribers (e.g. a public blog), are better served by anonymous broadcast [17–19, 60].

We have implemented Talek in Go and evaluated the system on a 3-server deployment using Amazon EC2. Our source code is public. Our evaluation shows that for a messaging workload where users send and receive 256-byte messages every 5 seconds, we can support 32,000 concurrent users sustaining a total throughput of 566,000 messages per minute with an average end-to-end latency of 5.57 seconds. Further, our design is compatible with horizontal scale-out to support higher message rates and/or more users, although this is left for future work. PIR-based reads are the primary bottleneck on total system performance, as the server-side computational cost of a single read operation scales with the size of the database. In all, we show that we can achieve 3–4 orders of magnitude better performance than comparable systems with the same security goals.

The paper highlights the following contributions:

- Oblivious logging is a new approach to achieving indistinguishability of access patterns, by efficiently storing logs of messages on the server in a way that looks random to an adversary. (Sections 4 and 5)
- Private notifications privately encode the set of new messages, helping clients prioritize reads. (Section 6)
- Implementation and evaluation of Talek, which applies these two techniques in an end-to-end pub/sub system with practical performance. (Sections 7 and 8)

2 Background

2.1 Publish-Subscribe Model

Publish-subscribe is a general messaging pattern, where messages are not programmed with specific receivers. Instead, users create topics and publish messages to these topics. A set of servers, called message brokers, store and forward messages to topic subscribers. Pub/sub systems are typically not used directly by end-users. Developers use pub/sub to share data in an application.

As a practical example, a group messaging application could use topics to store messages threads. Although Talek only supports single-writer many-reader logs, we can emulate many-to-many communication by creating a log for each writer. Users publish to their own log and each member of the group subscribes to all logs of the group. Pub/sub topics can also be used to model a log of operations to a shared calendar, a sequence of moves in a mobile game, or a private news feed. In practice, Talek can be linked to existing applications using interfaces similar to AWS Mobile SDK, Azure App Service, or Google Cloud SDK.

Talek associates a secret with each topic. Subscribers are granted access to topics by receiving a secret from the publisher. These secrets are shared using an in-band mechanism called *control logs*, which we elaborate in Section 5.3. A subscriber uses the secret to find and decrypt messages placed on the server by the publisher. If a topic's secret is shared with the adversary, publisher anonymity for that topic is compromised, but subscribers' anonymity and publisher anonymity for other topics are preserved. Talek is best suited for applications where publishers communicate with small groups of trusted subscribers. If subscribers cannot be trusted, a publisher can create individual topics for each subscriber at the cost of additional write overhead.

In our system, a Talek service can be shared across many applications, masking which application a user is accessing. Any particular instance of Talek is configured with a read rate, write rate, and message size. Thus, de-

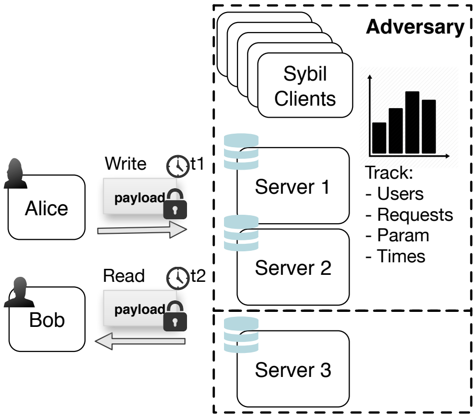

Figure 1: System and threat model in Talek. We assume the adversary can control all but one of l servers in the system ($l = 3$ in figure). Clients send network requests directly to the servers. Adversarial servers are free to record additional data, such as the source, type, parameters, timing, and size of all requests to link users who are likely to be communicating together.

velopers will choose an instance of Talek with configuration parameters appropriate to the application needs.

2.2 Threat Model

Figure 1 illustrates a system with mutually distrusting clients located across a wide-area network, sharing data through Talek services, each hosted in a unique data center. We use the term, server, as an abstraction of a unique Talek service controlled by an independent administrative domain.[1] The adversary's goal is to build a statistical model of users who are likely to be communicating.

Talek assumes the adversary controls all but one of a set of servers. Clients do not know which server is honest. The adversary can also control the network and generate an unbounded number of clients. We assume message storage capacity is scaled to the number of clients. We assume all servers are collecting information about all client network requests, such as the source, operation type, parameters, timing, and size of requests. The Talek protocol ensures correctness and unlinkability, even when adversarial servers and clients exhibit arbitrarily malicious behavior, such as if they collude, share secrets, and send faulty responses to clients. Our security guarantees must hold even as clients are observed over long periods of time.

While per-topic secrets are shared in-band, we assume clients who wish to share data know each others' long-term public keys. Talek is compatible with bootstrapping keys from existing applications [1], using identity-based encryption [9,10,43], or through an out-of-band channel.

Talek is designed to work with groups of trusted users. We assume that publishers trust their subscribers not to disclose topic secrets when they grant access to a topic. Generally, clients colluding with servers cannot expose any publishers or subscribers. However, malicious subscribers can collude with any server to expose only the publisher of a topic for which it has the topic secret.

During normal operation, all servers must be available and reachable by all clients. Any single server can deny all use of the entire system by refusing to respond or by responding with faulty information. However, this behavior is detectable to the developer and clients. We assume developers will choose services with high reputations for availability. While we do not discuss it in this paper, Byzantine fault tolerant variations of private information retrieval [21, 29] can be used for better liveness guarantees at the cost of higher overhead. Adversarial clients can degrade service in denial of service attacks.

We assume the existence of secure encryption, key-exchange protocols, signatures, hash functions, and random number generators. We also assume that server public keys is known to all users. These issues are orthogonal to the properties Talek is designed to provide [58].

2.3 Security Goals

Informally, our security goal is to *require that any subset of $l-1$ servers can learn nothing about the access pattern of any user*. We define our security goal of *access sequence indistinguishability* more formally in Appendix A.1. To the adversary, an idle user is indistinguishable from a user communicating with other users. In addition to indistinguishability between two access patterns by the same client, the adversary should not be able to determine which users may be communicating together. These properties must hold regardless of how long a client is observed. We cannot hide IP addresses or when a user is online or offline from the system, because users are directly interacting with the servers. However, this must not undermine our security goal.

Definition 1. *(Access Sequence Indistinguishability) We say that the system provides access sequence indistinguishability if for any polynomial-time probabilistic adversary, the adversary's advantage in the security game of Appendix A.1 is negligible in the security parameter.*

Access sequence indistinguishability provides one of the strongest definitions of privacy available. It is stronger than k-anonymity [57], where the adversary can narrow the user to one of k users. It is also stronger than plausible deniability [36], where information leakage is allowed up to a certain confidence bound. Most systems that rely on these assumptions are vulnerable to statistical attacks that deanonymize users as they use the system over long periods of time.

Let seq denote a data access sequence by a client:

$$seq := [(\tau_1, op_1, k_1), \ldots, (\tau_n, op_n, k_n)]$$

[1]Our design allows each independent server to be implemented across multiple machines for scalable performance and fault tolerance, but that is beyond the scope of this paper.

where each τ_i denotes the time that an operation op_i is made, which can be a `Publish` or `Poll` to topic, k_i. Given a data access sequence, the client generates a series of network requests to the servers. Let

$$events := [(\tau_1', req_1, par_1), \ldots, (\tau_n', req_n, par_n)]$$

denote the sequences of events seen by a server from the client. Network request req_i arrives at time τ_i' with parameters par_i.

Informally, we achieve our security goal by designing the system such that

1. The schedule of requests seen by the server, $\{\tau_i', req_i, par_i\}$, is independent from the data access sequence, seq. Requests are made by all clients at an independent rate regardless of whether the client actually needed to perform a `Read` or `Write`.
2. All parameters, par_i, look random from the perspective of any $l-1$ set of servers. Dummy request parameters look indistinguishable from parameters of legitimate requests.

2.4 System Goals

In order to be practical for modern workloads, Talek must also satisfy the following goals:

Scalable: The system should support large numbers of ephemeral clients over a wide-area network, comparable to the application workloads supported by other privacy-preserving systems.

General Purpose: The system should be able to support a wide-range of pub/sub workloads, from point-to-point messaging to one-to-many news feeds.

Low Latency: High-priority messages should be delivered in seconds, in order to support messaging.

2.5 Private Information Retrieval (PIR)

Talek uses the privacy guarantees of PIR in the context of a general-purpose pub/sub protocol. PIR allows a single client to retrieve a block from a set of storage replicas without revealing to any server the blocks of interest to the client. There exist two major categories of PIR techniques, computational PIR (C-PIR) [41] and information-theoretic PIR (IT-PIR) [16, 21, 29]. Talek is compatible with both varieties. However, C-PIR has been shown to be orders of magnitude more expensive in computation and network usage, making it impractical for the types of application workloads we target. This trade-off comes with the benefit of supporting a stronger threat model. With C-PIR, privacy is preserved even when all servers collude. In this paper, we focus on IT-PIR due to its better performance.

In order to gain an intuition for the performance and cost of IT-PIR, we illustrate the protocol with an example (Figure 2). Let l represent the number of servers in the system, each storing a full copy of the database, partitioned into equal sized blocks. While IT-PIR generalizes to arbitrary numbers of servers and blocks, the example in Figure 2 contains $l = 3$ servers and $n = 3$ blocks ($\{B_1, B_2, B_3\}$).

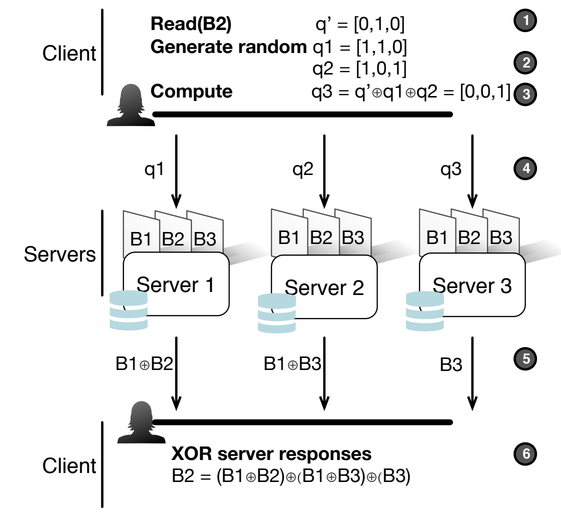

Figure 2: Information-theoretic PIR example. Each client sends a random request vector to each server, except for one, which receives the XOR of the other random requests plus the true request. Each server responds with data equal in size to a single data block. As long as some server does not collude, the remaining servers cannot determine q', which of n blocks the client is retrieving.

1. Suppose a client wanted to read the second block, $\beta = 2$, encoded by the bit vector, $q' = [0, 1, 0]$, which consists zeros and a one in position β.
2. The client generates $l-1$ random n-bit request vectors, q_1 and q_2.
3. The last request vector is computed by taking the XOR of the vectors from (1) and (2), $q_l = q' \oplus q_1 \oplus \ldots \oplus q_{l-1}$.
4. The client then sends q_i, to server i for $1 \leq i \leq l$. Because request vectors are generated randomly, this reveals no information to any collection of $< l$ colluding servers.
5. Suppose the server receives $q_i = [b_1, \ldots, b_n]$ and B_j represents the j^{th} block of the database. Each server computes R_i, the XOR of all B_j for which $b_j == 1$ and returns R_i to the client.
6. The client restores the desired block, B_β, by taking the XOR of all R_i. $B_\beta = R_1 \oplus \ldots \oplus R_l$

IT-PIR has desirable network properties: a client sends one request vector to and receives one block from each server. These requests and responses appear random to the network and the servers, assuming at least one server is honest. IT-PIR can be a computationally expensive operation, as the computational cost for a read request scales with the size and number of blocks in the system. The size of a client request scales with total number of blocks and the client work scales with the number of servers. IT-PIR also requires consistent snapshots across servers, with equal sized blocks in the data structure.

While PIR allows a client to read privately, in a read-/write system it is equally important to allow clients to write privately. Ostrovsky and Shoup introduced the notion of private information storage (PIS) [47], which allows a client to write to a row to a database of n rows without revealing which row was updated with poly-logarithmic communication complexity. In contrast, Talek aims to construct a pub/sub read-write mechanism with writes that take $O(1)$ time. We answer the following questions in designing our pub/sub system:

- *Random writes*: How can publishers write in a way that appears random to the server? (Section 4)
- *Consistent Snapshots*: How do we maintain consistent snapshots across servers despite updates, for PIR operations to work over? (Section 4)
- *Garbage collection*: How do we constrain database size, keeping PIR operations tractable? (Section 4)
- *Zero coordination*: How do subscribers leverage PIR without coordinating with the publisher? (Section 5)
- *Notifications*: How do we minimize the need to poll for new data? (Section 6)

3 Design Overview

Client Overview: Our system achieves our security goal by *requiring all users to behave identically from the perspective of any colluding set of $l-1$ servers*. Figure 3 illustrates how the system is organized; Figure 4 enumerates the interfaces and client/server state; Figure 5 lists constants that parameterize the design. Developers link their application to the Talek client library, calling `Publish` and `Subscribe` on the *client developer interface (CDI)*. When a function is called on the CDI, Talek places it on an internal *request queue*, which gets translated into privacy-preserving `Read` and `Write` network requests by the *network protocol interface (NPI)*.

Every user issues equal-sized requests for each operation on the NPI (e.g. `Read` and `Write`) at an independent rate, potentially issuing a dummy request if the respective request queue is empty. The key constraint is that the distribution of network requests from a device is independent from that user's real usage. In this paper, we describe the system using fixed rates of periodic requests for convenience. In practice, the rate should be sampled from a Poisson distribution based on the global application usage.

A dummy request, including its parameters and payload, must be indistinguishable from a legitimate request. Messages are encrypted with a CCA-secure encryption scheme [7] to provide confidentiality and authenticity. Thus, only the access pattern and not the contents of communication is disclosed when all servers collude. We define a globally-fixed message size, z, to which messages are split and padded to fit. In practice, an application might run two parallel instances of the Talek protocol, one for text-based data, and one with higher latency for images. Because we expect Talek to be used with mobile and web applications, we can take advantage of pre-existing data types specified in the application to facilitate such categorization.

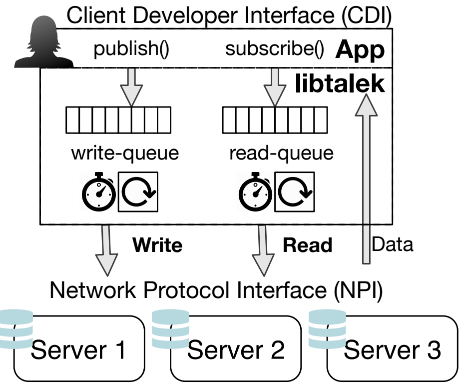

Figure 3: Overview of the Talek architecture. All clients must behave identically from the perspective of any $l-1$ servers. Any calls by the application to publish or subscribe is internally queued by the client library, which is then translated into a privacy-preserving network request. The client library independently issues requests with equal-sized parameters and messages that appear random to the adversary.

Client Developer Interface (CDI) `Publish`(topic, message) `Subscribe`(topic)
Network Protocol Interface (NPI) `Write`(bucket1, bucket2, encryptedMsg, interestVector) `Read`([]requestVectors) → encryptedData `GetUpdates`() → globalInterestVector
Client State • *topics* - List of subscribed topics • $writeQueue$ - Queue of write operations • $readQueue$ - Queue of read operations
Server State • *log* - Global log of write operations • *table* - Blocked cuckoo hash table

Figure 4: Summary of Talek interfaces and client/server state

Server Overview: The server is designed to store a limited set of messages in order to allow asynchronous senders and receivers to be decoupled in time, rather than participating in synchronous rounds of communication. Because the cost of PIR operations scales linearly with the size of the database, for good performance we fix the number of messages stored on the server to n, garbage collecting the oldest. Thus, n is directly related to the time-to-live, TTL, for a message, which dictates how tightly synchronized senders and receivers need to be. As the number of clients in the systems grows, the system must use larger values of n to support the same TTL.

In order to efficiently pack these messages into a dense data structure that is compatible with PIR, we store messages in a blocked cuckoo hash table [22], where each of the b buckets stores a fixed number of messages, d. Client `Write` requests explicitly specify two pseudo-randomly chosen buckets in which messages can be inserted, potentially resulting in cuckoo evictions (table rearrangement) if both buckets are full. In a `Read` request, the hash table is treated as a PIR database with each hash bucket as an entry. The client uses PIR to retrieve an entire hash bucket without revealing to the server which bucket it retrieved. Each server stores a consistent replica of the hash table to participate in the PIR protocol.

Blocked cuckoo hashing has a number of desirable properties for our system. Compared to chained hash tables, buckets have equal fixed size, a necessary requirement for PIR. Each message is stored in one of two buckets. To handle collisions, the size of the table must be larger than n by a small overhead factor (generally less than 20% for reasonable values of the bucket size d).

A client issues at most two `Read` requests to check both buckets where a message could be stored. If the client finds the message it is looking for in the first bucket, then it can use its next `Read` request for another task, rather than querying the second cuckoo hash location. From the server's perspective, the client is simply issuing a stream of opaque PIR requests.

Topics Overview: In order for publishers to write a series of messages without online coordination with subscribers, oblivious logging is used to hide messages within a stream of apparently random writes. This log is defined by a secret topic handle. Exposure of the topic handle (e.g., by an untrustworthy subscriber) would expose the publisher's write pattern, but not subscriber consumption. The topic handle is used with a pseudorandom function family, PRF, to generate a deterministic sequence of buckets, called a *log trail*. The publisher stores encrypted messages along the log trail. Subscribers use PIR to retrieve messages following the same sequence.

In order to avoid the need to poll for new messages, private notifications (§ 6) assist subscribers in knowing when to read. With each write, clients submit a Bloom-filter-based *interest vector*, which privately encodes the topic and sequence number of the message. Servers combine the interest vectors of all messages currently in the database to form a *global interest vector*, which privately encodes which messages are currently stored on the server. Clients periodically retrieve this global interest vector, which let them skip reading buckets with no new messages. Clients read and write on their independent schedules, which is not changed by information from this vector.

The next few sections describe each aspect of the system more formally. We first consider how Talek works with m idle online clients and l servers, illustrating the data structures, network requests, and a framework for security (Section 4). Then, we expand on foundation to hide legitimate traffic among requests using oblivious logging (Section 5) and private notifications (Section 6).

Globally Configured		
l	constant	Number of servers
n	constant	Number of messages stored on server
b	constant	Number of server-side buckets
d	constant	Depth of a bucket
z	constant	Size of a single message
w	constant	Per-user rate of writes
r	constant	Per-user rate of reads
Dynamically Measured		
m	variable	Number of online clients
TTL	$n/(m*w)$	Lifetime of a message on the server
$load$	$n/(b*d)$	Load factor of the server hash table

Figure 5: Variables in the system, including those configured by the developer and dynamic behavior measured at run-time.

4 Talek with Idle Users

Online clients issue dummy `Read` and `Write` requests at fixed rates of r and w respectively. We choose an arbitrary server to be the *leader*, $\mathscr{S}_0$, with the rest of the servers forming the *follower* set, $[\mathscr{S}_1, \dots, \mathscr{S}_{l-1}]$. All `Read` and `Write` requests are directed to the leader and forwarded down the chain of followers.

Talek is further configured with a window size, n, such that messages older than the most recent n are garbage collected and deleted. It is possible for clients to miss a message if they fall behind and it is garbage collected. In this case, subscribers can request retransmissions as described in Section 5.3. We show detailed pseudocode for the server in Appendix A.2.

4.1 Cryptographic Assumptions

Each server has a public-private key pair, pk, sk, generated using an algorithm $PKGen()$. We assume the public key of each server is known to all clients. We write $PKEnc_{pk}(text)$ for the encryption of $text$ under pk, and $PKDec_{sk}(cipher)$ for the decryption of $cipher$ under sk. Clients also have access to an efficient symmetric encryption scheme that provides *authenticated encryption with associated data* (AEAD). The associated data is authenticated, but not included in the ciphertext. We write $Enc_k(text, ad)$ for the encryption of $text$ with key k and associated data ad, and $Dec_k(cipher, ad)$ for the decryption of $cipher$. Our implementation uses an IND-CCA2 [7] RSA encryption scheme and AES-GCM for symmetric encryption. Let $PRF(key, input)$ denote a pseudorandom function family and $PRNG(seed)$ denote a cryptographically secure pseudorandom number generator. For the purposes of this description, let $|$ denote tagged concatenation.

4.2 Strawman: Chained Hash Tables

We first consider a strawman approach to designing the server. Clients periodically write into pseudo-random positions on the server. Suppose we model the server's state as a table of b buckets, and clients explicitly place

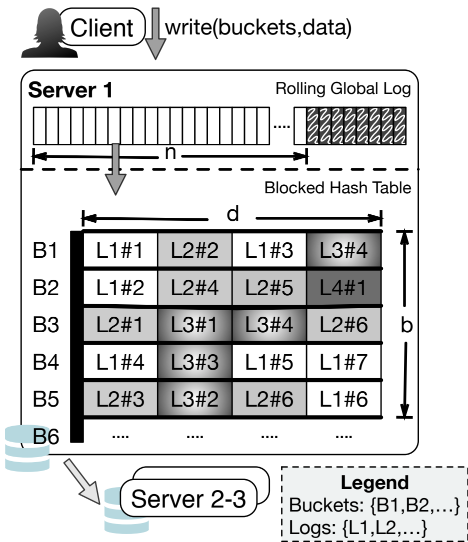

Figure 6: Oblivious logging data structures and workflow. The leader serializes all write operations into a global log, assigning each message a globally unique sequence number. Server-side state is replicated to all other servers from the leader. In order to garbage collect old messages, we keep only the latest n messages. Client writes specify two random buckets in which the message can be placed, forming a blocked cuckoo hashing scheme. Topic logs are spread across the hash table, which can be read by the subscribers using PIR.

writes in a bucket. To make reading oblivious to the server, clients would use PIR to retrieve an entire bucket. To prevent collisions, the server must have a mechanism for handling collisions.

One way to deal with collisions is to use chaining, where each bucket is a linked list of values. Because PIR requires elements of equal size, buckets would need to be padded to the length of the largest bucket. In the worst case scenario, one bucket could contain the entire database.

4.3 Write: Cuckoo Hashing

Talek organizes server-side state into a blocked cuckoo hash table [22, 48], where each server's storage is organized into b buckets, each bucket storing d messages, each of size z. PIR requests fetch an entire bucket of size $d \cdot z$. Figure 6 illustrates the server-side data structures. Cuckoo hashing has a number of desirable properties for PIR-based reads. In practice, the number of messages stored, n, is chosen as a fraction of the capacity of the cuckoo table, $b \cdot d$. This fraction is set to ensure with high probability, that a message will fit with minimal rearranging of the cuckoo table [22].

- PIR requires buckets be of equal size. Talek's blocked cuckoo hash table is configured with a fixed depth, d.
- Individual PIR operations are relatively expensive. Cuckoo hashing bounds the maximum number of client probes to 2.
- The cost of a PIR request scales linearly with the size of the database. Cuckoo hashing enables dense placement of messages in a pre-allocated data structure with minimal wasted space.

Because cuckoo hashing is a random algorithm and PIR requires consistent replicas across all servers, a shared random seed enables all servers to achieve identical state as long as items are inserted in the same order. The leader assigns each incoming request a global sequence number for consistent ordering.

Client `Write` requests are generated using the following protocol. A client, $\mathscr{C}$, periodically issues random `Write` requests to the server. $\mathscr{C}$ is preconfigured with a randomly chosen k_{idle}, which is used to generate the i-th random number by $PRF(k_{idle}, i)$, and an idle encryption key k_{enc}. Similarly, all servers share a key k_{cuckoo}, used to generate random values in the cuckoo algorithm below. The i-th client request is generated as follows:

1. $\mathscr{C}$ chooses two random buckets,
$$\beta_1 = PRF(k_{idle}, i|1) \mod b$$
$$\beta_2 = PRF(k_{idle}, i|2) \mod b$$
where b is the number of buckets.
2. $\mathscr{C}$ encrypts a random z-length bit-string,
$$data = Enc_{k_{enc}}(PRF(k_{idle}, i|3) \mod 2^z)$$
and submits $\beta_1|\beta_2|data$ to the leader, $\mathscr{S}_0$
3. Upon receiving the request, $\mathscr{S}_0$ forwards the request to all other follower servers, $\mathscr{S}_1 \ldots \mathscr{S}_{l-1}$, each following the cuckoo algorithm in steps 4–7.
4. Each server deletes the n-th oldest element.
5. The server inserts $\beta_1|\beta_2|data$ into the bucket at either β_1 or β_2 if there is spare capacity in either bucket.
6. If both buckets are full, choose $\beta_e \in \{\beta_1, \beta_2\}$, using randomness from k_{cuckoo}. Let $\delta_e = \beta_1|\beta_2|data$.
7. Repeat the following until all values are inserted
   (a) Try to insert δ_e in β_e if the bucket has space.
   (b) If not, randomly evict an entry in β_e and insert δ_e in its place.
   (c) Let δ_e equal the evicted value and β_e equal its alternate bucket location.

Correctness: The leader is only responsible for assigning a global sequence number, which does not affect security or correctness. If the leader misrepresents the global sequence number of a message (e.g. by giving a different sequence number to different follower servers), it could cause those replicas to become inconsistent. Because any follower could also deny service by failing to respond, the leader is in no more privileged a position to affect correctness or liveness of the system. In Section 5.3 we describe how clients detect misbehavior.

Performance: Cuckoo tables have a maximum capacity that is lower than the size of the table, $b \cdot d$. The ratio of the maximum capacity of the cuckoo table to the allocated space is known as the *load factor*, a function of the bucket depth, d. The load factor grows asymptotically towards 1 as d increases [22].

In Talek, the number of buckets b and the depth of each bucket d are tuned to the client workload. Clients issue `Reads` with a random b-bit request vector and receive a $O(d)$-sized bucket in response. A smaller value for b and higher d allows the developer to more densely pack the hash table and use smaller request vectors in PIR requests at the cost of larger network overhead. This configuration lends itself to frequent writes, common in chat applications. Conversely, a high value of b and low value of d resembles a traditional cuckoo hash table, resulting in a lower load factor, but better bandwidth utilization. This configuration is appropriate for infrequent writes of large messages, such as for images.

4.4 Read: Serialized PIR

In order to reduce the network costs of `Read` requests to the client, we use a *serialized* variation of PIR, which offloads work from clients to the leader. The client sends a single request to the leader, containing PIRs for each follower, and receives a single response from the leader. We use one-time pads to preserve the confidentiality of each server's results, while allowing the leader to combine them on behalf of the client. In contrast, traditional PIR requires that the client receive messages from each server, and locally calculate the result.

A client, $\mathscr{C}$, periodically issues random PIR requests to the server as follows:

1. $\mathscr{C}$ chooses a random bucket to read and generates b-bit PIR requests for each server, $\{q_0, \ldots, q_{l-1}\}$, where b is the number of server buckets.
2. $\mathscr{C}$ generates a high-entropy random seed for each server, $\{p_0, \ldots, p_{l-1}\}$
3. $\mathscr{C}$ encrypts each server's parameters with its respective public key and generates a `Read` request, $PKEnc_{pk_0}(q_0|p_0), \ldots, PKEnc_{pk_{l-1}}(q_{l-1}|p_{l-1})$
4. $\mathscr{C}$ sends this request to the leader, $\mathscr{S}_0$, who forwards it to the remaining follower servers.
5. In parallel, each server, $\mathscr{S}_i$, decrypts its respective PIR request vector, q_i and computes its response, R_i.
6. Each server, $\mathscr{S}_i$, also computes a random one-time pad, $P_i = PRNG(p_i)$, from the seed parameter. This one-time pad should be of equal size to R_i.
7. Each server, $\mathscr{S}_i$, responds to the leader with $R_i \bigoplus P_i$.
8. $\mathscr{S}_0$ combines the server responses and responds to $\mathscr{C}$ with $R_0 \bigoplus P_0 \bigoplus \ldots R_{l-1} \bigoplus P_{l-1}$
9. $\mathscr{C}$ restores the bucket of interest by XOR'ing this response with each server's one-time pad, $P_0 \bigoplus \cdots \bigoplus P_{l-1}$

Security: This serialized variant of PIR is functionally equivalent to the traditional PIR scheme described in Section 2.5. As long as the adversary only has access to $l-1$ servers' secret keys, it cannot decrypt the honest server's request vector and reconstitute the secret request. Similarly, each response is combined with a random one-time pad, which prevents the adversary from learning any information from any individual server's response. Because each server's one-time pad is computed from a shared secret with the client, the client can recover the underlying value.

Theorem 1. *Serialized PIR (Informally)*

1. *Security: As long as there exists at least one server's secret key that is unknown to the adversary, the adversary learns nothing of the user's secret request.*
2. *Correctness: The client receives the contents of the bucket corresponding to its request.*

4.5 Security Analysis of Idle Sequences

By definition, the access sequence, seq_i, of each client is a null list. For each client, the adversary observes $events_i$, a log of randomly generated requests. In the protocol as described thus far, $events_i$ is completely independent from seq_i. While we use fixed rates for `Read` and `Write` for convenience, our security goals are met as long as the rates are independent. For example, if requests follow a Poisson distribution between the hours of 9am and 5pm for every user, our security properties still hold. In the next section, we discuss how legitimate accesses are hidden among this cover traffic in a way that is indistinguishable to the adversary.

5 Oblivious Logging

The goal of *oblivious logging* is to translate secret calls to `Publish` and `Subscribe` on the client developer interface (CDI), into random-looking `Write` and `Read` network requests on the network protocol interface (NPI). Critically, these `Write` and `Read` requests must look indistinguishable to the adversary from the cover traffic described in Section 4. In this section, we describe *topic logs*: single-writer, many-reader logs stored on Talek servers. A topic log is only ever written by its creator, but it may be read by many clients. We show detailed pseudocode for the client in Appendix A.2.

5.1 Topic Handles and Messages

When a user creates a new topic log, Talek generates a *topic handle*, which contains a unique ID, id, encryption key, k_{enc}, and two seeds, k_{s1} and k_{s2}. The topic handle is a shared secret between the publisher and subscribers of a topic. All messages are encrypted with k_{enc} using a CCA-secure symmetric encryption scheme, $Enc_{k_{enc}}(message)$. We further assign all messages in a topic log a sequence number, $seqNo$. The two seed values are used in conjunction with a pseudorandom function family, $PRF(seed, seqNo) \in \{0 \ldots (b-1)\}$, to produce two log trails, unique and deterministic sequences of bucket locations for writes. Similar in nature to frequency hopping [24, 27], topic handles allow publishers and subscribers to agree on a pseudorandom sequence of buckets without online coordination.

5.2 Scheduling Requests

When a publisher wants to publish a message, M, with sequence number $seqNo$ to a topic, the Talek client library does the following:

- On the next periodic random `Write` request, $\beta_1|\beta_2|data$, replace its parameters with the following:

$$\beta_1 = PRF(k_{s1}, seqNo) \mod b$$
$$\beta_2 = PRF(k_{s2}, seqNo) \mod b$$
$$data = Enc_{k_{enc}}(M)$$

When a subscriber reads the next message in a topic at sequence number, $seqNo$, they do the following:

1. On the next periodic random `Read` request, replace it with a PIR read to the first bucket,
$$PRF(k_{s1}, seqNo) \mod b$$
2. If the returned bucket is missing the message, on the following periodic random `Read` request, replace it with a PIR read to the second bucket,
$$PRF(k_{s2}, seqNo) \mod b$$
3. Attempt to decrypt every message in the bucket using $Dec_{k_{enc}}(M)$ and return the result if found

For both `Write` and `Read`, legitimate requests must follow the same periodic schedule as when idle in Section 4. To the adversary, writes look indistinguishable from the idle case. Topic handles allow subscribers to find the latest content with zero coordination. Because reads are done with PIR, many subscribers can read the same log or poll the same bucket repeatedly without revealing any information.

5.3 Control Logs

In order to facilitate control messages between users, we automatically generate a *control log* between every pair of users that share at least one topic. We expect the topic handle for the control log to be generated and exchanged out of band when public keys are exchanged and verified. The control log is used by the Talek system to send retransmission requests, bootstrap new topic logs by exchanging new handles, and other control messages to coordinate between users.

Clients also use a control log to periodically send low-priority messages to itself. If these messages are lost, it serves as a hint to the client of a denial of service attack. A limitation of our work is that it does not give clients the ability to determine which server is misbehaving.

5.4 Security Analysis

We more formally prove the security of oblivious logging in Appendix A.1. Informally, we prove the security by reduction to cryptographic assumptions. For `Read`, we rely on the security properties offered by PIR. PIR queries that correspond to legitimate requests are indistinguishable from a PIR query for a random item [16]. For `Write`, we rely on the security properties of a PRF and our encryption algorithm. We use an IND-CCA secure encryption algorithm for message payloads. For any `Write`, the bucket locations are generated by a PRF, using either the topic handle's seed values, (k_{s1}, k_{s2}), or the idle seed, k_{idle}. In both cases, the output is indistinguishable from a random function. [31].

A malicious client still has the ability to try to deny service by deviating from the protocol and choosing a fixed bucket to DoS. DoS is limited by the fixed `Write` rate per client, the number of Sybil clients, and the size of the database, n. We rely on the self-balancing nature of cuckoo tables, where legitimate messages can be evicted to their alternate locations.

We assume the publisher trusts their subscribers. Topic handles can expose a publisher if shared with adversarial servers, by observing the users that write to a known log trail. Even so, PIR protects the privacy of subscribers. Publishers must only use each output from $PRF(seed, seqNo)$ for any sequence number once.

6 Private Notifications

Regular polling in Talek presents two problems. First, because every user polls at the same rate to meet our security goal, message latency gets worse as the user subscribes to more topics. Second, it is hard to know which topic should be polled at any given time.

We introduce a private notification system that allows users to efficiently determine when new messages have been published to a topic without revealing their topic list. By detaching reads from notifications, clients can prioritize reads and reduce how often they read buckets with no new messages.

6.1 Computing a Global Update Set

With every `Write` request, the client computes a Bloom filter, called an *interest vector*, encoding the topic ID and sequence number of the message being written,

$$v.insert(topicId|seqNo) \rightarrow [b_0 \ldots b_{|v|}]$$

If we use a large random topic ID and h cryptographically-secure hash functions (modeled as random oracles) in the Bloom filter, then the adversary has a negligible advantage in learning the client's input given the interest vector. Because the server can expect every interest vector to contain only one element, it can also filter any malicious interest vectors where there are more ones than hash functions used, $\sum_{i=0}^{|v|} b_i > h$. If the `Write` request is a dummy request, we randomly choose h bits to set to 1, with the remaining bits set to 0.

Servers maintain a *global interest vector*, computed by taking the union of all *message interest vectors* of messages stored on the server. This data structure efficiently encodes the topic and sequence numbers of every message on the server without revealing anything to the server. Clients periodically query for the global interest vector using `GetUpdates`, allowing each client to independently determine which topics have new messages.

6.2 Security Analysis

The security of private notifications relies on the cryptographic hash functions used in the Bloom filter. As long as we use a topic ID with sufficient entropy, each interest vector provides a negligible advantage in the indistinguishability security game. We describe this in more detail in Appendix A.1.

Private notifications are only used to prioritize reads on the internal request queue. As such, it has no impact on the security goals of oblivious logging. It simply reorders the schedule of private requests.

Because private notifications are a parallel mechanism to oblivious logging, servers could manipulate the vector to influence clients. In order to detect server misbehavior, clients can retrieve the global interest vector from every server, and ensure they are consistent.

7 Implementation

To demonstrate that Talek is practical, we implement a prototype in approximately 2,800 lines of code; the source code is online. We implement two versions. The first, written in Go, runs entirely on the CPU. The second offloads PIR operations to the GPU using a kernel written in C on OpenCL. We wrote Go language bindings to share memory between the CPU implementation and the GPU. The prototype uses SipHash [6] as the pseudorandom function, RSA for public-key encryption, and AES-GCM for symmetric encryption.

8 Evaluation

Our evaluation addresses the following questions:

- What is the cost per operation for clients and servers?
- How does system performance scale with more users?
- How does Talek compare with previous work?
- What is the end-to-end latency of messages?

8.1 Setup

All experiments are conducted on Amazon EC2 P2 instances. These virtual machines are allocated 4 cores on an Intel Xeon E5–2686v4 processor and 61 GB of RAM. They also include an NVIDIA K80 GPU with 2496 cores and 12 GB of memory.

We use 3 servers; One is chosen as the leader and the others are followers. We evaluate the system with messages sizes of 256 bytes and 1KB. We allocated an additional two VMs to run user clients. Each user client issues periodic `Read` and `Write` requests to the server. Note that the cost of a dummy request is identical to the cost of a legitimate request from the perspective of the server. Thus, we simply vary the global client read and write rates to simulate different application workloads.

	Messages on Server (n)	
	100K	1M
Client-side CPU costs		
Generate new topic handle	7753 μs	
Publish (1 KB messages)	70.5 μs	840.3 μs
PIR query	574 μs	6888 μs
PIR response (256 B messages)	36.5 μs	
PIR response (1 KB messages)	146 μs	
Server-side CPU costs		
PIR Read: CPU (256 B messages)	34.1 ms	271 ms
PIR Read: CPU (1 KB messages)	136 ms	1126 ms
PIR Read: GPU (256 B messages)	2.28 ms	15.4 ms
PIR Read: GPU (1 KB messages)	6.93 ms	74.2 ms
Write (256 B messages)	22.5 μs	22.6 μs
Write (1 KB messages)	21.1 μs	25.0 μs
Server-side storage costs		
256 B messages	60.4 MB	604 MB
1 KB messages	241 MB	2.41 GB
Network costs		
GetUpdates	59.9 KB	599 KB
Read request	9.39 KB	93.7 KB
Read response (1 KB messages)	4.16 KB	
Write request (1 KB messages)	1.08 KB	

Figure 7: Cost of individual operations in Talek. We vary the number of messages stored on the server, n, and message sizes (256 B and 1 KB). For values that do not depend on the number of messages on the server, we write a single value in the center column.

We validate that retrieved data is the same as what was put on the client's test driver.

While our experiments are run in a single data center, we expect the performance to be similar for a more realistic cross-data center setting. This would incur higher network latency, both to reach the leader and in communicating between servers. However, this setup does allow us to focus on the main bottleneck, the server-side computational cost of Talek.

8.2 Cost of Operations

To understand Talek's costs, we benchmark different components of the system. Each value is the average of 200 runs. We vary the size of messages between 256 bytes and 1 KB, and the number of messages, $n \in \{10K, 100K, 1M\}$. We fix the bucket depth in the blocked cuckoo table to 4, such that clients retrieve 4 messages at a time. This depth allows the cuckoo table to support a load factor of 95%. The number of buckets is chosen to hold n messages at the maximum load factor for the table. Figure 7 highlights the results.

In general, client costs are low due to IT-PIR. Each write encrypts the message and uses a PRF to determine the bucket location. The cost of `Publish`ing rises with the database size due to the Bloom filter. The cost of generating a PIR query also increases with the database size. Larger values of n translate to more buckets and larger PIR request vectors.

For the server, we implement two versions of IT-PIR. Our CPU implementation is primarily bottlenecked by

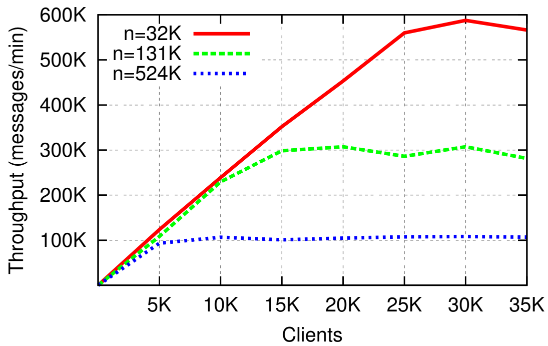

Figure 8: Throughput of the system when varying the number of real clients. Each client issues read and write requests every 5 seconds independently. Each line represents a different value for n, the number of messages stored on the server. For larger values of n, each read requires scanning a larger table, resulting in lower throughput.

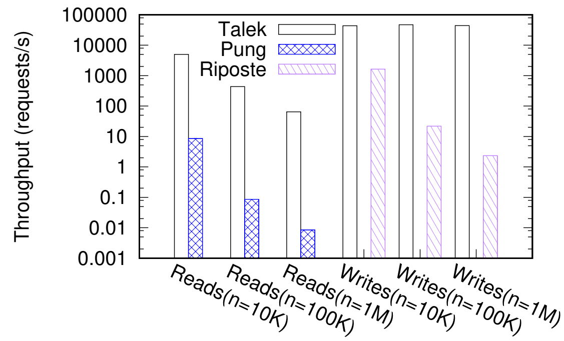

Figure 9: Performance comparison of `Read` and `Write` handlers of various systems. Pung uses C-PIR, which imposes a computational cost that is 3–4 orders of magnitude higher than our IT-PIR implementation. Riposte uses a write mechanism based on a reverse variant of PIR which incurs $O(\sqrt{n})$ cost compared to Talek's $O(1)$ writes.

memory bandwidth to the CPU. The GPU implementation accelerates performance by 1–2 orders of magnitude by taking advantage of the inherent parallelism of PIR operations across many GPU cores and the optimized on-device memory hierarchy. Writes incur negligible cost compared to the cost of reads. In our current implementation, we store all messages twice in memory. Writes are applied to the working copy stored in DRAM. Periodically, a snapshot of this state is copied into the GPU. Read requests are batched and forwarded to the GPU for processing. The leader is free to reorder reads without violating serializability.

Network costs between client and server are minimal. Clients must submit a read request containing a b-bit vector for each server. The size of `Read` responses and `Write` requests are within a small factor of the message size. The global interest vector returned from `GetUpdates` grows linearly with n in order to preserve a fixed false positive rate of 0.1. This cost is independent of the message size, such that its relative cost is lower for larger messages. In choosing a size for the Bloom filter, we trade off bandwidth with the false positive rate. The network costs per operation are identical between servers, as both `Read` and `Write` operations simply relay from the leader to followers.

8.3 Throughput

To understand Talek's peak performance, we conducted an experiment with a simulated messaging workload. Each client sends a message every five seconds, and receives a message every five seconds. For each data point, we spawn a number of clients and measure the leader's response rate over 5 minutes, giving the system enough time to reach steady-state performance. Writing in Talek is cheap, so we limit our workload such that each written message must be read before being garbage collected. If writes were not throttled, servers could easily accommodate a higher write throughput, while reads are bottlenecked by PIR computation.

Figure 8 shows the results for three values of $n \in \{32K, 131K, 524K\}$, the number of messages stored on the server. For small numbers of clients, the server achieves linear growth in throughput, demonstrating that the PIR operations are keeping up with read requests. The throughput is bottlenecked by the GPU's PIR process. Smaller values of n correspond to a smaller cuckoo table, resulting in cheaper PIR operations and higher throughput. We only evaluate the system with numbers of clients, m, such that $m < n$, corresponding to a message lifetime of at least one round of reads.

8.4 Comparison with prior work

In order to understand the relative performance between Talek with prior work, we benchmarked the `Read` and `Write` mechanisms. Figure 9 shows the relative throughput of each system.

Pung [5] is a read/write key-value based on computational PIR (C-PIR). Pung has a stronger threat model than Talek, where all servers are assumed to be untrusted. Pung uses an implementation of C-PIR called XPIR [4] for reads, which we compare to our IT-PIR implementation. For 256 B messages, XPIR took $117ms$ and $11.55s$ per message for table sizes of $10K$ and $100K$ respectively. Because the memory on the test machine was exhausted at $n = 200K$, we estimate that XPIR would take at least $100s$ for $n = 1M$ if the cost were to increase linearly. XPIR leverages lattice-based homomorphic encryption and efforts to build special-purpose cryptoprocessors [50, 51] have yielded at most a 10x performance boost. This still leaves a computational cost that is 3–4 orders of magnitude higher than Talek, limiting its practicality for real-world use. In contrast, Talek is easily parallelizable and accelerated using commodity GPUs.

Pung has higher network overhead that Talek. Pung uses an interactive binary search algorithm for retrieval, requiring $O(log(n))$ round trips between client and

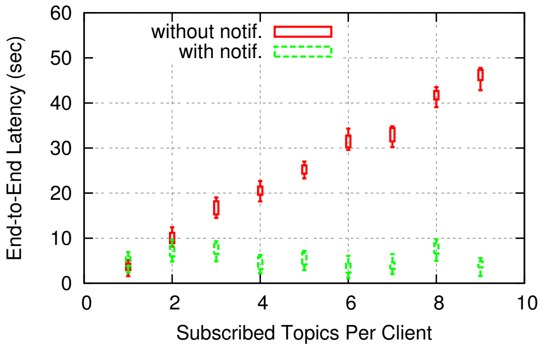

Figure 10: End-to-end latency of message delivery when a client is subscribed to multiple topics. Each data point represents 20 trials. With notifications, clients can prioritize topics with new messages.

server, compared to the $O(1)$ cuckoo table lookup in Talek. Thus, to retrieve a 1KB message from a database size of $n = 32K$ messages, Pung requires 15 rounds of PIR requests and $36.7MB$ of data, while Talek makes 2 requests and transfers $< 12KB$. Even if Pung used IT-PIR, their read/write protocol would be impractical for mobile workloads.

Riposte [17] is an anonymous broadcast system that uses PIR in reverse to anonymize writes to a database. Riposte has a weaker security goal based on anonymity within a round of communication and does not offer privacy over multiple rounds of communication. The Riposte implementation does not include an implementation for reads. Riposte writes incurs $O(\sqrt{n})$ cost compared to Talek's $O(1)$ writes.

Vuvuzela [59] has a weaker security goal based on differential privacy and noise but better performance. It scales to millions of users with a peak throughput of nearly 4M messages/min using the same number of servers as Talek. Although we have not implemented sharding, Talek is designed to be horizontally scalable to allow system throughput to increase by spreading buckets across servers and then combining the results. In this case, the leader and followers each consists of r replicas. Writes must be replicated to every replica server; however, this is acceptable because writes are inexpensive. Reads only require participation from one server in each replica group, which allows the system to scale for read-heavy workloads, similar to Vuvuzela.

8.5 End-to-End Latency with Notifications

In order to understand the latency of message delivery, we used the same messaging workload as in the throughput experiment, each user client sending and receiving messages every 5 seconds. Two additional clients are created, a sender and a receiver. We measure the end-to-end time for a message published by the sender to be seen by the receiver, varying the number of topics to which the receiver client is subscribed. The spread of each value over 20 trials reflects the read and write rates.

Figure 10 shows the results with and without private notifications. When notifications are off, the client polls each topic in a round robin fashion until it notices the new message. Because the read rate is fixed, the end-to-end latency grows linearly with the number of subscribed topics. With private notifications, the receiver periodically receives a global interest vector that encodes the topic with the new message, allowing it to prioritize that read. As a result, the end-to-end latency for a single message is relatively fixed.

9 Discussion

9.1 Cost of Cover Traffic

Because each Talek client must generate network traffic on a regular schedule that is independent from the user's real usage, developers must choose a global schedule that is appropriate for their application. Naturally, there is a trade-off between efficiency and latency. At its most efficient, the schedule matches the average workload, such that occasional users generate more dummy requests to meet the average and prolific users are rate-limited. At the cost of higher overhead, application developers can choose to send requests more frequently to proportionally reduce the average latency of messages. This design decision will largely depend on application workloads.

As a concrete example, consider an application where users read and write 1 message/day along a Poisson distribution. If Talek were configured to match this workload, then the system would incur virtually zero overhead from cover traffic, but it would take on average two days for messages to be delivered. If Talek were configured to read and write four times as often, then 90% of messages would be seen by subscribers within a day of writing. While this means 75% of messages are wasted, this amounts to at most a few megabytes per day for text applications like email and messaging.

9.2 Intersection attacks

Talek's design assumes that users are always online in order to send cover traffic. When Talek is used with asynchronous user interfaces, such as email, productivity apps, and turn-based games, Talek can do better than systems based on k-anonymity. If the application allows the user to have periods of offline usage, then Talek is potentially susceptible to intersection attacks if two users go online and offline at the same time, or if external events (e.g., protests) correlate with the user's online/offline status. In contrast, k-anonymity systems are vulnerable to intersection attacks even without these correlations; every network request leaks information. Because mobile devices frequently go offline to conserve energy, k-anonymity systems are weak under mobile workloads.

System	Security Goal	Threat Model	Technique	Application
Talek	indisting.	≥ 1	IT-PIR	pub/sub
Pynchon [52]	k-anon.	≥ 1	mixnet/IT-PIR	email
Riffle [42]	k-anon.	≥ 1	mixnet/IT-PIR	file-sharing
Riposte [17]	k-anon.	≥ 1	IT-PIR	broadcast
Dissent [18]	k-anon.	≥ 1	DC-nets	broadcast
Vuvuzela [59]	diff. privacy	≥ 1	mixnet	1–1 messaging
DP5 [11]	indisting.	≥ 1	IT-PIR	chat presence
Popcorn [35]	indisting.	≥ 1	C-PIR/IT-PIR	video stream
Pung [5]	indisting.	0	C-PIR	key-value store
ORAM [54–56]	indisting.	0	ORAM	storage

Figure 11: Comparison of privacy-related techniques. Indistinguishability-based security goals offer the strongest level of privacy. Prior work with indistinguishability are either too expensive or are tailored to a single application. k-anonymity systems leak which of k clients a particular message originates from, opening the systems to intersection attacks [20, 40, 44]. Differential privacy only provides a formal framework to quantify information leakage and privacy bounds. The threat model column denotes the number of servers that must be honest for security properties to hold.

10 Related Work

PIR-based systems: Pynchon Gate [52] is an email system where messages are sent to email servers through mixnets. Emails are dumped daily to distributor servers, where clients use IT-PIR to privately retrieve messages. While PIR hides which messages clients are interested in, the email server stores the communication patterns between email addresses. Talek could be used for email exchange with much lower latency and better privacy. Riffle [42] follows a similar design, using mixnets to send messages and IT-PIR to retrieve messages. Riffle provides k-anonymity; in each round of communication the adversary learns that a message could have originated from 1 of k users. Riffle and other systems based on k-anonymity are susceptible to intersection attacks [20, 40, 44], where communication patterns can be deanonymized over time with statistical analysis of k-anonymity sets if users are not always online.

Ostrovsky and Shoup introduced the notion of private information storage (PIS) [47], which allows a client to write to a row to a database of n rows without revealing which row was updated. However, doing so is expensive; PIS incurred poly-logarithmic communication complexity. Riposte [17] expands on this work and applies this technique to support a scalable broadcast messaging system, but it is also vulnerable to intersection attacks. Riposte writes require $O(\sqrt{n})$ messages. In contrast, by making random-looking writes to the server and relying on IT-PIR for reads, Talek writes are cheap, $O(1)$.

DP5 [11] is a chat presence system, which can be used to make users aware of their friends' online presence with strong privacy. The security of the protocol is dependent on chat presence workloads and does not generalize to pub/sub. Popcorn [35] uses both C-PIR and IT-PIR to construct a private read-only video streaming system over a static video database. Each of these systems leverages different sets of assumptions about application workloads to make PIR practical in their setting; in contrast, Talek provides a general-purpose mechanism to support a wide range of applications. Pung [5] is a key-value store with security goals based on indistinguishability. In Section 8.4, we discuss Talek's tradeoffs with respect to Pung, and we show that Pung incurs orders of magnitude higher computational and network costs.

Mixnet-based systems: Chaum mixnets [14, 15, 34, 38] and verifiable cryptographic shuffles [12, 28, 45] are a way to obfuscate the source of a message. Mixnets have been applied to private messaging [58], but require messages from honest users in every round to form an anonymity set. When a mixnet is used to access an encrypted database, unlinkability can be difficult to guarantee when the database is untrusted. Network-level onion routing [23, 39, 49] systems can also be used to access an encrypted database with similar limitations. Using differential privacy analysis, Vuvuzela [59] formalizes the amount of noise that honest shufflers would need to inject in order to bound information leakage at the database. These systems have a weaker security goal than Talek.

DC-nets: DC-net systems [13, 33], like Herbivore [53] and Dissent [18, 19, 60], are a method for anonymously broadcasting messages to a group using information-theoretic techniques. On each message transmission, all clients must broadcast random bits to every other client. DC-nets enable effective broadcast messaging,but are not a good fit for scalable pub/sub workloads because of the high network costs.

Oblivious RAM (ORAM): ORAM [30, 32, 46] is a set of protocols that allow a single trusted client to access untrusted storage without revealing access patterns, even to a strong adversary who controls the storage. High network costs of reads, on the order of $\Omega(\log N)$, and constant data reshuffling make ORAM impractical for systems with many users sharing data.

11 Conclusion

In this paper, we present Talek, a general-purpose private publish-subscribe system. Talek protects both the contents and metadata of users' application usage from untrusted servers. We show that strong security goals based on *access sequence indistinguishability*, where the adversary provably learns no information about which users may be communicating, is practical with two new techniques, *oblivious logging* and *private notifications*. Our evaluations confirm that our implementation achieves 3–4 orders of magnitude better performance than previous systems with similar security goals.

References

[1] Keybase. https://keybase.io/

[2] Pond. https://github.com//agl/pond

[3] Ricochet: Anonymous peer-to-peer instant messaging. https://github.com/ricochet-im/ricochet

[4] AGUILAR-MELCHOR, C., BARRIER, J., FOUSSE, L., AND KILLIJIAN, M.-O. XPIR: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies 2016*, 2 (2015), 155–174.

[5] ANGEL, S., AND SETTY, S. Unobservable Communication over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

[6] AUMASSON, J.-P., AND BERNSTEIN, D. J. Siphash: a fast short-input prf. In *International Conference on Cryptology in India* (2012), Springer, pp. 489–508.

[7] BELLARE, M., DESAI, A., POINTCHEVAL, D., AND ROGAWAY, P. Relations among notions of security for public-key encryption schemes. In *Annual International Cryptology Conference* (1998), Springer, pp. 26–45.

[8] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys 13*, 2 (June 1981), 185–221.

[9] BOLDYREVA, A., GOYAL, V., AND KUMAR, V. Identity-based encryption with efficient revocation. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 417–426.

[10] BONEH, D., AND FRANKLIN, M. Identity-based encryption from the weil pairing. In *Annual International Cryptology Conference* (2001), Springer, pp. 213–229.

[11] BORISOV, N., DANEZIS, G., AND GOLDBERG, I. DP5: A Private Presence Service. *Proceedings on Privacy Enhancing Technologies 2015*, 2 (2015), 4–24.

[12] BRICKELL, J., AND SHMATIKOV, V. Efficient Anonymity-Preserving Data Collection. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2006), ACM, pp. 76–85.

[13] CHAUM, D. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *Journal of Cryptology 1*, 1 (1988), 65–75.

[14] CHAUM, D., JAVANI, F., KATE, A., KRASNOVA, A., DE RUITER, J., AND SHERMAN, A. T. cMix: Anonymization by High-Performance Scalable Mixing.

[15] CHAUM, D. L. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM 24*, 2 (1981), 84–90.

[16] CHOR, B., KUSHILEVITZ, E., GOLDREICH, O., AND SUDAN, M. Private Information Retrieval. *Journal of the ACM (JACM) 45*, 6 (1998), 965–981.

[17] CORRIGAN-GIBBS, H., BONEH, D., AND MAZIÈRES, D. Riposte: An Anonymous Messaging System Handling Millions of Users. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 321–338.

[18] CORRIGAN-GIBBS, H., AND FORD, B. Dissent: Accountable Anonymous Group Messaging. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (2010), ACM, pp. 340–350.

[19] CORRIGAN-GIBBS, H., WOLINSKY, D. I., AND FORD, B. Proactively Accountable Anonymous Messaging in Verdict. In *22nd USENIX Security Symposium* (2013), pp. 147–162.

[20] DANEZIS, G., AND SERJANTOV, A. Statistical Disclosure or Intersection Attacks on Anonymity Systems. In *International Workshop on Information Hiding* (2004), pp. 293–308.

[21] DEVET, C., GOLDBERG, I., AND HENINGER, N. Optimally Robust Private Information Retrieval. In *21st USENIX Security Symposium (USENIX Security 12)* (2012), pp. 269–283.

[22] DIETZFELBINGER, M., AND WEIDLING, C. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. *Theoretical Computer Science 380*, 1 (2007), 47–68.

[23] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The Second-Generation Onion Router. Tech. rep., DTIC Document, 2004.

[24] DIXON, C., ANDERSON, T. E., AND KRISHNAMURTHY, A. Phalanx: Withstanding Multimillion-Node Botnets. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)* (2008), vol. 8, pp. 45–58.

[25] DWORK, C. Differential Privacy. In *Automata, languages and programming*. 2006, pp. 1–12.

[26] DWORK, C. Differential Privacy: A Survey of Results. In *Theory and applications of models of computation*. 2008, pp. 1–19.

[27] EPHREMIDES, A., WIESELTHIER, J. E., AND BAKER, D. J. A Design Concept for Reliable Mobile Radio Networks with Frequency Hopping Signaling. *Proceedings of the IEEE 75*, 1 (1987), 56–73.

[28] FURUKAWA, J., AND SAKO, K. An Efficient Scheme for Proving a Shuffle. In *Advances in Cryptology (CRYPTO 2001)* (2001), pp. 368–387.

[29] GOLDBERG, I. Improving the Robustness of Private Information Retrieval. In *Security and Privacy (SP), 2007 IEEE Symposium on* (2007), IEEE, pp. 131–148.

[30] GOLDREICH, O. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (1987), ACM, pp. 182–194.

[31] GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. On the cryptographic applications of random functions. In *Workshop on the Theory and Application of Cryptographic Techniques* (1984), Springer, pp. 276–288.

[32] GOLDREICH, O., AND OSTROVSKY, R. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.

[33] GOLLE, P., AND JUELS, A. Dining Cryptographers Revisited. In *Advances in Cryptology (Eurocrypt 2004)* (2004), pp. 456–473.

[34] GÜLCÜ, C., AND TSUDIK, G. Mixing E-mail with Babel. In *Network and Distributed System Security, 1996., Proceedings of the Symposium on* (1996), IEEE, pp. 2–16.

[35] GUPTA, T., CROOKS, N., MULHERN, W., SETTY, S. T., ALVISI, L., AND WALFISH, M. Scalable and Private Media Consumption with Popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, Mar. 2016), USENIX Association.

[36] HONG, J. I., AND LANDAY, J. A. An Architecture for Privacy-Sensitive Ubiquitous Computing. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services* (2004), ACM, pp. 177–189.

[37] ISLAM, M. S., KUZU, M., AND KANTARCIOGLU, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS* (2012), vol. 20, p. 12.

[38] JERICHOW, A., MULLER, J., PFITZMANN, A., PFITZMANN, B., AND WAIDNER, M. Real-time Mixes: A Bandwidth-Efficient Anonymity Protocol. *Selected Areas in Communications, IEEE Journal on 16*, 4 (1998), 495–509.

[39] JONES, N., ARYE, M., CESAREO, J., AND FREEDMAN, M. J. Hiding Amongst the Clouds: A Proposal for Cloud-based Onion Routing. In *FOCI* (2011).

[40] KEDOGAN, D., AGRAWAL, D., AND PENZ, S. Limits of Anonymity in Open Environments. In *International Workshop on Information Hiding* (2002), pp. 53–69.

[41] KUSHILEVITZ, E., AND OSTROVSKY, R. Replication is Not Needed: Single Database, Computationally-Private Information Retrieval. In *FOCS* (1997), IEEE, p. 364.

[42] KWON, Y. H., LAZAR, D., DEVADAS, S., AND FORD, B. Riffle: An Efficient Communication System with Strong Anonymity. In *Proceedings on Privacy Enhancing Technologies* (2016).

[43] LAZAR, D., AND ZELDOVICH, N. Alpenhorn: Bootstrapping secure communication without leaking metadata.

[44] MATHEWSON, N., AND DINGLEDINE, R. Practical Traffic Analysis: Extending and Resisting Statistical Disclosure. In *Proceedings on Privacy Enhancing Technologies* (2004), pp. 17–34.

[45] NEFF, C. A. Verifiable Mixing (Shuffling) of ElGamal Pairs. *VHTi Technical Document, VoteHere, Inc* (2003).

[46] OSTROVSKY, R. Efficient Computation on Oblivious RAMs. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing* (1990), ACM, pp. 514–523.

[47] OSTROVSKY, R., AND SHOUP, V. Private information storage (extended abstract). In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1997), STOC '97, ACM, pp. 294–303.

[48] PAGH, R., AND RODLER, F. F. Cuckoo Hashing. In *European Symposium on Algorithms* (2001), pp. 121–133.

[49] REED, M. G., SYVERSON, P. F., AND GOLDSCHLAG, D. M. Anonymous Connections and Onion Routing. *Selected Areas in Communications, IEEE Journal on 16*, 4 (1998), 482–494.

[50] ROY, S. S., VERCAUTEREN, F., MENTENS, N., CHEN, D. D., AND VERBAUWHEDE, I. Compact ring-LWE cryptoprocessor. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2014), Springer, pp. 371–391.

[51] SAN, I., AT, N., YAKUT, I., AND POLAT, H. Efficient Paillier cryptoprocessor for privacy-preserving data mining. *Security and Communication Networks 9*, 11 (2016), 1535–1546.

[52] SASSAMAN, L., COHEN, B., AND MATHEWSON, N. The Pynchon Gate: A Secure Method of Pseudonymous Mail Retrieval. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society* (2005), ACM, pp. 1–9.

[53] SIRER, E. G., GOEL, S., ROBSON, M., AND ENGIN, D. Eluding Carnivores: File Sharing with Strong Anonymity. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop* (2004), ACM, p. 19.

[54] STEFANOV, E., AND SHI, E. Multi-Cloud Oblivious Storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 247–258.

[55] STEFANOV, E., AND SHI, E. ObliviStore: High Performance Oblivious Cloud Storage. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 253–267.

[56] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path ORAM: an Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013), ACM, pp. 299–310.

[57] SWEENEY, L. k-anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10*, 05 (2002), 557–570.

[58] UNGER, N., DECHAND, S., BONNEAU, J., FAHL, S., PERL, H., GOLDBERG, I., AND SMITH, M. SoK: Secure Messaging. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 232–249.

[59] VAN DEN HOOFF, J., LAZAR, D., ZAHARIA, M., AND ZELDOVICH, N. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (2015), ACM, pp. 137–152.

[60] WOLINSKY, D. I., CORRIGAN-GIBBS, H., FORD, B., AND JOHNSON, A. Dissent in Numbers: Making Strong Anonymity Scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), pp. 179–182.

A Appendix

We provide a security proof for Talek's protocol by reduction to the cryptographic assumptions listed in Sections 2.5 and 4.

A.1 Access Sequence Indistinguishability

Definition:

Talek consists of the following, possibly randomized, algorithms:

$Publish(\tau, \delta) \rightarrow \{\omega^0, \ldots, \omega^{l-1}\}$*:* Clients use the Publish function to generate Write requests sent to the l servers. The Publish function takes as input, a topic τ and a message δ to publish to the topic, producing a set of l Write requests, one per server.

$Poll(\tau, i) \rightarrow \{q^0, \ldots, q^{l-1}\}$*:* Clients use the Poll function to generate Read request queries sent to the servers. The Poll function takes as input a topic τ and a sequence number on the topic, producing a set of l Read requests, one per server.

$Write(\sigma, \omega) \rightarrow \sigma'$*:* Servers use the Write function to process incoming write requests. The function takes as input, a server's internal state, σ, and a write request, ω, and ouputs the updated state of the server, σ'.

$Read(\sigma, q) \rightarrow R$*:* Servers use the Read function to process incoming read requests. The function takes as input, a server's internal state, σ, and a read request, q, and outputs a function of internal state, R.

$GetUpdates(\sigma) \rightarrow V$*:* Servers use the GetUpdates function to generate a global interest vector, V. The function takes as input a server's internal state, σ, and outputs a function of internal state, V.

We define access sequence indistinguishability using the following security game, played between the adversary, $\mathscr{A}$, and a challenger, $\mathscr{C}$. $\mathscr{A}$ is a probabilistic, polynomial-time adaptive adversary, who is in control of the network, all but one of the servers, and an unbounded number of clients. $\mathscr{A}$ can drop any message, send arbitrary messages from any of the adversarial clients to any server, respond arbitrarily to requests, and modify any server-side state for adversarial servers. Assume the presence of authenticated secure channels between each client-server pair (e.g. with TLS).

1. $\mathscr{A}$ chooses an non-negative integer, m, and submits this number to the challenger, who spawns m clients, $\mathscr{C}_0 \ldots \mathscr{C}_{m-1}$
2. The challenger flips a coin, $b \in \{0,1\}$, uniformly at random, which is fixed for the duration of the game.
3. For each of the challenger's clients, $\mathscr{C}_j$, $\mathscr{A}$ maintains two unique data access sequences, seq^0_j and seq^1_j.
4. Repeat the following until $\mathscr{A}$ chooses to end the game:
   - $\mathscr{A}$ chooses the i-th operation for both sequences for all challenger clients, $\{seq^0_0[i] \ldots seq^0_{m-1}[i]\}$ and $\{seq^1_0[i], \ldots seq^1_{m-1}[i]\}$. $\mathscr{A}$ submits the operations $seq^0_j[i]$ and $seq^1_j[i]$ to the respective client, $\mathscr{C}_j$. Chosen operations can be a $Publish$, $Poll$, or $NoOp$ to topics for which the adversary does not hold the secret topic handle, τ.
   - Each client, $\mathscr{C}_j$, plays one of the two operations, $seq^b_j[i]$, into the Talek client library.
   - Adversarially controlled clients can send arbitrary requests to any server. Adversarially controlled servers can also modify its own state and respond arbitrarily.
   - $\mathscr{A}$ observes the network events, $events^{b'}_j[i]$ sent from $\mathscr{C}$'s clients to adversarial servers. These events include $Write$ and $Read$ requests.
5. $\mathscr{A}$ outputs its guess for b'.

Definition 2. *(Access Sequence Indistinguishability) We say that the system provides access sequence indistinguishability if for any polynomial-time probabilistic adversary, any challenger clients, and any data access sequences,*

$$|Pr(b = b') - 1/2| \leq negl(\lambda)$$

in the security game, where λ is a security parameter and $negl$ is a negligible function.

Practically, this definition means that an adversary would not be able to distinguish between a real user's access patterns from random access patterns of arbitrary length. It follows from this definition that the adversary should also not be able to determine which users access the same topics, because the adversary could have chosen seq with overlapping topics across users. This security goal also prevents intersection attacks [20,40,44], a common problem in previous private messaging systems, where clients can be linked by observing actions over long periods of time.

Note that $\mathscr{A}$ only specifies the actions of correct users and does not specify access sequences between correct and adversarial clients. As described in Section 5.4, malicious clients with a topic secret could collude with an adversarial server to de-anonymize the publisher to that topic. While this weakens $\mathscr{A}$'s power in the game, it is consistent with our goal of providing privacy guarantees to groups of trusted users. Adversarial clients can still act arbitrarily against any server.

Proof:

We consider a series of games adapted from the game above, each defined from the previous one by idealizing some part of the protocol. For game i, we write p_i for the maximum advantage, $|Pr(b = b') - 1/2|$, that $\mathscr{A}$ holds in the security game. At each step, we bound the adversary's advantage between two successive games. Technically, each of the following games consists of a series of hybrid games, where we change each of the m clients one by one.

Game 0: Consider the game defined above with an adversary $\mathscr{A}$ that chooses m challenger clients, and submits sequences with α_0 calls to $Poll$ and α_1 calls to $Write$.

Game 1: (PIR $Read$) This game is as above, except we replace the PIR request vectors, q_j, generated in $Poll$, with a bitstring $q_j \leftarrow \{0,1\}^b$ sampled at random. Let $\epsilon^{PIR}(\lambda_0, n)$ bound the advantage of an adversary breaking the PIR assumption in n calls to $Read$ with security parameter λ_0. The adversary distinguishes between the request vectors in Game 0 and the randomly sampled requests in Game 1 with advantage $\epsilon^{PIR}(\lambda_0, \alpha_0)$.

$$p_0 \leq p_1 + m \cdot \epsilon^{PIR}(\lambda_0, \alpha_0)$$

Game 2: (IND-CCA with $Write$) This game is as above, except that for each client, we maintain a table T that maps ciphertexts under key k to plaintext messages δ. $Publish$ is modified to encrypt a dummy message instead of δ and to record in T the resulting ciphertext and δ. Attempting to decrypt any ciphertext not in the table is rejected. $Poll$ is modified to retrieve plaintext from T. We can apply our IND-CCA assumption for AEAD to each key k. Let $\epsilon^{AEAD}_{IND-CCA}(\lambda_1, n)$ be the advantage of an IND-CCA adversary that performs n oracle encryptions and decryptions with security parameter λ_1.

$$p_1 \leq p_2 + m \cdot \epsilon^{AEAD}_{IND-CCA}(\lambda_1, \alpha_1)$$

Game 3: (PRF with $Write$) This game is as above, except we replace the PRF used to generate the bucket locations of $Write$s with a perfect random function. Let $\epsilon^{PRF}_{distinguish}(\lambda_2, n)$ bound the advantage of an adversary breaking our PRF assumption in n calls to PRF with a security parameter, λ_2

$$p_2 \leq p_3 + 2 \cdot m \cdot \epsilon^{PRF}_{distinguish}(\lambda_2, \alpha_1)$$

Game 4: (Hash functions in interest vectors) This is game is as above, except we replace the h cryptographic hash functions used in the Bloom filter of the message interest vector with queries to a random oracle. Thus, the message interest vector will contain all 0's, except for 1's in h random positions. Let $\epsilon^{hash}(\lambda_3, n)$ bound

```
{
  topicId: uint128
  seed1: uint128,
  seed2: uint128,
  encKey: byte[]
}
```

(a) Topic Handle

```
Encrypt ({
  topicId: uint128,
  seqNo: uint64,
  value: byte[],
  signature: byte[],
}, encKey)
```

(b) Message Payload

Figure 12: Schema of the topic handle and a message payload. The topic handle is a shared secret between a trusted group of publishers and subscribers, used to reconstitute a topic log from the servers. Each message payload in the topic log is encrypted with a shared encryption key.

the advantage of an adversary breaking our assumption of cryptographic hash functions in n calls with a security parameter, λ_3.

$$p_3 \leq p_4 + m \cdot \epsilon^{hash}(\lambda_3, \alpha_1)$$

From this final game, all of the parameters in any network request have been replaced with random values. Because Game 4 involves all clients issuing periodic requests with random parameters, by definition the adversary's advantage, p_4, must be zero.

Privacy: Collecting the probabilities from all games yields:

$$\begin{aligned} p_0 \leq & m \cdot \epsilon^{PIR}(\lambda_0, \alpha_0) + \\ & m \cdot \epsilon^{AEAD}_{IND-CCA}(\lambda_1, \alpha_1) + \\ & 2 \cdot m \cdot \epsilon^{PRF}_{distinguish}(\lambda_2, \alpha_1) + \\ & m \cdot \epsilon^{hash}(\lambda_3, \alpha_1) \end{aligned}$$

p_0 becomes negligible for large security parameters λ_0, λ_1, λ_2, and λ_3.

A.2 Pseudocode

See following page.

```
//GlobalState
globalLog ← Array()                         ▷ Global log of write operations
seqNo ← 0                                   ▷ Global sequence number
hashtable ← BlockedCuckoo(b, d)             ▷ b buckets of depth d

//Writes the data into one of two buckets
function WRITE(buckets,data,interestVec)
    if isLeader() then
        seqNo ← seqNo + 1
        Append operation to globalLog
        Forward operation to follower servers with seqNo
    else
        Insert operation into globalLog at given seqNo
    end if
    Remove n-th oldest element from hashtable
    hashtable.insert(buckets[0], buckets[1], data)
end function
```

```
//Performs a PIR-based read
function READ(bucketVector)
    return bucketVector · hashtable
end function
```

```
//Returns the global interest vector
function GETUPDATES
    v ← BloomFilter()
    for all e ∈ last n elements of globalLog do
        v ← v ∪ e.interestVec
    end for
    return v
end function
```

Figure 13: Pseudocode for server-side RPC handlers (NPI). The NPI was designed such that all parameters for any operation reveal no information about the user's application usage. Writes are serialized by the leader and replicated in global order to the follower servers. When writing, clients explicitly specify the two potential hash table buckets into which data is inserted. When data is read using a PIR protocol, we expose a blocked cuckoo hash table with the n most recent messages in the log and return full buckets. For simplicity, we only describe the original IT-PIR algorithm, which we show is equivalent to the serialized PIR algorithm in Section 4.4

```
//GlobalState
topics ← Map()              ▷ Latest sequence numbers seen for each topic
readQueue ← Queue()
writeQueue ← Queue()

function PUBLISH(topic, message)
    Enqueue operation to writeQueue
end function

function SUBSCRIBE(topic)
    Add topic to topics
end function

function PERIODICWRITE
    if writeQueue.isEmpty() then
        Send a random write to the leader
    else
        topic, data ← writeQueue.dequeue()
        seqNo ← topics[t.id]++
        bucket1 ← PRF(t.seed1, seqNo)
        bucket2 ← PRF(t.seed2, seqNo)
        data' ← Enc_{topic.key}(data)
        intVec ← BloomFilter()
        intVec.insert(topic.id|seqNo)
        leader.Write([bucket1, bucket2], data', intVec)
    end if
end function
```

```
function PERIODICREAD
    if readQueue.isEmpty() then
        Send a random read to each server
    else
        topic, seqNo, seedChoice ← readQueue.dequeue()
        seed ← (seedChoice == 1)?topic.seed1 : topic.seed2
        data, query_l, query' ← [0 . . . 0]
        query'[PRF(seed, seqNo)] ← 1                    ▷ secret
        for each server in followers do
            query ← RandomBitString(numBuckets)
            data ← data ⊕ server.Read(query)
            query_l ← query_l ⊕ query
        end for
        query_l ← query_l ⊕ query'
        data ← data ⊕ leader.Read(query_l)
        if data contains (topic, seqNo) then
            return data
        else if seedChoice == 1 then
            Enqueue a read for (topic, seqNo, 2) to readQueue
        end if
    end if
end function
```

```
function PERIODICUPDATES
    globalIntVec ← leader.GetUpdates()
    for all topic ∈ topics do
        seqNo ← topics[topic.id]
        if globalIntVec.contains(topic.id, seqNo) then
            Enqueue a read for (topic, seqNo, 1) to readQueue
        end if
    end for
end function
```

Figure 14: Pseudocode for the Talek client library. Calls to publish and subscribe are queued in a global request queue. A periodic process either issues a random request or dequeues a legitimate operation to be translated into a privacy-preserving NPI request. Messages in a topic are are written to a deterministic random order of buckets. Subscribers then use PIR to retrieve these messages. For simplicity, we describe the protocol assuming an authenticated secure channel to each server. In practice, we use a functionally equivalent serialized version of PIR described in Section 4.4.