Talek: Private Group Messaging with Hidden Access Patterns

Raymond Cheng
University of Washington

William Scott
University of Washington

Elisaweta Masserova
Carnegie Mellon University

Irene Zhang
Microsoft Research

Vipul Goyal
Carnegie Mellon University

Thomas Anderson
University of Washington

Arvind Krishnamurthy
University of Washington

Bryan Parno
Carnegie Mellon University

ABSTRACT

Talek is a private group messaging system that sends messages through potentially untrustworthy servers, while hiding both data content and the communication patterns among its users. Talek explores a new point in the design space of private messaging; it guarantees access sequence indistinguishability, which is among the strongest guarantees in the space, while assuming an anytrust threat model, which is only slightly weaker than the strongest threat model currently found in related work. Our results suggest that this is a pragmatic point in the design space, since it supports strong privacy *and* good performance: we demonstrate a 3-server Talek cluster that achieves throughput of 9,433 messages/second for 32,000 active users with 1.7-second end-to-end latency. To achieve its security goals without coordination between clients, Talek relies on information-theoretic private information retrieval. To achieve good performance and minimize server-side storage, Talek introduces new techniques and optimizations that may be of independent interest, e.g., a novel use of blocked cuckoo hashing and support for private notifications. The latter provide a private, efficient mechanism for users to learn, without polling, which logs have new messages.

KEYWORDS

privacy, anonymity, messaging

1 INTRODUCTION

Messaging applications depend on cloud servers to exchange data, giving server operators full visibility into the communication patterns between users. Even if the communication contents are encrypted, network metadata can be used to infer which users share messages, when traffic is sent, where data is sent, and how much is transferred. This can allow the servers and/or network providers to infer the contents of the communication [53]. 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, e.g., journalists and activists, protecting metadata is critical to their job function and safety [68, 69].

As we describe in §10, a wide variety of systems explore ways of protecting the privacy of such metadata. We can classify this prior work into two groups based on the privacy *guarantees* offered and the *threat model* each system defends against. The first group of work [8–10, 19, 46, 47, 50, 72] offers strong security guarantees against very strong threat models (e.g., assuming that only the clients themselves are trusted). Unfortunately, this typically imposes prohibitive computational or network costs. The second group [2, 4, 28, 29, 31, 58–60, 63, 76, 79, 87, 89, 90] offers weaker security guarantees (such as k-anonymity [85], plausible deniability [52] or differential privacy [39, 40]) and often much weaker threat models too; e.g., a fraction of the servers must be honest. However, in exchange for weakening the guarantees and threat model, these systems often achieve impressive performance results.

In this work, we explore an intriguing middle ground: we define *access sequence indistinguishability*, a notion similar to (but slightly stronger than) the security guarantees from systems in the first group, and combine it with an "anytrust" threat model [91], which is slightly weaker than the threat models of the first group, but stronger than those of the second group. Intuitively, the anytrust threat model assumes that different organisations, e.g., Mozilla, EFF, and WikiLeaks, each provide servers and at least one (unknown) organisation can be trusted. The hope is that we can promise the strong guarantees of the first group and the strong performance of the second, while defending against a strong threat model.

To explore this design point, we construct Talek, a private communication system targeting small groups of trusted users communicating amongst themselves (e.g., friends chatting via IRC or text messaging). To support such applications, Talek offers the abstraction of a private log with a single writer and multiple readers. Clients store and retrieve asynchronous messages on untrusted servers without revealing any communication metadata. If the group of friends and at least one server are uncompromised, Talek prevents an adversary from learning anything about their communication patterns. Combined with standard message encryption, Talek conceals both the *contents* and *metadata* of clients' application usage without sacrificing cloud reliability and availability.

Similar to prior systems, to hide communication patterns, Talek clients issue fix-sized, random-looking network requests at a rate which is independent of application-level requests. Hence, application-level requests must occasionally be delayed, and "dummy" network requests must be issued when no application requests are ready. As with any privacy system, careful application-specific tuning is necessary to trade off between the amount of cover traffic sent and the latency of real application requests.

Unlike prior systems with Talek's strong guarantees, to achieve good performance, Talek leverages the anytrust threat model, which allows us to use *information theoretic* private information retrieval (IT-PIR) [27, 35, 45]. IT-PIR requires an anytrust assumption, but in exchange it avoids the use of heavyweight crypto operations required for other flavors of PIR [57]. Abstractly, PIR allows a client

to retrieve the i-th record from a "database" of n items held collectively by l servers, without the servers learning which record was retrieved. However, PIR alone (of any flavor) is not enough to support efficient group messaging. In particular, it does not explain how messages are privately written to the servers, how readers find messages sent to them, nor how to structure the PIR database to facilitate maximum efficiency.

In Talek, within a group of clients reading and writing to a message log, a shared secret determines a pseudorandom, deterministic sequence of database locations for messages. Any user with the log secret (a capability) can follow the pseudorandom sequence, reading new messages independently. These secrets may be shared through in-band control logs. One of Talek's key innovations is to show how these messages can be stored in a *blocked cuckoo hash table* [36] to provide efficient time and space usage in the context of IT-PIR. We further optimize read performance via a novel technique - *private notifications*. Private notifications allow users who subscribe to multiple logs to learn, in a privacy-preserving fashion, which of their logs have new messages. Hence, they can internally prioritize reading from those logs and avoid inefficient polling of logs that have no new messages.

Like any privacy system, Talek has several limitations. First, the current implementation of Talek does not guarantee liveness; any user in a trusted group can block writes to that group's log. Similarly, a faulty server can impact availability. Clients can detect but not attribute such faults. Hence, service providers should be chosen with reputations for high availability. An alternative is to use robust versions of PIR [35, 45] at the cost of higher overhead. We defer investigating this option to future work. Second, because clients connect directly to Talek servers, as with other practical systems [9, 10, 58–61, 87, 89], we do not hide when users are online. Orthogonal systems (e.g., Tor [37]) may help here. Third, as discussed earlier, we expect Talek to be used for communication among small groups of trusted users. If a log secret is shared with the adversary, writer anonymity for that log is compromised, but readers' anonymity and writer anonymity for other logs are preserved. Applications that require broadcasts to many untrusted users (e.g., a public blog) are better served by anonymous broadcast systems [28, 29, 31, 90].

We have implemented two versions of Talek, one entirely in Go and one that offloads PIR operations to a GPU; our code is publicly available. We evaluated the system on a 3-server deployment using Amazon EC2. To provide a realistic group messaging workload, we replay the Ubuntu IRC message logs from 2016 [3]. Overall, we find that this design point is surprisingly practical. Even with 32,000 clients actively reading and writing messages according to a fixed schedule every second, we show that clients use 148 MB per day to achieve an average end-to-end message latency of 1.7 seconds, measured from the time a sender enters a message to the time the recipient sees it. Under this workload, our server supports a peak throughput of 9,433 messages per second, orders of magnitude better performance than systems with similar security goals.

In summary, we make the following contributions.

- Talek, a system that explores an important design point within private group messaging with strong guarantees, strong threat model, and high performance.
- A novel use of blocked cuckoo hashing for IT-PIR.
- Private notifications which privately encode the set of new messages, helping clients prioritize reads.
- Two open-source implementations of Talek exploring the tradeoffs between CPU and GPU-based computation.

2 BACKGROUND: PIR

Talek uses the privacy guarantees of PIR in the context of a group messaging 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) [57] and information-theoretic PIR (IT-PIR) [27, 35, 45]. Talek is compatible with both varieties. Prior work [9, 10] explored C-PIR, since it supports the strongest possible threat model: only the communication partners must be trusted. However, their results indicate that C-PIR imposes significant computation and network overheads, raising practicality concerns. With Talek, we focus on IT-PIR, which requires an anytrust threat model but holds the promise of better performance.

To provide intuition for the performance and cost of IT-PIR, we illustrate a standard protocol [27] with an example. Let l represent the number of servers, each storing a full copy of the database, partitioned into equal sized blocks. While IT-PIR generalizes to arbitrary numbers of servers and blocks, our example contains $l = 3$ servers and $n = 3$ blocks ($\{B_1, B_2, B_3\}$).

(1) Suppose a client wants to read the second block, $\beta = 2$, encoded by the bit vector, $q' = [0, 1, 0]$, which consists of zeros and a one in position β.
(2) The client generates $l - 1$ (e.g., 2) random n-bit request vectors, q_1 and q_2.
(3) The client computes the last request vector as 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 $1 \leq i \leq l$. Since request vectors are generated randomly, this reveals no information to any collection of $< l$ colluding servers.
(5) Suppose B_j represents the j^{th} block of the database. Each server i receives $q_i = [b_1, \ldots, b_n]$ and 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, i.e., $B_\beta = R_1 \oplus \ldots \oplus R_l$ (since $q' = q_1 \oplus \ldots \oplus q_l$, and q' is all zeroes except at index β).

IT-PIR has desirable network properties: a client sends one request vector 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. The size of a client request scales with total number of blocks, and the client work scales with the number of servers.

For the servers, IT-PIR can be computationally expensive even though the individual operations (XOR) are cheap, since the computational cost for a read request scales at least linearly with the size and number of blocks in the system. Theoretical work suggests this limitation is inherent (§10). IT-PIR also requires consistent snapshots across servers, with equal sized blocks in the data structure.

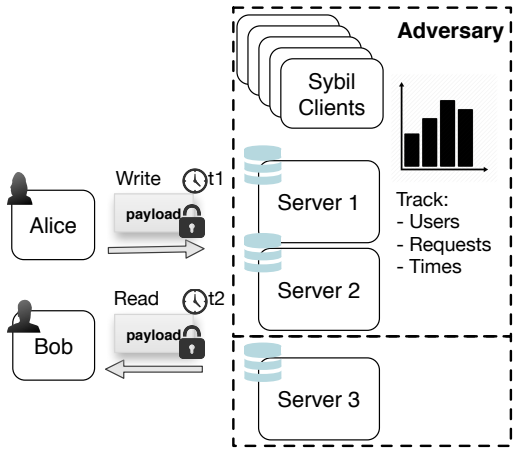

Figure 1: **Talek's System and Threat Model.** We assume the adversary can control all but one of l servers in the system (here, $l = 3$). Clients send network requests directly to the servers. Adversarial servers are free to record additional data, such as the source, type, parameters, timing, and size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

3 PROBLEM DEFINITION & THREAT MODEL

Figure 1 illustrates our problem setting: various clients located across a wide-area network communicate via a messaging service. The service is provided by l servers each controlled by a different administrative domain.

We wish to prevent the adversary from learning any information about the communication patterns amongst honest users. While not described in this paper, other end-to-end messaging guarantees [88], such as message integrity, authentication [13, 78], forward secrecy [6], and fork consistency [64, 66], can be provided by Talek by including additional data in the message payload. We focus on the privacy of access sequences, since that forms the foundation for private messaging.

We define a model of client and server interactions (§3.1), and then introduce *access sequence indistinguishability* (§3.2) to capture the adversary's inability to learn information about the communication between honest clients.

Access sequence indistinguishability provides one of the strongest definitions of privacy available in private group messaging. It is reminiscent of the definitions used in early works on oblivious RAM (ORAM) [46, 47, 72], but it supports multiple distinct readers and writers. It is stronger than k-anonymity [85], where the adversary can narrow the user's identity down to one of k users (where for performance, k is typically much smaller than the number of online users). It is also stronger than plausible deniability [52], where information leakage is allowed up to a certain confidence bound. It is different from unobservability [51], in that we do not hide the maximum number of messages sent by the user, but we do account for the order of the messages in the access sequences. It is most similar to the notion of relationship unobservability under explicit retrieval (UO-ER) [9, 10], which is based on the notion of relationship unobservability [75]. We believe access sequence indistinguishability offers a cleaner definition, however, and differs in at least two small, concrete ways:

(1) In UO-ER, at least one message is sent *and* received in each round. In access sequence indistinguishability, the send and receive rates are decoupled.
(2) In access sequence indistinguishability, the adversary is fully adaptive, in the sense that the adversary can observe the network events and modify its strategy in each round. In UO-ER, each time the adversary adapts its strategy, the state of the clients resets, which may reduce the adversary's utility from acting adaptively.

3.1 Model

Our setting consists of a finite set of servers and clients, and we assume that all parties are stateful. The overall system consists of the following, possibly randomized and interactive, subroutines:

$RealWrite(\tau, seqNo, M) \rightarrow \{\omega^0, \ldots, \omega^{l-1}\}$: Clients use the RealWrite function to generate Write requests sent to the l servers. The RealWrite function takes as input a log handle τ and a message M with the sequence number $seqNo$ to publish to the log, producing a set of l Write requests, one per server.

$FakeWrite() \rightarrow \{\omega^0, \ldots, \omega^{l-1}\}$: Clients use the FakeWrite function to generate a set of random "dummy" Write requests, one per server.

$RealRead(\tau, seqNo) \rightarrow \{q^0, \ldots, q^{l-1}\}$: Clients use the RealRead function to generate Read request queries sent to the servers. It takes as input a log handle τ and a sequence number $seqNo$ in the log, producing a set of l Read requests, one per server.

$FakeRead() \rightarrow \{q^0, \ldots, q^{l-1}\}$: Clients use FakeRead to generate a set of random Read requests, one per server.

$GetUpdates() \rightarrow V$: Clients use the GetUpdates function (modeled as an interactive subroutine) to retrieve the global interest vector, V, from the servers. The global interest vector encodes the set of (log handle, message sequence number)-pairs for all messages currently on the server.

$ProcessWrite(\omega^i) \rightarrow \perp$: Server i uses the ProcessWrite function to process an incoming write request, ω^i, and update the server's internal state.

$ProcessRead(q^i) \rightarrow R^i$: Server i uses the ProcessRead function to process incoming read requests. The function takes as input a read request, q^i, and outputs a reply R^i.

$ProcessGetUpdates() \rightarrow V^i$: Server i uses ProcessGetUpdates to generate a global interest vector, V^i.

3.2 Security Game Definition

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 adversary who is in control of the network, t out of l servers (for Talek, $t = l - 1$), and a polynomial 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. The challenger, in turn, emulates (internally) the honest clients and servers.

Intuitively, the game allows the adversary to repeatedly, dynamically, and adaptively specify any two actions for each legitimate client to take. The challenger has the client execute one of the two

actions, based on the random bit b chosen for the game, and the adversary can then choose new actions for the legitimate clients and/or have its malicious clients and servers take actions. This continues until the adversary produces a guess for the value of b. If the adversary does no better than random chance, this implies that the adversary cannot determine which users access the same logs, because the adversary could have chosen to have client actions with overlapping logs across users.

In the definition, for simplicity, we assume the presence of authenticated secure channels between each client-server pair (e.g., with TLS) and omit the corresponding operations.

(1) $\mathscr{A}$ chooses a 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_j^0 and seq_j^1.
(4) The challenger maintains a table T of log handles that have been created. T is initially empty.
(5) In each round, until $\mathscr{A}$ ends the game, one of the following four actions happens:
   (a) $\mathscr{A}$ **creates a log**
   - $\mathscr{A}$ submits a create log request to the challenger. This request specifies the index of a client acting as a writer of the log, and the indices of the clients acting as readers of the log.
   - $\mathscr{C}$ checks the client indices provided in the create log request, and if all indices correspond to the clients spawned by the challenger, $\mathscr{C}$ generates a log handle τ. $\mathscr{C}$ saves τ in T and returns the corresponding index *ind* in T to the adversary.

   (b) $\mathscr{A}$ **causes clients to** *GetUpdates*
   - $\mathscr{A}$ identifies a set of clients I.
   - Clients in I execute *GetUpdates*. Challenger-controlled servers respond with *ProcessGetUpdates*.

   (c) $\mathscr{A}$ **extends the access sequences**
   - For all m challenger clients, $\mathscr{A}$ chooses the i-th operation for both sequences: $\{seq_0^0[i] \ldots seq_{m-1}^0[i]\}$ and $\{seq_0^1[i], \ldots seq_{m-1}^1[i]\}$. $\mathscr{A}$ submits operations $seq_j^0[i]$ and $seq_j^1[i]$ to the respective client, $\mathscr{C}_j$. An operation can be $(ind, RealWrite(\cdot, seqNo, M))$, $(ind, RealRead(\cdot, seqNo))$, $FakeWrite()$, or $FakeRead()$, where *ind* is the index of the log handle for the request.
   - Each client, $\mathscr{C}_j$, receives one operation: $seq_j^b[i]$. If it is a $FakeRead()$ or $FakeWrite()$ operation, $\mathscr{C}_j$ adds it to its read- or write queue, respectively. Otherwise, the challenger looks up the index *ind* in table T. If it finds the index and corresponding log handle τ, client $\mathscr{C}_j$ sets the log handle of $seq_j^b[i]$ to τ and adds this updated operation to the read queue if $seq_j^b[i]$ is a *RealRead*, and to the write queue if it is a *RealWrite*.
   - If a write request must be issued this round (according to the write rate w), each challenger client $\mathscr{C}_j$ dequeues a request from its write queue and executes it. If the write queue is empty, $\mathscr{C}_j$ executes a *FakeWrite*.
   - If a read request must be issued this round (according to the read rate r), each challenger client $\mathscr{C}_j$ dequeues a request from its read queue and executes it. If the read queue is empty, $\mathscr{C}_j$ executes a *FakeRead*.
   - Challenger-controlled servers use the *ProcessRead* and *ProcessWrite* routines to respond to the corresponding requests.
   - Adversary-controlled clients can send arbitrary requests to any server. Adversary-controlled servers can modify their own state and respond arbitrarily.
   - $\mathscr{A}$ observes the network events, $events_j^{b'}[i]$ sent from $\mathscr{C}$'s clients to all servers, including the outputs of *RealWrite*, *FakeWrite*, *RealRead*, and *FakeRead*.

   (d) $\mathscr{A}$ **ends the game with its guess, b', for b.**

Definition 3.1. (Access Sequence Indistinguishability) The system provides access sequence indistinguishability with security parameter λ if for any polynomial-time probabilistic adversary

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

in the security game, where *negl* is a negligible function.

In the game, when the adversary asks to create a log, we restrict the members of the group to honest clients, which corresponds directly to the trusted group assumption. We do not hide retrievals of the global interest vector via *GetUpdates*, but we do hide the fact that a user of a trusted group is executing a real read or write request. In Talek, $t = l - 1$, since this corresponds to the anytrust threat model.

Availability. Threats to availability are out of scope for this paper. During normal operation, all servers must be available and reachable by the clients. Each server can deny service availability by refusing to respond or by responding with faulty information. However, the application developer and clients can detect (though not attribute) such faulty behavior. We assume developers will choose services known for availability. While we do not discuss it in this paper, Byzantine fault tolerant variations of private information retrieval [35, 45] can be used for better availability guarantees at the cost of higher overhead. Adversarial clients can also degrade service through denial-of-service attacks.

Intersection attacks. An ideal private messaging system would make an honest user's actions perfectly indistinguishable from all other honest users actions. Even such a perfect system would still have its limitations; for example, if only two honest users ever participate, then an attacker can infer that those two users are likely to be communicating with each other.

In practice, most private messaging systems do not achieve ideal hiding; instead they hide a user's actions with a smaller anonymity set, e.g., the set of all online users, or online writers, or even a smaller subset thereof. These compromises are made due to practical constraints or the desire for better performance. For example, if users are allowed to go offline without causing the rest of the system to stop functioning, then the anonymity set of any active user is necessarily reduced to only those users active at the same time. Since these anonymity sets change over time (e.g., based on when users come online), users may be vulnerable to intersection attacks [33, 56, 67] in which the adversary observes a particular user's anonymity sets over time, and, by intersecting those sets, narrows down the user's overall anonymity set.

Talek provably provides access sequence indistinguishability, but, as with most private messaging systems [9, 10, 58–61, 87, 89],

for practical reasons Talek allows users to go offline. As a result, an honest user is hidden amongst all honest online users. This offers stronger privacy than systems based on k-anonymity. In such systems, for performance reasons, k is typically set to be much smaller than the set of all online users (e.g., k might be the number of users actively writing in a round). Furthermore, in each round of communication, the chosen set of k users may vary, which means the adversary has many opportunities to perform intersection attacks on these small sets.

3.3 Assumptions

To provide access sequence indistinguishability, Talek relies on several assumptions. We assume that server storage capacity is scaled to the number of clients. We assume that communicating clients already know each others' long-term public keys. Talek is compatible with bootstrapping keys from other applications [5, 34] or identity-based encryption [16, 17, 62]. We assume each server has a public-private key pair, pk, sk, generated using an algorithm $PKGen()$, and that all server public keys are known to all users. Establishing such keys is orthogonal to the properties Talek provides.

As with prior work [9, 10], Talek supports groups of mutually trusting users. We assume users communicating together trust each other to protect shared secrets. If this trust is misplaced, the writer's anonymity may be lost, but PIR still preserves reader anonymity.

4 DESIGN OVERVIEW

In this section, we give a brief overview of Talek's key design decisions. Later, we present Talek's core operations more formally and in greater detail (§5), followed by two optimizations to improve read performance: serialized PIR (§6) and private notifications (§7).

Talek's core abstraction is a private log, which enables a single writer to share messages with many readers. Groups can then be formed with separate logs for every writer, such that all members subscribe to the logs of all other group members.

In general, we do not expect to see a single universal Talek service; instead, we expect developers (or federations thereof) will stand up separate Talek instances (e.g., one for Instagram and one for Twitter). In practice, a single application might even run two parallel instances of Talek, e.g., one for text-based data, and one with higher latency for images. Running separate instances will allow developers to better tune Talek's parameters to their application.

4.1 Talek's Client Interface

Talek achieves our security goal by *requiring all users to behave identically from the perspective of any colluding set of $l-1$ servers.* A key step in this direction is decoupling the *real* rate at which a user generates application read/write requests from the rate at which network read/write operations are performed.

As Figure 2 illustrates, developers link their messaging application to the Talek client library, which maintains internal read- and write queues. Whenever a request is issued by the messaging application, Talek places its payload on the corresponding internal queue. The Talek client library issues equal-sized `Read` and `Write` network requests at a rate that is independent of the user's real request rate, issuing a dummy request (i.e., a *FakeWrite* or a *FakeRead*) if the respective queue is empty. To preserve privacy, a *FakeWrite*

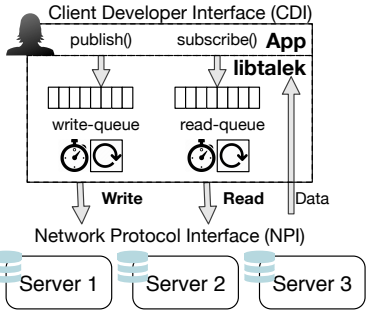

Figure 2: **Talek Client Interface.** Application calls are translated by the client library into scheduled messages with equal-sized parameters and contents that appear random to an adversary. Clients behave identically from the perspective of any $l-1$ servers.

(*FakeRead*) request, including its parameters and payload, must be indistinguishable from a *RealWrite* (*RealRead*) request. Talek achieves this by defining a globally-fixed message size, z, to which messages are split and padded to fit. Messages are then encrypted with a symmetric authenticated encryption scheme [14] to provide confidentiality and authenticity.

Choosing appropriate network rates. In practice, the developer should measure real global usage and sample randomly from this aggregate distribution. They could also configure a fixed burst of messages every time a user comes online. Privacy is preserved as long as the distribution of requests is independent of user activity.

4.2 Oblivious Logging Overview

Decoupling the real rate at which a user generates read/write requests from the network request rate is insufficient for privacy, as it fails to hide which users read each other's messages. Hence, Talek uses PIR (§2) to allow users to retrieve records privately. However, PIR alone does not explain how to write messages to the servers or how to let communication partners know which messages to read.

To allow clients to read *and* write messages privately without explicit coordination, Talek issues writes to pseudo-random locations on each server, similar to prior work [9, 10, 61, 87, 89]. We call our version of this technique *oblivious logging*. The sequence of write locations is determined by applying a pseudorandom function, *PRF*, to a secret log handle shared (using control logs §5.5) between the log's writer and readers. Readers use PIR to retrieve messages by following the PRF-derived sequence of locations.

Hence, server simply see a series of seemingly random writes and PIR-protected reads, hiding the communication patterns between writers and readers. Exposure of the log handle (e.g., by an untrustworthy group member) will expose the writer's access pattern, but not reader consumption patterns, since the latter are protected information theoretically by PIR's guarantees.

4.3 Talek's Server Design

Servers store a limited set of messages to allow asynchronous senders and receivers to be decoupled in time. Since the cost of PIR operations scales linearly with database size, for good performance we fix the number of messages stored on each server to n, garbage collecting the oldest. n is directly related to the time-to-live, TTL, for messages, which dictates how tightly synchronized senders and

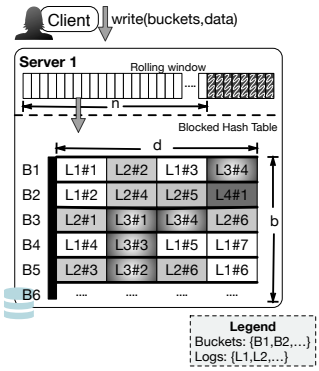

Figure 3: **Server Data Structures and Workflow.** Garbage collection discards all but the latest n messages. Client writes specify two random buckets in which each message can be placed, forming a blocked cuckoo hash table. Logs are spread over the hash table, and read using PIR. Messages in the same log are colored with the same shade in the diagram.

receivers must be. As the number of clients in the system grows, the system requires larger values of n to support the same TTL.

Like prior messaging systems [9, 10], we face the challenge of finding a way to efficiently pack messages into a dense data structure that is compatible with PIR. One of Talek's key innovations is the use of a blocked cuckoo hash table [36, 74], characterized by a fixed value of b buckets each containing d messages. Each client `Write` request explicitly specifies (to the servers) two pseudo-randomly chosen buckets in which the message can be inserted. The servers apply the standard cuckoo-hash insertion algorithm, which potentially results in cuckoo evictions if both buckets are full, but guarantees that every item ends up in one of its two hash-selected locations. When the client performs 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.

Blocked cuckoo hashing has a number of desirable properties. Compared to chained hash tables, all buckets have an equal fixed size, a necessary requirement for PIR. Further, only a single copy of each message written is stored, minimizing storage overhead. To handle collisions, the size of the table does need to be larger than n, but with blocked cuckoo hashing the overhead is generally less than 20% for reasonable values of the bucket size d. This also helps minimize the cost of PIR operations, which grow with the size of the database. Finally, there are only two possible locations for each message, and thus a client issues at most two `Read` requests to check both buckets where a message could be stored; this is important as PIR operations are expensive. If the client finds the message it is looking for in the first bucket, then it can use its next `Read` request for another *RealRead*, rather than querying the second cuckoo hash location. From the server's perspective, the client is simply issuing a stream of opaque PIR requests.

5 TALEK'S CORE OPERATIONS

This section expands on the overview from §4 to provide details on the Talek protocol for creating single-writer, multi-reader logs. Log handles (§5.2) allow readers to find the latest content from writers without explicit coordination. They dictate how messages are written into the servers' cuckoo hash tables (§5.3), and how readers retrieve messages using PIR (§5.4). Because reads are done with PIR, many readers can poll the same log repeatedly without revealing information. Control logs (§5.5) simplify group messaging sessions. Finally, we offer some thoughts on the tradeoffs involved in choosing Talek's parameters (§5.6) and provide some intuition (§5.7) for Talek's proof of security (Appendix A.2).

Figure 4 summarizes the client/server interfaces and state; detailed pseudocode for the client and servers are in Appendix B.

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

Figure 4: **Talek Interfaces and Client/Server State.** Global interest vectors are described in §7.

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: **Talek Variables.** Globally configured parameters are fixed across all clients and servers for an instance of Talek.

5.1 Notation

For convenience, Figure 5 summarizes constants that parameterize Talek's design. In §5.6, we discuss configuring these parameters.

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. We write $Enc_k(text)$ for the encryption of $text$ with key k, and $Dec_k(cipher)$ for the decryption of $cipher$. Let $PRF(key, input)$ denote a pseudo-random function family and $PRNG(seed)$ denote a cryptographically secure pseudorandom number generator. We use | to denote tagged concatenation.

5.2 Log Handles

When a user creates a new log, Talek generates a secret *log handle*, τ, containing a unique ID, id, encryption key, k_{enc}, and two seeds, k_{s1} and k_{s2}. Using control logs (§5.5), the log handle is shared between the writer and readers of a log. All messages in the log are encrypted with k_{enc}, e.g., $Enc_{k_{enc}}(message)$. We further assign all messages in a log a sequence number, $seqNo$. This sequence number is internal to Talek, and does not preclude application-layer logic to track and order messages. The two seed values are used in conjunction with a pseudorandom function family, $PRF(seed, seqNo) \in \{0 \ldots (b-1)\}$,

to produce two unique and deterministic sequences of bucket locations for writes. Similar to frequency hopping [38, 41] and constructions in prior work [9, 10, 61, 87, 89], log handles allow writers and readers to agree on a pseudorandom sequence of buckets without online coordination. As discussed in §3, additional properties, like forward secrecy, can be layered atop this basic construction.

5.3 Writing into a Cuckoo Hash Table

A key part of Talek's design is that server-side state is arranged in a blocked cuckoo hash table [36, 74], where each server's storage is organized into b buckets, each bucket storing d messages, each of size z. The value of b is chosen in conjunction with the PRF family, such that the output of a PRF is uniformly distributed across the buckets. The servers' message capacity, 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 [36]; hence insertion costs $O(1)$.

Because cuckoo hashing is history-dependent and PIR requires consistent replicas across all servers, we must ensure that all writes are inserted in the same order. We address this via serialized PIR (§6). The algorithm for inserting into a blocked cuckoo hash table is also randomized, so the servers agree on a random seed, s_{cuckoo}, which they use to ensure they all make the same (pseudo-)random choices (if one or more servers make different choices, it affects liveness, but not privacy).

When the Talek client library dequeues a message, M, with sequence number $seqNo$ from its internal write queue, it runs the following $RealWrite(\tau, seqNo, M)$ algorithm:

$$\begin{aligned} \beta_1 &= PRF(\tau.k_{s1}, seqNo) \in [0..b-1] \\ \beta_2 &= PRF(\tau.k_{s2}, seqNo) \in [0..b-1] \\ data &= Enc_{\tau.k_{enc}}(seqNo|M) \end{aligned}$$

and sends the `Write` request, $\beta_1|\beta_2|data$, to the servers. If the write queue is empty, it instead runs $FakeWrite()$; i.e., it chooses β_1 and β_2 at random in $[0..b-1]$ and computes

$$data = Enc_{\tau.k_{enc}}(dummySeqNo|0^z)$$

i.e., the encryption of a reserved sequence number and z zeroes.

Upon receiving the `Write` request, each server follows the standard cuckoo insertion algorithm:

(1) Delete the n-th oldest element.
(2) Insert $\beta_1|\beta_2|data$ into either bucket β_1 or bucket β_2 if there is spare capacity in either bucket.
(3) If both buckets are full, choose $\beta_e \in \{\beta_1, \beta_2\}$, using randomness derived from s_{cuckoo}. Let $\delta_e = \beta_1|\beta_2|data$.
(4) 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 there.
    (c) Set δ_e to be the evicted value, and set β_e to its alternate bucket location.

5.4 Reading via PIR

When the Talek client library dequeues a log read at sequence number $seqNo$ from its internal read queue, it generates a `Read` request using the following $RealRead(\tau, seqNo)$ algorithm. The library performs a PIR read (following the protocol from §2) for bucket $PRF(\tau.k_{s1}, seqNo)$ (i.e., the first possible cuckoo location). The library attempts to decrypt every message in the bucket by computing $Dec_{\tau.k_{enc}}(data)$. If decryption succeeds and matches $seqNo$, M is returned to the application. If not, the next time a `Read` request should be issued (as determined by the randomized reading schedule), the library performs a PIR read for the second cuckoo location, i.e., $PRF(\tau.k_{s2}, seqNo)$.

If the internal read queue is empty, the library runs $FakeRead()$, which issues a PIR read for an arbitrary bucket.

5.5 Control Logs

To facilitate control messages between users, we establish a *control log* between every pair of users who want to communicate. We expect the log handle for the control log to be generated and exchanged out of band when users exchange and verify public keys.

When a user wants to give log access to one of their friends, they send the new handle via the control log shared with the intended subscriber. Once all subscribers have access to the log handle, the log writer can write messages once to be read by many subscribers, instead of sending individual messages through each pairwise control log. Revocation occurs by creating a new handle and sharing it with the non-revoked subscribers.

Control logs are also used by Talek to coordinate between users. If a user has been offline for an extended period, she can ask the writer for the most recent sequence number of a particular log. Similarly, users can send retransmission requests for missed messages. By definition, these requests leak information to the sender, but the sender and receiver are assumed to trust each other.

A client also uses a control log to periodically send heartbeat messages to itself. If these messages are lost, it serves as a hint to the client of a denial-of-service attack.

5.6 Configuration Considerations

Online clients issue `Read` and `Write` requests at rates of r and w respectively, as dictated by the random distribution (e.g., uniform or Poisson) chosen by the application developer. Ideally, the distribution should closely match the application's typical usage patterns. Choosing a more aggressive distribution (e.g., with more frequent reads and writes) will improve the user experience but incur higher protocol overheads due to extra cover traffic sent when no real requests are ready. Conversely, a more conservative distribution will waste fewer resources at the cost of potentially increasing the latency of real messages.

Talek is 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 the message is garbage collected. In this case, readers can request a retransmission (§5.5).

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. For example, the load factor for $d = 1$ is less than 0.5, such that one must allocate twice as much space as the number of items in the table. The load factor grows asymptotically towards 1 as d increases [36]. For values of $d > 3$, the load factor is over 95%. A high load factor translates to a more

densely packed table and cheaper PIR operations for the same number of messages in the database, n.

In Talek, the number of buckets b and the bucket depth d are fixed to match the expected client workloads. 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 enables cheaper PIR reads and smaller request vectors in PIR requests at the cost of larger network overhead for the response. 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.

5.7 Security Analysis

We formally prove the security of oblivious logging in §A.2. Informally, we assume that the writer trusts the consumers of its log and prove 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 [27]. For Write, we rely on the security properties of a PRF and our encryption algorithm. We use a symmetric authenticated encryption algorithm for message payloads. For any Write, the bucket locations are either generated by a PRF using the log handle's seed values, (k_{s1}, k_{s2}), or chosen at random. An attacker who can distinguish between these two cases immediately breaks the PRF's security.

A malicious client has the ability to impact service availability by deviating from the protocol and sending all its writes to a single bucket. This attack (and other forms of denial-of-service) is limited by servers enforcing the fixed Write rate w per client, the number of Sybil clients the attacker can obtain, and the size of the database, n. The attack is further mitigated by the self-balancing nature of cuckoo tables, where legitimate messages can be evicted to their alternate locations, and the fact that the malicious client will not know which buckets to target to disrupt a specific communication.

6 SERIALIZED PIR

Similar to Riffle [60], to reduce the network costs of Read requests for the client, we introduce a *serialized* version of IT-PIR, which offloads work from clients to the servers. Unlike Riffle's design, ours avoids the need for the servers to keep per-client state.

At a high level, we choose an arbitrary server to be the *leader*, $\mathscr{S}_0$, with the rest of the servers forming the *follower* set, $[\mathscr{S}_1, \ldots, \mathscr{S}_{l-1}]$. All Read and Write requests are directed to the leader. The leader adds a global sequence number to each request (to ensure operations are handled consistently across the servers) and forwards it to the followers. Each Read request sent to the leader contains a PIR query for each server, encrypted under its respective public key. The followers each send their response back to the leader (rather than the client). The response is carefully masked to preserve the confidentiality of each server's results while allowing the leader to combine them on behalf of the client. Overall, the client's outbound request bandwidth is the same as before, but the inbound bandwidth is only the size of a single response, rather than l responses.

To issue a serialized PIR request the client, $\mathscr{C}$ proceeds as follows.

(1) $\mathscr{C}$ generates b-bit PIR requests for each server, $\{q_0, \ldots, q_{l-1}\}$ (b is the number of server hash 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 followers along with a global sequence number.
(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 mask, $P_i = PRNG(p_i)$, from the seed parameter. This mask should the same size as R_i.
(7) Each server, $\mathscr{S}_i$, responds to $\mathscr{S}_0$ 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 mask, $P_0 \bigoplus \ldots \bigoplus P_{l-1}$

Security: With respect to privacy, this serialized variant of PIR is provably equivalent to the traditional PIR scheme described in §2. The proof is straightforward and sketched in §A.2. Hence, even if the leader is malicious, it can only undermine liveness, not privacy.

Correctness and Liveness: 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, it could cause those replicas to become inconsistent. Because any follower could also deny service by failing to respond or deviating from the protocol, the leader is in no more privileged a position to affect correctness or liveness of the system than any other server in the system. In §5.5 we describe how clients detect such attacks on availability. Furthermore, serialized PIR is compatible with the Byzantine-fault-tolerant varieties of PIR, which can be used to improve liveness guarantees (§3.2).

7 PRIVATE NOTIFICATIONS

Regularly polling for new messages presents two problems. First, because every user polls at the same rate, message latency increases as a user subscribes to more logs. Second, it is hard to know which log to poll at any given time.

Inspired by previous uses of Bloom filters for private membership queries [21, 42], we introduce a private notification system that allows users to determine when new messages have been published to a log without revealing the list of logs to which they subscribe. By detaching reads from notifications, clients can prioritize reads and reduce how often they read buckets for logs without new messages.

The private notifications system works as follows:

(1) With every Write request, clients send an *interest vector*, privately encoding the log handle and message sequence number of this request.
(2) Servers maintain a *global interest vector* which encodes the set of (log handle, message sequence number)-pairs for all messages on the server.
(3) Clients periodically query the lead server for the global interest vector.

We expand on these algorithms below and in Appendix B.

Computing an Interest Vector. During a $RealWrite(\tau, seqNo, M)$, the client creates an interest vector by inserting into an empty Bloom filter that uses *cryptographic* hash functions:

$$intVec \leftarrow BloomFilter()$$
$$intVec.insert(\tau.id|seqNo)$$

For a *FakeWrite* request, we insert a random value instead of $\tau.id|seqNo$.

Maintaing a Global Interest Vector. The global interest vector is the union of the interest vectors written to the server. Servers periodically sign their global interest vector and exchange signatures in a multi-signature scheme [18], which allows the lead server to aggregate all signatures into a single compact one. Upon a *GetUpdates*() request from the client, the leader sends that signature, along with the global interest vector, to the client for verification.

In practice, Talek employs compressed Bloom filters [70], and both clients and servers maintain a window of Bloom filter deltas [70], with each delta summarizing changes since the previous delta. By discarding old deltas as new ones arrive, we avoid saturating the global interest vector. Altogether, compressed Bloom filters with delta compression result in updates of less than 10k bytes for our experiments with 1M messages stored on the server. We set clients to fetch updates every 20 reads to further amortize this cost.

Security Analysis. The security of private notifications relies on the cryptographic hash functions (modeled as a random oracle) used in the Bloom filter. As long as we use a log ID with sufficient entropy, each interest vector provides a negligible advantage in the indistinguishability security game. We give a formal proof in §A.2 (Game 4).

Private notifications are only used to prioritize reads on the internal request queue. As such, it has no impact on Talek's security goals, since it has no visible effect on the network protocol interface. It simply reorders the internal schedule of private requests.

8 IMPLEMENTATION

To evaluate Talek's practicality, we have implemented a prototype in approximately 6,200 lines of code; the source code is available on GitHub.We have implemented 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, sharing memory between the CPU implementation and the GPU. The prototype uses SipHash [11] as the PRF, and NaCl's box API [15] for public and symmetric authenticated encryption. NaCl's API relies on a combination of Curve25519, Salsa20, and Poly1305.

9 EVALUATION

Our evaluation addresses the following questions:

- **9.2** What is the cost of operations for clients and servers?
- **9.3** What is the cost of cover traffic?
- **9.4** How does system performance scale with more users?
- **9.5** What is the end-to-end latency of messages?
- **9.6** How does Talek compare with previous work?

9.1 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 allocated two VMs to run user clients. Each user client issues periodic `Read` and `Write` requests to the server.

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 these latencies will not impact the results here which focus on the main bottleneck: the server-side computational cost of Talek.

To evaluate a realistic workload, we used the Ubuntu IRC logs from 2016 [3], consisting of 1554790 messages over 32,834 unique usernames. We generate a unique log in Talek for each writer in an IRC channel. We varied the number of users, $m \in (0, 32K]$, and the number of messages in the database, $n \in \{10K, 32K, 100K, 500K, 1M\}$. We fix the message size to z =1 KB. In order to understand the trade-off between bandwidth and end-to-end message latency, we vary the global client read and write rates.

	Messages on Server (n)		
	10K	100K	1M
Client CPU costs (μs)			
Generate new log handle	7753	7753	7753
Write	67	67	67
Issue PIR query	65	574	6888
Process PIR response	146	146	146
Server CPU costs (ms)			
PIR Read: CPU	1.34	11.10	88.10
PIR Read: GPU	0.07	0.54	4.36
Write	0.02	0.02	0.02
Server storage costs (MB)			
1 KB messages	24	241	2410
Network costs (KB)			
GetUpdates	0.21	1.40	14.00
Read request	0.96	9.39	93.72
Read response	4.16	4.16	4.16
Write request	1.08	1.08	1.08

Figure 6: **Cost of Individual Talek Operations.** We vary the number, n, of 1 KB messages stored on the server.

9.2 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 number of messages on the server, $n \in \{10K, 100K, 1M\}$. 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 6 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 generating a PIR query for `Read` 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 streams the database through the local CPU cache while accumulating responses to service each batch of queries. Its performance is limited by memory-bus throughput. The GPU implementation accelerates performance by 1–2 orders of magnitude

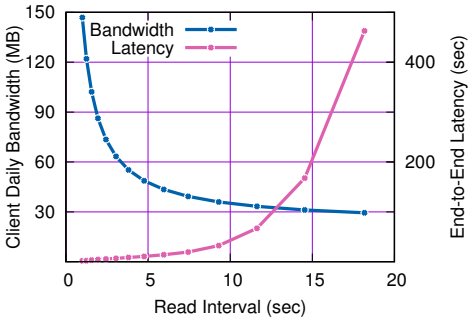

Figure 7: **Impact of Read Interval.** Average daily bandwidth per client and end-to-end message latency as a function of read interval when replaying Ubuntu IRC logs from 2016. When clients are configured to send messages once per second, each client sends/receives ~148 MB per day to achieve end-to-end message latencies < 2 seconds. As we increase the interval of the read schedule, more read requests represent real work, at the expense of end-to-end latency.

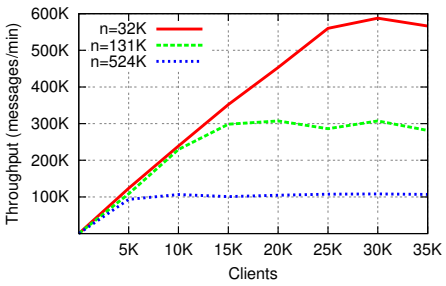

Figure 8: **Talek's Throughput.** Throughput of the system when varying the number of real clients. Each client independently issues read and write requests every 5 seconds. Each line represents a different value for n, the number of messages on the server. Larger values of n require scanning a larger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

by taking advantage of the inherent parallelism of PIR operations across many GPU cores and the optimized on-device memory hierarchy. Unless otherwise stated, henceforward, results are for our GPU implementation. Batch coding or preprocessing [65] would further improve this throughput bottleneck, by an estimated 3×, at the cost of higher latency for writes to become visible to subsequent reads. Writes incur negligible cost compared to the cost of reads.

Storage costs scale as expected, since our current implementation stores all messages twice. 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. 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 0.02 and a TTL of 100 write intervals. This cost is independent of message size. Updates trade off bandwidth with 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.

9.3 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 message latency. To quantify this tradeoff, we ran an experiment replaying the Ubuntu IRC logs into Talek. We model clients as mobile devices that only send requests when the device is online. Studies have shown that users are engaged with their mobile devices only 8.6% of the time [25]. We configured servers to store 524,000 messages, such that servers can keep up with read requests if each client reads one message per second. We then configure client write schedules to maintain a message TTL of one day on the servers.

Figure 7 shows the effect of increasing client read intervals. When clients are configured to read every second, average end-to-end message latency is 1.71 seconds. The latency exceeds the read interval, since many clients subscribe to multiple logs, so a burst of writes from different writers may be bottlenecked by the read interval. This latency comes at the cost of every client using 148 MB of bandwidth per day. For comparison, the installer for Adobe Reader is ~200 MB, and the top ten most downloaded apps in 2018 on the Google Play store [86] range from 13-66 MB. If daily bandwidth usage is of concern, we can drastically reduce it by increasing the read interval by a few seconds, while keeping latencies below 10 seconds. At higher read intervals, we reach optimal network usage, where most read requests corresponds to real work. However, as read requests are increasingly rate-limited, message latencies grow quickly. In practice, application developers will need to choose read intervals based on acceptable usability and network costs. This design decision will depend on application workloads.

In future sections, we fix the read interval to 5 seconds, balancing network and latency considerations. At this read interval, end-to-end message latencies are 10.7 seconds, about half of read requests are *FakeRead* requests, and each client will send about 49.4 MB per day over the network. For comparison, Snapchat, the #2 top downloaded mobile app for both iOS and Android, consumes 14-86 MB per day per client [1]. Assuming the system supports 32,000 active users, servers will need a network connection of at least 18.3 MB/s.

9.4 Throughput

To understand Talek's peak performance, we experimented 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 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

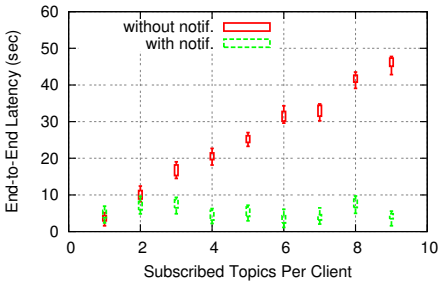

Figure 9: **Talek's Latency.** End-to-end latency of message delivery when a client subscribes to multiple logs. Each data point represents 20 trials. With notifications, clients prioritize logs with new messages.

with numbers of clients, m, such that $m < n$, corresponding to a message lifetime of at least one round of reads.

To further improve server performance, each Talek "server" could internally consist of multiple machines, each handling a portion of the PIR database. We defer this optimization to future work.

9.5 End-to-End Latency with Notifications

To understand the latency of message delivery, we used the same messaging workload as in the throughput experiment. We measure the end-to-end time for a message sent by the sender to be seen, varying the number of logs to which the receiver client is subscribed. The spread of each value over 20 trials reflects the read and write rates. Figure 9 shows the results with and without private notifications. When notifications are off, the client must poll each log in a round robin fashion. Because the read rate is fixed, the end-to-end latency grows linearly with the number of subscribed logs. With private notifications, the receiver learns which logs have new messages, allowing it to prioritize that read. As a result, the end-to-end latency for a single message is relatively fixed.

9.6 Comparison with Prior Work

Talek explores a new design point in the space of private messaging. To understand the performance implications, we compare Talek's performance to representatives of other interesting design points using their publicly available implementations. Pung [10] and its refinement [9], which we dub Pung++, share a similar security goal with Talek, but adopt a stronger threat model incompatible with IT-PIR. Riposte [28] shares Talek's threat model and employs "reverse" IT-PIR, but offers anonymous broadcast (i.e., senders are anonymous but no private read operation is supported) and targets a weaker privacy goal (k-anonymity). Vuvuzela [89] shares Talek's threat model but targets another (weaker) privacy goal (differential privacy). We describe additional protocol differences in §10.

In Figure 10, we summarize the asymptotic complexity of each system. These asymptotics, however, hide vastly different constants.

Hence, Figure 11 shows the concrete single-threaded throughput of each system. For each system, we sanity checked that our results were consistent with those reported by the authors. For Pung, we use a single server. For Talek, Vuvuzela, and Riposte, systems that rely on an anytrust threat model, we use a common security parameter of 3 servers. In these systems, larger numbers of servers only improves security, not performance. Write performance is similar between Talek, Pung++, and Vuvuzela, which all dominate Riposte due to its $O(\sqrt{n})$ cost. Read performance, however, demonstrates further tradeoffs. Pung++'s throughput lags Talek's CPU implementation by up to two orders of magnitude, which in turn lags Vuvuzela by 1-3 orders of magnitude. The Riposte implementation does not include an implementation for reads due to its focus on broadcast applications. Pung++ reads are more expensive predominantly due to its use of homomorphic encryption for C-PIR (to support a stronger threat model) and differences in its read protocol. Vuvuzela's weaker security goal allows it to scale better than Talek. Talek's GPU implementation improves read performance by $\sim 20\times$.

		Talek	Pung++	Riposte	Vuvuzela
Client CPU	Read	$O(ln)$	$O(\sqrt{n})$	✗	$O(l)$
	Write	$O(1)$	$O(1)$	$O(\sqrt{n})$	$O(l)$
Total Server CPU	Read	$O(ln)$	$O(n)$	✗	$O(\frac{l^2+ln}{n^2})$
	Write	$O(l)$	$O(1)$	$O(ln)$	$O(\frac{l^2+ln}{n^2})$
Total Server Storage		$O(ln)$	$O(n)$	$O(ln)$	$O(l+n)$
Network	Read request	$O(ln)$	$O(\sqrt{n})$	✗	$O(1)$
	Read response	$O(d)$	$O(1)$	✗	$O(1)$
	Write	$O(1)$	$O(1)$	$O(l\sqrt{n})$	$O(1)$

Figure 10: **Asymptotic Comparison.** We compare costs between related work that uses an anytrust or stronger threat model. Parameters are the number of servers, l, the number n of client messages in the system and, for Talek, the number of messages in a bucket d. Riposte does not specify a read mechanism. Client CPU is for one read/write request. Total server CPU is the total cost of l servers (or for Pung++, a single server) for one read or write request. Network costs are for read/write requests between the client and the server(s). Pung++'s costs do not include constructing and sending an oracle that maps message indices to locations within a bucket (§10), while Talek's costs do not include private notifications, as both are orthogonal to the main design.

10 RELATED WORK

We provide an overview of related systems that hide communication patterns. Unger et al. provide a more detailed survey [88]. For convenience, Figure 12 summarizes the discussion.

PIR-based systems. Theoretical work proves that the collective work required by the servers to answer a client's request scales at least linearly with the size of the database [12]. While it is possible to circumvent this lower bound using database preprocessing [12] or an offline/online model [30], we cannot take advantage of such approaches since our database changes constantly.

On the systems side, some work in this space leverages assumptions about specific application workloads to make PIR practical in a particular setting. For example, DP5 [19] is a private chat presence system. The security of the protocol depends on chat presence

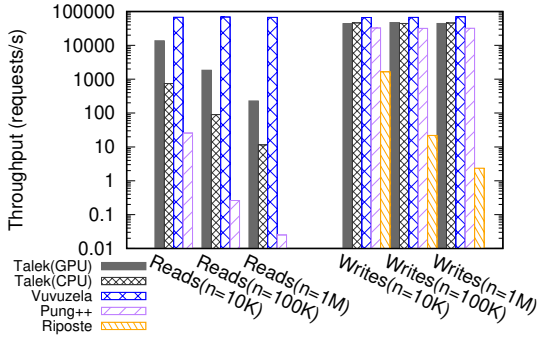

Figure 11: **Performance Comparison** of `Read` and `Write` handlers of various systems. Because Riposte is a broadcast protocol, it does not include a read operation. Note the log scale on the y-axis.

System	Security Goal	Threat Model	Technique	Application
Talek	indisting.	≥ 1	IT-PIR	group msg.
Pynchon [79]	k-anon.	≥ 1	mixnet/IT-PIR	email
Riffle [60]	k-anon.	≥ 1	mixnet/IT-PIR	file-sharing
Riposte [28]	k-anon.	≥ 1	IT-PIR	broadcast
Dissent [29]	k-anon.	≥ 1	DC-net	broadcast
Atom [58]	k-anon.	$\geq f$	mixnet	broadcast
Vuvuzela [89]	diff. privacy	≥ 1	mixnet	1–1 msg.
Stadium [87]	diff. privacy	$\geq f$	mixnet	1–1 msg.
Karaoke [61]	diff. privacy	$\geq f$	mixnet	1–1 msg.
Pung [9, 10]	indisting.	0	C-PIR	group msg.
ORAM [82–84]	indisting.	0	ORAM	storage
DP5 [19]	indisting.	≥ 1	IT-PIR	chat presence
Popcorn [50]	indisting.	≥ 1	C-PIR/IT-PIR	video stream
XRD [59]	indisting.	$\geq f$	mixnet	1–1 msg.

Figure 12: **Comparison of Privacy Systems.** Indistinguishability-based security goals offer the strongest level of privacy. Systems based on k-anonymity and differential privacy leak information over time [33, 56, 67]. The threat model column denotes the number of servers that must be honest for security properties to hold; systems marked with f require a fraction (e.g., 20-80%) of servers to be honest.

workloads and does not generalize to group messaging. Similarly, Popcorn [50] uses both C-PIR and IT-PIR to construct a private read-only video streaming system over a static video database.

Some systems support general-purpose anonymous writing, but do not support private reading. Ostrovsky and Shoup propose private information storage (PIS) [73], which allows a client to write to a row in a database of n rows without revealing which row was updated. Writing via PIS is expensive, incurring poly-logarithmic communication, compared with $O(1)$ for Talek. Riposte [28] expands on this work to support a scalable broadcast system. Riposte and Talek share a similar anytrust threat model, but Riposte has a weaker security goal based on k-anonymity within a round of communication. Riposte does not promise privacy over multiple rounds of communication, and writes require $O(\sqrt{n})$ messages. Other systems, like Talek, aim for general-purpose private messaging, and hence support a wider range of applications. For instance, Pynchon Gate [79] is system where emails are sent to servers via mixnets. Emails are dumped daily to distributor servers, where clients use IT-PIR to privately retrieve them. While PIR hides which messages clients read, the email server stores the communication patterns between email addresses. Riffle [60] uses mixnets to send messages and IT-PIR to retrieve them. Riffle provides k-anonymity; in each round, the adversary learns that a message originated from 1 of k users. As discussed in §3, in practice, k-anonymity typically provides weaker privacy than access sequences indistinguishability.

Pung [10] (and its refinement [9]) supports a key-value store based on C-PIR with a security goal of UO-ER, which is similar to Talek's access sequence indistinguishability (see §3). Pung targets a stronger threat model than Talek; Pung assumes all servers are untrusted. Hence, Pung only requires one server to provide functionality compared with l for Talek. Pung's stronger threat model comes at the cost of performance. Pung uses an implementation of C-PIR for reads called SealPIR [7, 9], which leverages the SEAL homomorphic encryption library [80] based on the Fan-Vercauteren fully-homomorphic encryption (FHE) system [43]. Thus, Pung incurs orders of magnitude higher computational and network costs, compared with Talek.

Additionally, Pung uses an interactive binary search algorithm for retrieval, requiring $O(log(n))$ round trips between client and server, compared to the $O(1)$ cuckoo table lookup in Talek. Thus, to retrieve a 1 KB message from a database size of $n = 32$K messages, Pung requires >36 MB of data. Pung++ reduces this overhead by leveraging a more efficient C-PIR protocol, probabilistic batch coding, and *reverse* cuckoo hashing. The latter places each element into *all* candidate buckets, and requires each client to have an oracle to tell it which index within a bucket contains the message it wants to retrieve. The authors suggest instantiating the oracle by having the client retrieve a Bloom filter that encodes the index of every message. Hence, the client must download $O(n)$ data each time it wants to discover the location of new messages. Hence, Pung++ still requires more than >1 MB of data for the same workload, while Talek makes 2 requests and transfers <12 KB (10× less).

Mixnet-based Systems. Chaum mixnets [23, 24, 49, 54] and verifiable cryptographic shuffles [20, 44, 71] are a way to obfuscate the source of a message. As with PIR, some mixnet-based systems provide anonymous broadcasting, rather than private messaging. For example, Atom [58] implements anonymous broadcast by dividing the servers into multiple groups and then repeatedly shuffling a batch of ciphertexts within each group and forwarding parts of the batch to neighboring groups. Atom assumes that some fraction of servers are honest. In particular, Atom's security relies on each group having at least one honest server.

Mixnets have been applied to scalable private messaging [32], 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 systems [37, 55, 77] can also be used to access an encrypted database with similar limitations.

Using differential privacy analysis, Vuvuzela [89], Stadium [87], and Karaoke [61] formalize the amount of noise that honest shufflers would need to inject to bound information leakage at the database. These systems have a weaker security goal than Talek – which provides indistinguishability even under an active adversary, but they offer substantially better performance than Talek. Vuvuzela scales to millions of users with a peak throughput of nearly 4M messages/min using the same number of servers as Talek. Stadium [87] achieves Vuvuzela's security goal with even better performance, but it weakens the threat model to assume that some fraction (the authors suggest 50-75%) of servers behave honestly. Karaoke [61] extends Vuvuzela with an efficient noise verification technique and a security goal of optimistic indistinguishability, where no information is leaked under a passive adversary, but the system falls back to differential privacy under an active attack. Similar to Stadium, Karaoke assumes some fraction (the authors suggest 60-80%) of servers behave honestly.

Like Stadium and Karaoke, XRD [59] is a mix-net system that assumes a fraction of servers behave honestly and requires similarly large fraction to achieve good performance. In contrast to Stadium and Karaoke, XRD offers cryptographic privacy similar to the guarantees provided by Talek.

Stadium, Karaoke, and XRD include support for horizontal scaling (linear scaling for Stadium and Karaoke, square root scaling for XRD), so that adding more servers improves the system's throughput. As discussed in §9.4, Talek is compatible with instantiating each "server" via multiple machines to provide better throughput. This provides a subtly different form of scaling, however. Stadium,

Karaoke, and XRD can increase performance by increasing the number of participating organizations, while Talek increases the number of machines held by the participating organizations. The difference stems from Talek's anytrust threat model.

DC-nets. DC-net systems [22, 48], like Herbivore [81] and Dissent [29, 31, 90], are a method for anonymously broadcasting messages to a group using information-theoretic techniques. For each message, all clients must broadcast random bits to every other client. DC-nets enable effective broadcast messaging, but they are not optimal for group messaging due to high network costs.

Oblivious RAM (ORAM). ORAM [46, 47, 72] allows 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, $\Omega(\log n)$, and constant data reshuffling make ORAM costly for systems with many users sharing data.

ACKNOWLEDGEMENTS

This work was supported in part by grants from Google, the Alfred P. Sloan Foundation, and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work was also supported by DARPA and NIWC under contract N66001-15-C-4065. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

11 CONCLUSION

We have explored a new point in the design space of private group messaging: a very strong security guarantee based on access sequence indistinguishability coupled with an only slightly weakened anytrust threat model. The result is a new built system, Talek, which incorporates a careful series of design decisions and new optimizations, including IT-PIR based on blocked cuckoo hashing, serialized PIR, private notifications, and GPU-based acceleration. Together, these demonstrate that this design point supports strong performance for realistic workloads.

REFERENCES

[1] 2015. New Data Shows Mobile Data Consumption Skyrocketing Following Snapchat DiscoverâĂŹs Launch. techcrunch.com.
[2] 2016. Pond. github.com/agl/pond.
[3] 2016. Ubuntu IRC Logs. https://irclogs.ubuntu.com/.
[4] 2017. Ricochet: Anonymous peer-to-peer instant messaging. github.com/ricochet-im/ricochet.
[5] 2018. Keybase. https://keybase.io/.
[6] 2018. Signal Privacy Policy. whispersystems.org.
[7] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR: Private information retrieval for everyone. *Privacy Enhancing Technologies* (2016).
[8] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. 2017. MCMix: Anonymous messaging via secure multiparty computation. In *26th USENIX Security Symposium*. 1217–1234.
[9] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2018. PIR with compressed queries and amortized query processing. In *Symposium on Security and Privacy*. IEEE.
[10] Sebastian Angel and Srinath Setty. 2016. Unobservable Communication over Fully Untrusted Infrastructure. In *Operating Systems Design and Implementation*. USENIX.
[11] Jean-Philippe Aumasson and Daniel J Bernstein. 2012. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*. Springer, 489–508.
[12] Amos Beimel, Yuval Ishai, and Tal Malkin. 2000. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Proceedings of the Annual International Cryptology Conference*.
[13] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. 1996. Keying hash functions for message authentication. In *Annual International Cryptology Conference*. Springer.
[14] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. 1998. Relations among notions of security for public-key encryption schemes. In *Annual International Cryptology Conference*. Springer, 26–45.
[15] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 159–176.
[16] Alexandra Boldyreva, Vipul Goyal, and Virendra Kumar. 2008. Identity-based encryption with efficient revocation. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 417–426.
[17] Dan Boneh and Matt Franklin. 2001. Identity-based encryption from the Weil pairing. In *Annual International Cryptology Conference*. Springer, 213–229.
[18] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. 2003. Aggregate and verifiably encrypted signatures from bilinear maps. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 416–432.
[19] Nikita Borisov, George Danezis, and Ian Goldberg. 2015. DP5: A Private Presence Service. *Proceedings on Privacy Enhancing Technologies* 2015, 2 (2015), 4–24.
[20] Justin Brickell and Vitaly Shmatikov. 2006. Efficient Anonymity-Preserving Data Collection. In *International Conference on Knowledge Discovery and Data Mining*. ACM.
[21] Yan-Cheng Chang and Michael Mitzenmacher. 2005. Privacy preserving keyword searches on remote encrypted data. In *International Conference on Applied Cryptography and Network Security*. Springer, 442–455.
[22] David Chaum. 1988. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *Journal of Cryptology* 1, 1 (1988), 65–75.
[23] David Chaum, Farid Javani, Aniket Kate, Anna Krasnova, Joeri de Ruiter, and Alan T Sherman. 2017. cMix: Anonymization by High-Performance Scalable Mixing. In *International Conference on Applied Cryptography and Network Security*. Springer, 557–578.
[24] David L Chaum. 1981. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM* 24, 2 (1981), 84–90.
[25] Xiaomeng Chen, Ning Ding, Abhilash Jindal, Y Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015. Smartphone energy drain in the wild: Analysis and implications. *ACM SIGMETRICS Performance Evaluation Review* 43, 1 (2015), 151–164.
[26] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 41–50.
[27] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *Journal of the ACM (JACM)* 45, 6 (1998), 965–981.
[28] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. 2015. Riposte: An Anonymous Messaging System Handling Millions of Users. In *Symposium on Security and Privacy*. IEEE.
[29] Henry Corrigan-Gibbs and Bryan Ford. 2010. Dissent: Accountable Anonymous Group Messaging. In *Proceedings of Computer and Communications Security*. ACM.
[30] Henry Corrigan-Gibbs and Dmitry Kogan. 2019. Private Information Retrieval with Sublinear Online Time. Cryptology ePrint Archive, Report 2019/1075.
[31] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. 2013. Proactively Accountable Anonymous Messaging in Verdict. In *USENIX Security*.
[32] George Danezis, Roger Dingledine, and Nick Mathewson. 2003. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *Symposium on Security and Privacy*. IEEE.
[33] George Danezis and Andrei Serjantov. 2004. Statistical Disclosure or Intersection Attacks on Anonymity Systems. In *International Workshop on Information Hiding*.
[34] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. 2018. PIR-PSI: Scaling Private Contact Discovery. *IACR Cryptology ePrint Archive* 2018 (2018), 579. https://eprint.iacr.org/2018/579
[35] Casey Devet, Ian Goldberg, and Nadia Heninger. 2012. Optimally Robust Private Information Retrieval. In *USENIX Security*.
[36] Martin Dietzfelbinger and Christoph Weidling. 2007. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. *Theoretical Computer Science* 380, 1 (2007), 47–68.
[37] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. *Tor: The Second-Generation Onion Router*. Technical Report. DTIC Document.
[38] Colin Dixon, Thomas E Anderson, and Arvind Krishnamurthy. 2008. Phalanx: Withstanding Multimillion-Node Botnets. In *USENIX Symposium on Networked Systems Design and Implementation*.

[39] Cynthia Dwork. 2006. Differential Privacy. In *Automata, languages and programming*. 1–12.
[40] Cynthia Dwork. 2008. Differential Privacy: A Survey of Results. In *Theory and applications of models of computation*. 1–19.
[41] Anthony Ephremides, Jeffrey E Wieselthier, and Dennis J Baker. 1987. A Design Concept for Reliable Mobile Radio Networks with Frequency Hopping Signaling. *Proc. IEEE* 75, 1 (1987), 56–73.
[42] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. ACM, 1054–1067.
[43] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144.
[44] Jun Furukawa and Kazue Sako. 2001. An Efficient Scheme for Proving a Shuffle. In *Advances in Cryptology (CRYPTO 2001)*. 368–387.
[45] Ian Goldberg. 2007. Improving the Robustness of Private Information Retrieval. In *Security and Privacy (SP), 2007 IEEE Symposium on*. IEEE, 131–148.
[46] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Symposium on Theory of Computing*. ACM.
[47] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.
[48] Philippe Golle and Ari Juels. 2004. Dining Cryptographers Revisited. In *Advances in Cryptology (Eurocrypt 2004)*. 456–473.
[49] Ceki Gülcü and Gene Tsudik. 1996. Mixing E-mail with Babel. In *Network and Distributed System Security*. ISOC.
[50] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath TV Setty, Lorenzo Alvisi, and Michael Walfish. 2016. Scalable and Private Media Consumption with Popcorn. In *Symposium on Networked Systems Design and Implementation*. USENIX. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/gupta-trinabh
[51] Alejandro Hevia and Daniele Micciancio. 2008. An indistinguishability-based characterization of anonymous channels. In *Privacy Enhancing Technologies Symposium*. Springer, 24–43.
[52] Jason I Hong and James A Landay. 2004. An Architecture for Privacy-Sensitive Ubiquitous Computing. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*. ACM, 177–189.
[53] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.. In *NDSS*, Vol. 20. 12.
[54] Anja Jerichow, Jan Muller, Andreas Pfitzmann, Birgit Pfitzmann, and Michael Waidner. 1998. Real-time Mixes: A Bandwidth-Efficient Anonymity Protocol. *Selected Areas in Communications, IEEE Journal on* 16, 4 (1998), 495–509.
[55] Nicholas Jones, Matvey Arye, Jacopo Cesareo, and Michael J Freedman. 2011. Hiding Amongst the Clouds: A Proposal for Cloud-based Onion Routing. In *FOCI*.
[56] Dogan Kedogan, Dakshi Agrawal, and Stefan Penz. 2002. Limits of Anonymity in Open Environments. In *International Workshop on Information Hiding*. 53–69.
[57] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*. IEEE, 364–373.
[58] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. 2017. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 406–422.
[59] Albert Kwon, David Lu, and Srinivas Devadas. 2020. XRD: Scalable Messaging System with Cryptographic Privacy. In *To appear in Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[60] Young Hyun Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. 2016. Riffle: An Efficient Communication System with Strong Anonymity. In *Proceedings on Privacy Enhancing Technologies*.
[61] David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2018. Karaoke: Distributed private messaging immune to passive traffic analysis. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 711–725.
[62] David Lazar and Nickolai Zeldovich. 2016. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Operating Systems Design and Implementation*. USENIX.
[63] Sarah Jamie Lewis. 2018. Cwtch: Privacy Preserving Infrastructure for Asynchronous, Decentralized, Multi-Party and Metadata Resistant Applications. https://cwtch.im/.
[64] Jinyuan Li, Maxwell Krohn, David Mazieres, and Dennis Shasha. 2004. SUNDR: Secure Untrusted Data Repository. In *USENIX Symposium on Operating Systems Design and Implementation*.
[65] Wouter Lueks and Ian Goldberg. 2015. Sublinear Scaling for Multi-Client Private Information Retrieval. In *International Conference on Financial Cryptography and Data Security*. 168–186.
[66] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. 2011. Depot: Cloud Storage with Minimal Trust. *ACM Transactions on Computer Systems (TOCS)* 29, 4 (2011), 12.
[67] Nick Mathewson and Roger Dingledine. 2004. Practical Traffic Analysis: Extending and Resisting Statistical Disclosure. In *Privacy Enhancing Technologies*.
[68] Susan E McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. 2015. Investigating the Computer Security Practices and Needs of Journalists.. In *USENIX Security*.
[69] Susan E McGregor, Franziska Roesner, and Kelly Caine. 2016. Individual versus Organizational Computer Security and Privacy Concerns in Journalism. *Privacy Enhancing Technologies* (2016).
[70] Michael Mitzenmacher. 2002. Compressed bloom filters. *Transactions on Networking* (2002).
[71] C Andrew Neff. 2003. Verifiable Mixing (Shuffling) of ElGamal Pairs. http://www.votehere.net/vhti/documentation/egshuf.pdf. *VHTi Technical Document, VoteHere, Inc* (2003).
[72] Rafail Ostrovsky. 1990. Efficient Computation on Oblivious RAMs. In *Symposium on Theory of Computing*. ACM.
[73] Rafail Ostrovsky and Victor Shoup. 1997. Private Information Storage. In *Symposium on Theory of Computing*. ACM. https://doi.org/10.1145/258533.258606
[74] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo Hashing. In *European Symposium on Algorithms*. 121–133.
[75] Andreas Pfitzmann and Marit Hansen. 2010. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management (Version 0.34). https://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf.
[76] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. 2017. The Loopix anonymity system. In *26th USENIX Security Symposium*. 1199–1216.
[77] Michael G Reed, Paul F Syverson, and David M Goldschlag. 1998. Anonymous Connections and Onion Routing. *Selected Areas in Communications* (1998).
[78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978).
[79] Len Sassaman, Bram Cohen, and Nick Mathewson. 2005. The Pynchon Gate: A Secure Method of Pseudonymous Mail Retrieval. In *Workshop on Privacy in the Electronic Society*. ACM.
[80] SEAL 2018. Simple Encrypted Arithmetic Library (release 3.1.0). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.
[81] Emin Gün Sirer, Sharad Goel, Mark Robson, and DoÇğan Engin. 2004. Eluding Carnivores: File Sharing with Strong Anonymity. In *Workshop on ACM SIGOPS European Workshop*. ACM, 19.
[82] Emil Stefanov and Elaine Shi. 2013. Multi-Cloud Oblivious Storage. In *SIGSAC conference on Computer & Communications security*. ACM.
[83] Emil Stefanov and Elaine Shi. 2013. ObliviStore: High Performance Oblivious Cloud Storage. In *Security and Privacy*. IEEE.
[84] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an Extremely Simple Oblivious RAM Protocol. In *SIGSAC Conference on Computer & Communications Security*. ACM.
[85] Latanya Sweeney. 2002. k-anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 05 (2002), 557–570.
[86] Anna Tobin. 2018. These Were The Most Downloaded And Profitable Games and Apps Of 2018. https://www.forbes.com/sites/annatobin/2018/12/21/these-were-the-most-downloaded-and-profitable-games-and-apps-of-2018/.
[87] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. 2017. Stadium: A Distributed Metadata-Private Messaging System. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 423–440.
[88] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. 2015. SoK: Secure Messaging. In *Security and Privacy*. IEEE.
[89] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Symposium on Operating Systems Principles*. ACM.
[90] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. 2012. Dissent in Numbers: Making Strong Anonymity Scale. In *Operating Systems Design and Implementation*. USENIX.
[91] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. 2012. Scalable anonymous group communication in the anytrust model. In *Proceedings of the European Workshop on System Security (EuroSec)*.

A SECURITY PROOFS

A.1 Building Blocks

We provide formal definitions of the cryptographic building blocks that we use in the security proof of our construction.

We define a Private Information Retrieval scheme as specified by Chor et al. [26], adapted to allow up to t servers to collude (whereas they assume no collusion). The scheme allows a user, on a desired index i and a random input r of length l_{rnd}, to produce k queries of length l_q (one for each server). The servers respond according to the strategies $A_1, \ldots, A_k$ with replies of length l_a. The user then reconstructs the desired bit of the database based on these replies.

Definition A.1. Private Information Retrieval âĂŞ One-Round Schemes. A k-server Private Information Retrieval (PIR) scheme for a database of length n consists of:

- k query functions, $Q_1, \ldots, Q_k : [n] \times \{0,1\}^{l_{rnd}} \to \{0,1\}^{l_q}$
- k answer functions, $A_1, \ldots, A_k : \{0,1\}^n \times \{0,1\}^{l_q} \to \{0,1\}^{l_a}$
- a reconstruction function, $R : [n] \times \{0,1\}^{l_{rnd}} \times (\{0,1\}^{l_a})^k \to \{0,1\}$

These functions should satisfy

Correctness: For every $x \in \{0,1\}^n, i \in [n]$, and $r \in \{0,1\}^{l_{rnd}}$

$$R(i, r, A_1(x, Q_1(i,r)), \ldots, A_k(x, Q_k(i,r))) = x_i$$

t-Privacy: For every $i, j \in [n], (q_1, \ldots, q_t) \in (\{0,1\}^{l_q})^t$ and $\{s_1, \ldots, s_t\} \subset \{1, \ldots, k\}$

$$\mathbf{Pr}((Q_{s_1}(i,r), \ldots, Q_{s_t}(i,r)) = (q_1, \ldots, q_t)) =$$
$$\mathbf{Pr}((Q_{s_1}(j,r), \ldots, Q_{s_t}(j,r)) = (q_1, \ldots, q_t))$$

where the probabilities are taken over a uniformly chosen $r \in \{0,1\}^{l_{rnd}}$.

Note that while this definition is for a one-round PIR scheme, it can be extended to multiple rounds where the queries in each round are allowed to depend on answers received in previous rounds.

We use the following standard definitions of a pseudorandom function and the IND-CCA security of the encryption scheme:

Definition A.2. Pseudorandom function

A function $F : \{0,1\}^n \times \{0,1\}^{m_1} \to \{0,1\}^{m_2}$ is called a pseudorandom function (PRF) if it satisfies the following:

- For all $s \in \{0,1\}^n$ and all $x \in \{0,1\}^{m_1}$, $F(s,x)$ can be computed in polynomial time.
- Any PPT adversary $\mathscr{A}$ is successful in the following game with probability at most $\frac{1}{2} + negl(n)$, where $negl(\cdot)$ is a negligible function:
  - The challenger chooses $s \leftarrow \{0,1\}^n$ and a bit $b \leftarrow \{0,1\}$ at random.
  - For $i = 1, \ldots, q$, where q is polynomial in n, D chooses $x_i \in \{0,1\}^{m_1}$ and sends it to the challenger.
  - If $b = 0$, the challenger replies with $F(s, x_i)$. Otherwise, if x_i has not been queried before, the challenger picks $y_i \in \{0,1\}^{m_2}$ uniformly at random and sends it to $\mathscr{A}$. If x_i has been queried previously, the challenger sends the same response as the last time x_i was queried.
  - $\mathscr{A}$ outputs $b^* \in \{0,1\}$ and wins if $b^* = b$.

Definition A.3. IND-CCA security

An encryption scheme (Gen, Enc, Dec) is said to be IND-CCA secure if any PPT adversary $\mathscr{A}$ is successful in the following game with the probability at most $\frac{1}{2} + negl(n)$, where $negl(\cdot)$ is a negligible function:

- The challenger generates keys $(pk, sk) \leftarrow Gen(1^n)$.
- $\mathscr{A}$ receives pk as input.
- $\mathscr{A}$ gets a black-box access to $Dec_{sk}(\cdot)$.
- $\mathscr{A}$ chooses x_0, x_1.
- The challenger chooses $b \leftarrow \{0,1\}$ at random and gives $\mathscr{A}$ the challenge ciphertext $c = Enc_{pk}(x_b)$.
- $\mathscr{A}$'s access to $Dec_{sk}(\cdot)$ is now restricted - $\mathscr{A}$ is not allowed to ask for the decryption of c.
- $\mathscr{A}$ outputs $b^* \in \{0,1\}$ and wins if $b = b^*$.

Note that authenticated encryption implies single-message IND-CCA security, and single-message IND-CCA security implies multi-message IND-CCA security.

Finally, we are using the random oracle model where all parties have access to a random oracle, which is defined as follows:

Definition A.4. Random oracle

Given a security parameter n and a length function $l_{out}(\cdot)$, a random oracle R is a map from $\{0,1\}^*$ to $\{0,1\}^{l_{out}(n)}$.

A.2 Proof of Access Sequence Indistinguishability

We provide a security proof for Talek's protocol by reduction to the cryptographic assumptions provided in Appendix A.1. Definitions of *RealWrite* and *FakeWrite* are provided in §5.3. *RealRead*, *FakeRead* are defined in §5.4. Finally, interest vectors (addressed in **Game 4**) are defined in §7 .

We consider a series of games adapted from the game in §3.2, 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: This is the original game defined in §3.2 with an adversary $\mathscr{A}$ that chooses m challenger clients, and submits sequences with α_0 calls to *RealRead*, and α_1 calls to *RealWrite*, using the protocol defined in the paper. All *RealRead*s and *FakeRead*s from the client trigger *ProcessRead* on the servers. All *RealWrite*s and *FakeWrite*s from the client trigger *ProcessWrite* on the servers. Thus, the adversary has control over the messages sent by the clients over the network. We intend to show that the adversary's advantage in this game is negligible.

Game 1: (PIR *Read*) This game is as above, except that in each $RealRead(\tau, seqNo)$, we replace the real read location (specifically the bucket $PRF(\tau.k, seqNo)$) by the index of an arbitrary bucket, as specified in *FakeRead*. Let $\epsilon^{PIR}(\lambda_0, n)$ bound the advantage of an adversary breaking the PIR assumption in n calls to PIR read (via *RealRead*) with security parameter λ_0.

Given an adversary $\mathscr{G}$ that distinguishes between **Game 0** and **Game 1** and makes n calls to *RealRead*, we can construct an adversary $\mathscr{P}$ on the security of the n-round PIR scheme. $\mathscr{P}$ starts by sending the set of indices of the servers, controlled by $\mathscr{G}$ to its own challenger $\mathscr{C}_P$. Then, $\mathscr{P}$ behaves like a challenger to $\mathscr{G}$, following the description of the access sequence indistinguishability game, except for the case where a read request must be issued and the read queue is not empty (thus, a $RealRead(\tau, seqNo)$ is executed).

In this case, $\mathscr{P}$ computes $i := PRF(\tau.k, seqNo)$, where k is equal to k_{s1} or k_{s2}, depending on whether the first or the second cuckoo location must be read; $\mathscr{P}$ also chooses j at random in $[0..b-1]$. Then, $\mathscr{P}$ forwards i and j to $\mathscr{C}_P$. Upon receiving the $l-1$ PIR request queries from its challenger in response, $\mathscr{P}$ forwards these to $\mathscr{G}$. Upon $\mathscr{G}$ ending the game, $\mathscr{P}$ responds to its own challenger $\mathscr{C}_P$ with i, if $\mathscr{G}$'s response was **Game 0**, and j, if $\mathscr{G}$'s response was **Game 1**. Note that i corresponds to the real location specified by the *RealRead* request, whereas j is sampled at random, as defined by *FakeRead*. Thus, if $\mathscr{C}_P$ chooses index i, then the game $\mathscr{G}$ is in is exactly **Game 0**, and if $\mathscr{C}_P$ chooses index j, then the game $\mathscr{G}$ is in is exactly **Game 1**. Since the adversary makes α_0 calls to *RealRead*, we get:

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

Game 2: (IND-CCA with *Write*) This game is as above, except that *RealWrite* is modified to encrypt a dummy message instead of $seqNo|M$.

Let $\epsilon^{AE}_{IND-CCA}(\lambda_1, n)$ be the advantage of an adversary who breaks the IND-CCA assumption and performs n calls to *RealWrite* with security parameter λ_1.

Given an adversary $\mathscr{G}$ that distinguishes between **Game 1** and **Game 2** and makes n calls to *RealWrite*, we can construct an adversary $\mathscr{P}$ on the n-message IND-CCA security of the used encryption scheme. $\mathscr{P}$ behaves like a challenger to $\mathscr{G}$, following the description of the access sequence indistinguishability game, except for the case where a write request must be issued and the write queue is not empty (thus, a $RealWrite(\tau, seqNo, M)$ is executed). In this case, $\mathscr{P}$ generates a random message M^* with the same length as $seqNo|M$. Then, $\mathscr{P}$ forwards $seqNo|M$ and M^* to $\mathscr{C}_P$. Upon receiving the challenge ciphertext c in response, $\mathscr{P}$ forwards $\beta_1|\beta_2|c$ to $\mathscr{G}$, where β_1 and β_2 are the numbers of the buckets as specified by the `Write` request. Upon $\mathscr{G}$ ending the game, $\mathscr{P}$ responds to its own challenger $\mathscr{C}_P$ with 0 (meaning the challenge was an encryption of $seqNo|M$), if $\mathscr{G}$'s response was **Game 1**, and 1, if $\mathscr{G}$'s response was **Game 2** (meaning the challenge was an encryption of M^*). Note that if $\mathscr{C}_P$ chooses $seqNo|M$, then the game $\mathscr{G}$ is in is exactly **Game 1**, and if $\mathscr{C}_P$ chooses M^*, then the game $\mathscr{G}$ is in is exactly **Game 2**. Since the adversary makes α_1 calls to $RealWrite$, we can apply the IND-CCA definition and get:

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

Game 3: (PRF with *Write*) This game is as above, except we replace the PRF used to generate the bucket locations of *RealWrite*s with a truly random function, such that the client submits a *FakeWrite*. Let $\epsilon^{PRF}_{distinguish}(\lambda_2, n)$ bound the advantage of an adversary breaking the PRF assumption after n calls to the PRF with a security parameter λ_2. Technically, this game consists of a series of up to $2n$ games: two for each log handle used in a write request. In the following, we describe the game for replacing the PRF used in the generation of β_1.

Given an adversary $\mathscr{G}$ that distinguishes between **Game 2** and **Game 3** and makes n calls to *RealWrite*, we can construct an adversary $\mathscr{P}$ on the security of the used PRF that is called up to n times. $\mathscr{P}$ behaves like a challenger to $\mathscr{G}$, following the description of the access sequence indistinguishability game, except for the case where a write request must be issued and the write queue is not empty (thus, a $RealWrite(\tau, seqNo, M)$ is executed). In this case, $\mathscr{P}$ forwards $seqNo$ to $\mathscr{C}_P$. Upon receiving the challenge β in response, $\mathscr{P}$ uses it instead of β_1 in the generated `Write` request and forwards the request to $\mathscr{G}$ (note that exactly the same procedure can be done for β_2). Upon $\mathscr{G}$ ending the game, $\mathscr{P}$ responds to its own challenger $\mathscr{C}_P$ with *PRF*, if $\mathscr{G}$'s response was **Game 2**, and *Random function*, if $\mathscr{G}$'s response was **Game 3**. Note that if $\mathscr{C}_P$ chooses to use the PRF on the provided input, then the game $\mathscr{G}$ is in is exactly **Game 2**, and if $\mathscr{C}_P$ chooses to use a random function, then the game $\mathscr{G}$ is in is exactly **Game 3**. Since the adversary makes α_1 calls to $RealWrite$ and each of these calls contains two calls to the PRF, we conclude:

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

Game 4: (Hash functions in interest vectors) This game is as above, except we replace the h cryptographic hash functions used in the Bloom filter of the interest vector with queries to a random oracle. Technically, this game consists of a series of hybrid games, where we change each of the h hash function one by one. Let $\epsilon^{hash}(\lambda_3, n)$ bound the advantage of an adversary breaking the random oracle assumption in n calls with a security parameter, λ_3.

Given an adversary $\mathscr{G}$ that distinguishes between **Game 3** and **Game 4** and makes n calls to *RealWrite*, we can construct an adversary $\mathscr{P}$ on the random oracle assumption that is called n times. First, $\mathscr{P}$ chooses n log IDs at random. $\mathscr{P}$ behaves like a challenger to $\mathscr{G}$, following the description of the access sequence indistinguishability game, except for the case where a write request must be issued and the write queue is not empty (thus, a $RealWrite(\tau, seqNo, M)$ is executed). In this case, $\mathscr{P}$ selects one of the n log IDs chosen in the beginning (using the same log id if the log handle has been used in a *RealWrite* before) and forwards it to $\mathscr{C}_P$. The `Write` is then generated by $\mathscr{C}_P$ as specified in the previous game, except that the generation of the interest vector submitted with this request is changed. The interest vector is generated as specified in the previous game, except for the hash function in question. The position in the interest vector defined by this hash function is defined by the challenge received from $\mathscr{C}_P$. Upon receiving this challenge *pos*, $\mathscr{P}$ puts a 1 in the position *pos* of the interest vector of the generated `Write` request and forwards the request to $\mathscr{G}$. Upon $\mathscr{G}$ ending the game, $\mathscr{P}$ responds to its own challenger $\mathscr{C}_P$ with *Hash function*, if $\mathscr{G}$'s response was **Game 3**, and it responds with *Random oracle*, if $\mathscr{G}$'s response was **Game 4**. Note that if $\mathscr{C}_P$ chooses to use the hash function on the input provided, then the game $\mathscr{G}$ is in is exactly **Game 3**, and if $\mathscr{C}_P$ chooses to use a random oracle, then the game $\mathscr{G}$ is in is exactly **Game 4**. Since the adversary makes α_1 calls to *RealWrite*, and each of these calls contains one call to each of the h cryptographic hash functions, we conclude:

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

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 negligible.

Privacy: Collecting the probabilities from all games yields:

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

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

Proof of Security for Serialized PIR. We provide intuition for the security of the serialized PIR introduced in §6 by reduction to the security of the underlying PIR system discussed in §2, and the cryptographic assumptions of the cryptographic primitives used. First, because we use authenticated encryption, sending encryptions of the seed p_i and PIR request values q_i for each server i through the lead server is as secure as sending values p_i, q_i to server i directly through authenticated secure communication channels. Next, because of the security of the PRNG scheme, sending values p_i and computing $P_i = PRNG(p_i)$ on the server side is comparable to the security of the client generating random bit vectors P_i and sending those to the servers instead. Finally, because at this point vectors P_i are randomly generated and can be viewed as a one-time pad, computing responses R_i and sending values $R_i \oplus P_i$ to the lead server is as secure as computing and sending responses R_i to the client directly. At this point, the modified protocol corresponds exactly to the original version discussed in §2.

B PSEUDOCODE

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

(a) Log Handle

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

(b) Message Payload

Figure 13: Schema of the log handle and a message payload. The log handle is a shared secret between a trusted group of users, used to reconstitute a log from the servers. Each message payload in the log is encrypted with a shared encryption key.

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

1:  //Writes the data into one of two buckets
2:  function Write(buckets,data,interestVec)
3:      if isLeader() then
4:          seqNo ← seqNo + 1
5:          Append operation to globalLog
6:          Forward operation to follower servers with seqNo
7:      else
8:          Insert operation into globalLog at given seqNo
9:      end if
10:     Remove n-th oldest element from hashtable
11:     hashtable.insert(buckets[0], buckets[1], data)
12: end function
```

```
1:  //Performs a PIR-based read
2:  function Read(bucketVector)
3:      return bucketVector · hashtable
4:  end function
```

```
1:  //Returns the global interest vector
2:  function GetUpdates
3:      v ← BloomFilter()
4:      for all e ∈ last n elements of globalLog do
5:          v ← v ∪ e.interestVec
6:      end for
7:      return v
8:  end function
```

Figure 14: Pseudocode for server-side RPC handlers (NPI). The NPI was designed such that the 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 (§6). 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 (in §A.2) is equivalent to the serialized PIR algorithm described in §6.

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

1:  function Publish(log, message)
2:      Enqueue operation to writeQueue
3:  end function

1:  function Subscribe(log)
2:      Add log to logs
3:  end function

1:  function PeriodicWrite
2:      if writeQueue.isEmpty() then
3:          Send a random write request to the leader
4:      else
5:          log, data ← writeQueue.dequeue()
6:          seqNo ← logs[t.id]++
7:          bucket1 ← PRF(t.seed1, seqNo)
8:          bucket2 ← PRF(t.seed2, seqNo)
9:          data′ ← Enc_{log.key}(data)
10:         intVec ← BloomFilter()
11:         intVec.insert(log.id|seqNo)
12:         leader.Write([bucket1, bucket2], data′, intVec)
13:     end if
14: end function
```

```
1:  function PeriodicRead
2:      if readQueue.isEmpty() then
3:          Generate a random read request to each server
4:      else
5:          log, seqNo, seedChoice ← readQueue.dequeue()
6:          seed ← (seedChoice == 1)?log.seed1 : log.seed2
7:          data, query_I, query′ ← [0 . . . 0]          ▷ init to zero
8:          query′[PRF(seed, seqNo)] ← 1                 ▷ secret
9:          for each server in followers do
10:             query ← RandomBitString(numBuckets)
11:             data ← data ⨁ server.Read(query)
12:             query_I ← query_I ⨁ query
13:         end for
14:         query_I ← query_I ⨁ query′
15:         data ← data ⨁ leader.Read(query_I)
16:         if data contains (log, seqNo) then
17:             return data
18:         else if seedChoice == 1 then
19:             Enqueue a read for (log, seqNo, 2) to readQueue
20:         end if
21:     end if
22: end function
1:  function PeriodicUpdates
2:      globalIntVec ← leader.GetUpdates()
3:      for all log ∈ logs do
4:          seqNo ← logs[log.id]
5:          if globalIntVec.contains(log.id, seqNo) then
6:              Enqueue a read for (log, seqNo, 1) to readQueue
7:          end if
8:      end for
9:  end function
```

Figure 15: 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 log are written deterministically to random buckets. Subscribers use PIR to retrieve these messages. The code above uses serialized PIR (§6).