Unobservable Communication over Fully Untrusted Infrastructure

Sebastian Angel, ***The University of Texas at Austin and New York University;***
Srinath Setty, ***Microsoft Research***

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/angel

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Unobservable communication over fully untrusted infrastructure

Sebastian Angel
UT Austin and NYU

Srinath Setty
Microsoft Research

Abstract

Keeping communication private has become increasingly important in an era of mass surveillance and state-sponsored attacks. While hiding the contents of a conversation has well-known solutions, hiding the associated metadata (participants, duration, etc.) remains a challenge, especially if one cannot trust ISPs or proxy servers. This paper describes a communication system called Pung that provably hides all content and metadata while withstanding global adversaries. Pung is a key-value store where clients deposit and retrieve messages without anyone—including Pung's servers—learning of the existence of a conversation. Pung is based on private information retrieval, which we make more practical for our setting with new techniques. These include a private multi-retrieval scheme, an application of the power of two choices, and batch codes. These extensions allow Pung to handle $10^3\times$ more users than prior systems with a similar threat model.

1 Introduction

Can two or more users exchange messages over a public network without anyone else learning that they communicated? And can this be done in a practical manner without trusting any other entities (e.g., other users, ISPs, proxy servers)? This paper answers these questions affirmatively with a communication system that provides strong privacy guarantees, even against active, global adversaries.

While the questions we consider are decades old [32], there is a renewed interest motivated by an increase in service providers disclosing their users' information without consent [7, 13, 16, 83, 90, 109], as well as questionable mass surveillance practices [15, 27, 50, 62, 63] that defy existing privacy laws and long-held beliefs [44, 116, 128, 129, 133]. In response, companies have mobilized to deploy end-to-end encryption solutions to safeguard the privacy of users' communications [1–3, 5, 61]. While end-to-end encryption protects the content of the messages exchanged, it does not hide their existence nor other metadata (e.g., identity of participants, duration), which can be just as sensitive [38, 88, 117, 120].

Fortunately, the threat of metadata leakage has not been lost on academics and practitioners; there is a vast literature on preventing such disclosures [22, 25, 31–33, 40–43, 47, 51, 75, 79, 81, 82, 92, 93, 98, 113, 114, 119, 121, 130, 136]. While these works make great strides toward providing strong guarantees and supporting many users, we find that most require trusting one or more entities in the communication infrastructure (e.g., proxy servers, ISPs, large coalitions of users) to achieve their goals. In many contexts, such assumptions can be sensible. However, deployment considerations such as "where to find a trusted entity or an incorruptible consortium to run the system" are often left unspecified and are arguably hard to answer. Furthermore, there is enough precedent to think that private communication is a setting where trustworthiness can be subverted by financial and political interests [13, 39, 83, 90, 118]. There are proposals that do not require trusting the communication infrastructure [26, 33, 42, 60, 66, 131], but they have been primarily theoretical since the resulting systems support only dozens of concurrent users.

This tension between trust and performance drives our work. Our view is that private communication can be achieved with reasonable performance, even in the presence of strong adversaries. To substantiate this position we build Pung, a system that provably hides all metadata associated with users' conversations—even against adversaries who control all the communication infrastructure (ISPs, cloud providers, etc.) and arbitrary coalitions of users. We find that a 4-server deployment of Pung supports 135K messages/minute with 32K active users: $10^5\times$ more messages and $10^3\times$ more users than any prior system that withstands a similar adversary (§7.3). When we extend this comparison to systems under weaker threat models we find that Pung is promising but is not yet a replacement: Pung handles $85\times$ fewer users (§7.2).

To support tens of thousands of users at modest costs, Pung addresses two challenges. The first is architectural: devising a way for users to send and receive messages without a trusted proxy. Our proposal is simple, and consists of combining untrusted servers and powerful cryptography through a synthesis of known ideas (§3). The second, and more salient aspect of Pung is reducing the costs of its cryptographic machinery. Our contributions here include algorithms that amortize expensive operations when users send and receive multiple messages (§4).

In more detail, Pung is an untrusted key-value store that exposes private deposit and retrieval procedures to users. Pung's deposit procedure is based on the ability of communicating users to agree on a shared *label* (or "key" in the key-value store) under which to store a message (§3.1). Pung's retrieval procedure builds on a powerful—but expensive—cryptographic primitive: private information retrieval (PIR) [36]. PIR allows clients to fetch items from a server without revealing to the server which items were

fetched. While PIR has been used in other private communication systems [79, 92, 119], its interface is not a good fit for Pung: PIR requires clients to know the exact index of the items they wish to retrieve in a data structure stored at the server. In Pung, this data structure is continuously modified, and clients know only a label (§3.1).

To improve the performance of PIR, Pung targets applications where users retrieve multiple messages: email, group chats, bug reporting, and sensor data collection (§8). Pung then introduces a private multi-retrieval scheme that departs from most prior approaches (e.g., [18, 64, 67, 86]) in two ways. First, instead of modifying the design or implementation of PIR, Pung encodes the underlying data structures; these techniques are independent of the PIR scheme used (§4.1). Second, Pung leverages an inherent property of private communication systems: to resist traffic analysis they operate in rounds in which a bounded number of messages is sent and received by each user (§3.1). Users who wish to send or receives messages past this limit must wait several rounds to do so. Pung exploits this restriction with a multi-retrieval scheme that is probabilistically—rather than perfectly—complete: in a few cases clients can only retrieve a fraction of the items they wish to retrieve, but they can try again later. This results in a more efficient scheme than all prior PIR schemes that support multi-retrievals (§4.2).

To integrate PIR with Pung, we adapt an existing oblivious search technique [35] that allows clients to retrieve messages with labels (§3.3), and extend it to work on the encoded data structures that Pung uses for multi-retrieval (§4.4). Pung also introduces several other features. First, Pung supports group communication. Second, Pung provides a service that allows users to privately derive a shared secret to bootstrap their conversations, provided they know each other's public keys (§6). Last, messages in Pung are long-lived and can be retrieved at a later time by clients who participate infrequently (§8).

Nonetheless, our work has several limitations. While we reduce costs compared to prior approaches, these costs—especially network costs—remain high (§7.4). Furthermore, many of our techniques are only beneficial when clients retrieve multiple messages. Like all past private communication systems, Pung does not hide the fact that users are part of the system (it only hides if and with whom they are communicating); users are also required to participate even when they have nothing to send or retrieve. Pung does not provide liveness guarantees (censorship resistance) in the face of malicious servers or ISPs. This is fundamental since an ISP could simply refuse to route network packets. Lastly, Pung does not currently support an efficient dialing protocol to enable clients to "cold-call" one another (§5). Despite these limitations, we believe that Pung takes an important step toward enabling untrusted private communication.

2 Goals and threat model

In this section we discuss our goals and assumptions, and the general ecosystem that Pung targets.

2.1 Private communication over the Internet

Our objective is to develop a messaging system that allows two or more users to communicate over the Internet (or any other public network) while hiding the content of all messages exchanged in addition to the metadata of the exchange. The types of metadata that we wish to keep hidden from anyone—except from the users directly involved—include the start and end time of a conversation, the number of messages exchanged, the identity of the participants, etc. Some of this information is difficult to keep private since many existing services rely on it for their proper functioning. For instance, ISPs need to know destinations to route packets, email and chat service operators—who would in principle deploy and manage Pung—need to know the messages that make up a conversation, etc. Consequently, Pung must balance the requirements of existing services and infrastructure with the preservation of the following security goals:

Message integrity and privacy. The content of a message must be intelligible only to its intended recipient. Furthermore, no one should be able to tamper with a message while it is in transit without the recipient being able to detect alterations. Specifically, we target the strongest cryptographic properties that capture these goals, namely integrity of ciphertexts under chosen plaintext attacks (INT-CTXT) [21, 70], and indistinguishability under adaptive chosen ciphertext attacks (IND-CCA2) [99, 110].

Metadata privacy. An adversary must not be able to determine if (or when) a user sent or received a message. Furthermore, an adversary must not be able to link a message exchange with the users that participated in that exchange. Specifically, we target the privacy notion of *relationship unobservability* as defined by Pfitzmann and Hansen [105]. Informally, relationship unobservability states that an adversary does not learn useful information from observing (or actively interfering with) all network traffic, provided that the sender and the recipient are not compromised. In the case of such compromise, relationship unobservability offers little value: the sender could trivially disclose that it is sending a message and to whom; a receiver could similarly leak the sender's identity.

The above restriction is consistent with our setting of two-way communication. However, as we note in Section 9, relationship unobservability is not a panacea. For instance, it is not on its own sufficient to protect whistleblowers who wish to remain anonymous from everyone—including all recipients. We give a formal definition of metadata privacy and provide proofs of security for all of our techniques in our extended report [12, Appendix A].

2.2 Security assumptions

Pung achieves the security properties above under the following set of assumptions.

Cryptographic assumptions. Pung requires an authenticated encryption scheme (e.g., [21, 53]) to meet our goals of message integrity and privacy. Pung also relies on a computational private information retrieval (CPIR) scheme (e.g., [10, 58, 73]) and a pseudorandom function (e.g., [19, 20]) for ensuring metadata privacy (§3.1–§3.3).

Trust assumptions. Pung assumes that users who wish to communicate know their peer's public key (or can exchange a secret through and out-of-band channel). Pung provides privacy guarantees only to pairs (or groups) of users who communicate through Pung while following the prescribed protocol. However, these guarantees are not predicated on the behavior of any other user in the system, or the communication channel between users. In particular, Pung's guarantees hold even if *all* of the infrastructure that Pung uses (servers, ISPs, DNS, etc.) is compromised and operates arbitrarily.

Liveness assumption. Pung assumes that services used by clients to communicate with each other do not deny service. That is, we expect ISPs to carry traffic, DNS to provide name resolution, and servers to process requests. While this assumption is not needed for Pung to meet our security goals (§2.1), it is essential for Pung to be usable.

3 Design and architecture

Pung adopts a client-server architecture in which third-party servers mediate the exchange of messages between users. Figure 1 depicts this architecture. From the perspective of end users, a Pung cluster acts as a storage service. This parallels services like Gmail or Outlook that store messages on behalf of users.

Users exchange messages via a Pung client application that deposits the messages into *mailboxes* in the Pung cluster. These mailboxes are addressed by a *label* that is known to both the sender and the recipient. Recipients can access a message sent to them by retrieving the contents of a mailbox from the Pung cluster using an appropriate label. Pung's "mailbox" architecture is borrowed from prior systems [25, 40, 75, 79, 119, 130]. A key difference is which entities run the storage nodes, the kinds of processing that these nodes do, and the mechanisms for storing and retrieving messages. We discuss each of these components in the following sections, but we first highlight how this architecture fits within our target ecosystem.

Pung's mailbox architecture forces all messages sent and retrieved to go through entities like ISPs and the Pung cluster. These services rely on (or can easily infer) the types of metadata that we wish to hide, since they process all network traffic. Consequently, protecting metadata without harming the functioning of these services requires that the rate at which clients send and receive network packets be disentangled from the rate at which they send and retrieve messages in Pung. This requirement is key to preventing many types of traffic analysis attacks [68, 96, 112]. Unfortunately, it results in an unavoidable inefficiency: clients must send and retrieve messages at an independent (e.g., constant, Poisson) rate, *even* when the user is idle. This requires that clients queue excess requests and add cover traffic or *chaff* [115] (fake requests that are indistinguishable from real ones).

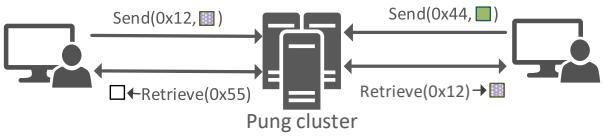

FIGURE 1—Client applications issue send and retrieve requests to the Pung cluster at a given rate, introducing fake requests whenever the user is idle (or issues fewer requests than the rate).

We now discuss how mailbox labels are derived, and how clients can use them to send and retrieve messages.

3.1 Mailbox labels and discretized rounds

The Pung protocol proceeds in discretized rounds or time epochs. Round duration is configurable and depends on the use case. The Pung cluster acts as a point of synchronization for clients and dictates when a new round starts. While this allows the Pung cluster to force clients out of sync, doing so results in a denial of service but does not violate our goals (§2.1). During each round, client applications issue exactly one send and one retrieve. This ensures that clients issue requests at a constant rate (§3). In Section 4 we relax this model and let clients issue multiple send and retrieve requests per round, enabling several applications (§8) and achieving lower (amortized) costs (§7.3). Finally, Section 5 discusses how clients can manage existing connections, and how they can agree on a round on which to start a new conversation.

Deriving mailbox labels. The Pung cluster is effectively a key-value store that treats mailbox labels as keys, and (encrypted) messages as values. This means that users' communication depends on their ability to agree on a label under which to store and retrieve messages. This label should be unique (to avoid multiple pairs of users overwriting each other's messages), and it must also be independent of the users communicating (otherwise an adversary could link a label to a conversation). Pung achieves both of these properties through a combination of shared secrets and a pseudorandom function (PRF).

Recall from Section 2.2 that we assume that users who wish to communicate have access to each other's public key (e.g., RSA key), or have exchanged a secret through an out-of-band channel. In Section 6 we present a directory service that allows users to derive a shared secret

directly from public keys. Consequently, the rest of this section assumes that users have a shared secret which acts as a master key. This master key is used to derive two additional keys, k_L and k_E, with a key derivation scheme [76]. The derived keys are used for mailbox label generation and message encryption, respectively. We also assume that users have a unique identifier, *uid*, within each pair of communicating users. For example, if Alice and Bob wish to communicate with each other, Alice could be "0" and Bob could be "1". This information need not be private, so users could choose any identification scheme including using their names or public keys.

Each user can derive the corresponding labels for the current round r, $label_S(r)$ and $label_R(r)$, by invoking the pseudorandom function (PRF) keyed with k_L:

$$label_S(r) = \text{PRF}_{k_L}(r \,||\, uid_{peer})$$
$$label_R(r) = \text{PRF}_{k_L}(r \,||\, uid_{own})$$

where r is a fixed-width integer and $||$ is the concatenation operator when r and *uid* are treated as binary strings. Note that labels need not be symmetric: a user can send a message to Alice and retrieve one from Bob in the same round. In such cases, the labels would be generated using different keys and *uid*s. If a user is idle and has nothing to send or retrieve, it generates random mailbox labels.

3.2 Sending messages in Pung

Sending a message in Pung consists of deriving the recipient's mailbox label ($label_S$), and encrypting the message with an authenticated encryption scheme (§2.2) using key k_E. The client then sends the resulting ciphertext, $c = AE(k_E, m)$, along with the mailbox label, to the Pung cluster as a $(label_S, c)$-tuple. Idle users send a tuple that consists of a random label and an encryption of a random message instead. We assume that all messages are the same size or that padding is applied.

3.3 Retrieving messages from the Pung cluster

Observe that if the Pung cluster were to broadcast to all users the $(label, c)$-tuples received during a round, users could iterate through the list locally and find the tuple with the label that is of interest to them (or determine that it is not present). Intuitively, this operation would not leak any information about which label (if any) was of interest to a retriever, and would not allow the adversary to determine with whom a user is communicating (or if the user is idle). Of course, broadcasting all tuples would incur prohibitive network costs. Fortunately, retrieving an item from an untrusted server without revealing *which* item was retrieved is the problem addressed by private information retrieval (PIR) [36]. PIR protocols trade off computation at the server to achieve lower network costs than the above broadcast scheme. We summarize PIR next, since it is the basis of message retrieval in Pung.

Private information retrieval (PIR). We focus on *computational* PIR (CPIR) schemes [10, 28, 30, 57, 58, 73, 77, 135] that hide users' access patterns under cryptographic hardness assumptions.[1] At a high level, a CPIR scheme operates over a collection *DB* of n items held by a server, and consists of three procedures: QUERY, ANSWER, DECODE. The $\text{QUERY}(idx, n)$ procedure is run by the client; it outputs a query q that encodes the index, *idx*, in *DB* of the desired element. The $\text{ANSWER}(q, DB)$ procedure is run by the server; it returns an encrypted response a that contains the element in *DB* at the index encoded in q. This step requires the server to perform cryptographic operations over *all* elements in *DB*. The $\text{DECODE}(a)$ procedure is run by the client; it decrypts a to recover the desired element in *DB*. Below we describe a simple CPIR scheme based on an additively homomorphic cryptosystem.[2]

The client first generates a *query vector* q of length n by calling the $\text{QUERY}(idx, n)$ procedure. Every entry in q is a different encryption of 0, except for the entry at position *idx* which is an encryption of 1. The client sends this query to the server, who executes $\text{ANSWER}(q, DB)$ to produce a ciphertext c that encrypts the element in *DB* at position *idx*. To do this, the server creates a vector x by interpreting every entry $e_i \in DB$ as an integer, and computing the product of e_i and the ciphertext q_i. This can be accomplished through repeated additions of q_i by leveraging the additive homomorphic property of the cryptosystem: $x_i = \prod_1^{e_i} q_i$. The server then adds up every entry in x to obtain a. This procedure works because the vector x consists of $n-1$ ciphertexts that encode 0, and one ciphertext that encodes e_{idx}. Adding all of them results in an encryption of e_{idx}, without the server learning which index was requested. Lastly, the client runs $\text{DECODE}(a)$ to decrypt a and get the desired element.

All of the CPIR schemes to which we refer (and on which we rely) are more efficient than the above straw man, but they have a similar flavor. Crucially, they enjoy communication costs sublinear in n (i.e., they are cheaper than transferring the entire collection). Furthermore, some CPIR schemes (e.g., [10]) have low enough computational costs that their processing latency is actually lower than transferring the entire collection over today's networks (this was believed to be an unlikely scenario [125]).

Retrieving messages. Since PIR allows clients to privately retrieve an item from the server at some index, one possibility is to use labels as indices: clients can retrieve a message from $label_R(r)$ with $q = \text{QUERY}(label_R(r))$. However, the size of the collection would need to match

[1] IT-PIR schemes [36, 48, 59] are an efficient alternative to CPIR but rely on multiple servers, at least one of which must be correct. This conflicts with our goals and threat model (§2).

[2] An additively homomorphic cryptosystem supports an operation "$\cdot$" that can be used on ciphertexts to produce a new ciphertext encoding the sum of their plaintexts. That is, $Enc(x) \cdot Enc(y) = Enc(x + y)$.

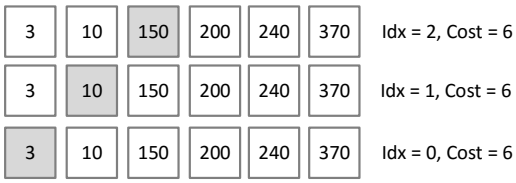

FIGURE 2—A client wishing to retrieve an item with label "3" from a server holding a sorted list of 6 items would need to perform three rounds of probing. During each probe, the client guesses an index, uses PIR to retrieve the $(label, c)$-tuple at that index, and refines the guess accordingly. "Cost" indicates the number of items processed by the server in each probe.

the range of the labels (§3.1), which is 256 bits in our implementation (§6). This would require Pung servers to materialize and operate over a collection of 2^{256} items!

Instead, we can arrange for Pung servers to insert all $(label, c)$-tuples sent by clients in some search data structure (e.g., sorted list, search tree) and present them as a collection *DB* of size n (where n is the total number of nodes in the data structure). This enables clients and Pung servers to perform PIR directly on *DB*, but there is a problem: clients know from which label they wish to retrieve, but they do not know the mapping between labels and the index of the desired tuple in the data structure representing *DB*, or if the tuple even exists. This can easily be addressed by having clients obtain this label-to-index mapping from the Pung cluster. However, when the collection is large (n>100K), clients can use a search scheme to reduce network costs. We discuss this below.

The key idea is that clients can find their desired element in *DB* via an "oblivious" search. Figure 2 depicts an example of this search when *DB* is stored as a sorted list. In this case, the client performs $\log(n)$ probes to locate its desired element (or determine that it is not present). Even if the client gets lucky and finds its element early, it must continue until the end to preserve privacy; the remaining probes can just use any indices. Since each probe is a PIR query to the entire collection *DB*, the server must process n elements each time; the time complexity of this search is therefore $\Theta(n\log(n))$. However, this scheme has a lot of redundancy: the server processes each item $\log(n)$ times. Chor et al. [35] show that one can eliminate this "double counting" overhead by using data structures that can be (logically) split into independent chunks while retaining the search capability. We elaborate on this idea below in the context of the specific construction that Pung uses.

BST retrieval. We choose to use a complete[3] binary search tree (BST) as our underlying data structure for several reasons. First, a complete BST is balanced, enabling search in $\mathcal{O}(\log(n))$ probes. Second, for any dataset there is a unique complete BST, so the Pung cluster need not communicate the structure to clients (aside from n). Last, since every level of a complete BST is full (except for possibly the last) and every node contains an actual data item, there is no need for padding or auxiliary elements; it can be represented as a contiguous array without overhead.

[3] A k-ary tree is complete if all of its levels (except possibly the last) are full, and the last level is filled from left to right.

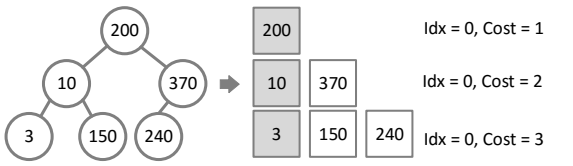

FIGURE 3—The Pung cluster can store $(label, c)$-tuples in a complete BST, allowing clients to treat each level as an independent collection. Clients can issue a PIR query for the top level, and can recursively derive the index of lower levels using BST semantics. This figure depicts the search for label "3".

```
1:  function BST-RETRIEVAL(L*, n)
2:      h ← ⌊log2(n)⌋                          // last level of the BST
3:      c* ← ⊥          // target ciphertext (⊥ means not yet found)
4:      idx ← 0                                // index of the current level
5:      len ← 1                                // length of the current level
6:      len^h ← n − (2^h − 1)                  // length of the last level
7:
8:      for i = 0 to h do
9:    // use PIR to get element at position idx from collection at level i
10:         q ← QUERY(idx, len)
11:         a ← send i and q to server and get answer
12:         (L, c) ← DECODE(a)
13:
14:         if c* == ⊥ then
15:             if L* < L then                 // access left child next
16:                 idx ← 2 · idx
17:             else if L* > L then            // access right child next
18:                 idx ← 2 · idx + 1
19:             else                  // L* == L, found target ciphertext
20:                 c* ← c
21:
22:         len ← 2^(i+1) or len^h             // length of the next level
23:
24:         if idx ≥ len or c* ≠ ⊥ then
25:             idx ← random index between 0 and len − 1
26:     return c*
```

FIGURE 4—Client procedure for retrieving an encrypted message $c^\star$ from a mailbox with label $L^\star$. The server holds a collection of n $(label, c)$-tuples in a complete binary search tree.

Based on this, we set up the Pung cluster to store the collection of $(label, c)$-tuples in a complete BST, and have clients treat all the nodes at the same depth in the tree (i.e., on the same level) as a (logically) separate collection. As depicted in Figure 3, clients can then process each of the $\log(n)$ collections sequentially from top to bottom, deriving the index of the next level from the semantics of the BST. The pseudocode for this procedure is listed in Figure 4. Since each collection (and therefore each element) is accessed exactly once, there is no overhead due to double counting. Indeed, the time complexity of this BST-based retrieval scheme is $\Theta(n)$, which is the same as if the clients had known the index in the first place.

Compared to performing PIR over a known index, clients do incur a $\log(n)\times$ higher network cost due to retrieving a tuple at every level. As an optimization, clients could fetch (non-privately) all of the tuples of the first few levels, saving both bandwidth and CPU. This is because CPIR queries and answers are typically much larger than the elements in the collection; when the collection is small, it is more efficient to download all elements (i.e., naive PIR) than to use a CPIR scheme (§7.4).

The above sending and retrieval procedures are sufficient to build a version of Pung that meets all of our security goals (§2.1): it enables users to communicate with each other privately, hiding the content and preserving the integrity of messages, without leaking any metadata associated with a conversation. Furthermore, none of the security guarantees depend on the correctness of the Pung cluster. For instance, if the Pung cluster modifies the ciphertext associated with any tuple, clients can detect this due to the integrity guarantees of the authenticated encryption scheme. If the server drops tuples or stores them in a data structure that is not a complete BST, clients will be unable to find the tuple of interest to them (a denial of service), but the integrity of the content and the privacy of the communication is preserved. The drawback with the above scheme is its costs: the server has to process the entire collection for each client request. Additionally, for applications where clients wish to retrieve more than one message in a round (§8), costs scale linearly with the number of messages retrieved. The next section describes ways to significantly amortize costs for regimes in which clients retrieve multiple messages simultaneously.

4 Reducing costs via multi-retrievals

This section describes how to reduce the CPU costs of the Pung cluster when clients retrieve multiple messages.

4.1 Prior approaches to multi-retrieval

One approach to retrieving k items from the server is to run the protocol in Section 3.3 k times, but this results in costs that are linear in k. An alternative is to create new PIR schemes that support a batch of k retrievals with sublinear costs. Groth et al. [64] achieve significant improvements with this approach, but their focus is reducing network costs—the resulting CPU overheads are prohibitive in our context. Another approach is to modify the implementation, rather than the design, of existing PIR schemes. In particular, as we discuss in Section 3.3, the query of many PIR schemes is a vector of encrypted entries. The server can aggregate the queries submitted by (potentially different) users into batches of size k, and construct a matrix. This enables the server to leverage fast matrix multiplication algorithms (e.g., Strassen's algorithm [126]) to evaluate PIR's ANSWER procedure. Several works have shown that this yields modest benefits [18, 67, 86].

In Pung, we take a different approach—inspired by *batch codes* [69]—from the schemes above: instead of modifying the design or the implementation of a particular PIR protocol, we focus solely on changing the representation of the underlying data.[4] We discuss batch codes in detail in Section 4.4 since we use them as a final refinement to our scheme. At a high level, they enable the server to encode a collection into smaller subcollections, in such a way that clients can retrieve any k items by querying each subcollection at most once. Below we highlight several reasons for designing a new mechanism rather than directly applying batch codes.

Challenges and opportunities. First, many batch codes suffer from a major drawback: the number of elements that a client downloads increases rapidly with k. This means that for small k (3 or 4), network costs are within a small factor of retrieving items one by one; but they quickly rise to untenable levels with larger k. Second, batch codes' *perfect completeness* guarantee (i.e., that clients can retrieve any k items) is too conservative for our setting. In particular, Pung does not require that clients can *always* retrieve all k messages during a given round: since messages in Pung are long-lived (§6), clients can retry the next round. This behavior is actually inevitable in systems resistant to traffic analysis, such as Pung: recall that clients send and retrieve messages at some rate; any client who receives messages in excess of this rate must wait at least two rounds. Below we describe an alternative that works well for larger k, but is probabilistic. That is, a client can sometimes only retrieve a subset of the k messages that it wished to retrieve in a given round.

4.2 Probabilistic private multi-retrieval

We now introduce a new probabilistic *multi-retrieval* scheme. A multi-retrieval scheme allows a server to efficiently process multiple retrievals from the *same* client by amortizing costs. Our proposal is more efficient than prior approaches, especially for larger values of k (> 4).

At the core of multi-retrieval is the observation that as long as every item in the server's collection is processed at least once, the underlying PIR protocol will ensure that the server does not learn which tuples were retrieved. As we discuss in Section 3.3, one can take a collection and structure it as a tree, allowing each level to be treated independently. This results in clients retrieving $\log(n)$ tuples, while the server processes each element just once; incurring the same CPU costs as a single retrieval. The reason that BST-RETRIEVAL (Fig. 4) is not technically a multi-retrieval scheme is that clients have no control over which tuples are fetched (they are forced to follow BST semantics), and consequently the procedure can only

[4] Using PIR as a black box means that other optimizations (e.g., fast matrix multiplication) benefit Pung as well.

output a single message. We now show a way to divide the collection into smaller subcollections while still allowing clients some control over which items to fetch.

Server setup. The server initially performs a static partitioning of the label space (e.g., 2^{256}) into B buckets (we set B to the maximum number of messages that users retrieve in a round, i.e. k). Each bucket holds all $(label, c)$-tuples whose labels fall into its partition. At the end of the send phase, the server takes all the $(label, c)$-tuples sent by clients and distributes them across the B buckets based on their label. Small buckets store tuples in an arbitrary order, while larger buckets store tuples in an array that represents a complete BST (§3.3). The latter enables clients to use BST-RETRIEVAL (Fig. 4), which saves network resources. Finally, the server sends clients the number of items in each bucket, and awaits retrieval requests.

Client lookup. A client can retrieve multiple messages simultaneously by treating each bucket as an independent collection and retrieving one $(label, c)$-tuple from each bucket. This is done by calling an appropriate retrieval procedure on each bucket with a label that falls within the bucket's range and the size of the bucket: for BST-encoded buckets, the client uses BST-RETRIEVAL; for other buckets, the client requests the label-to-index mapping, and retrieves a $(label, c)$-tuple by directly sending the output of PIR's QUERY procedure to the server. If a client does not wish to retrieve a tuple from a particular bucket, it performs the retrieval using a random label. Note that since BST-RETRIEVAL (or PIR's QUERY) is executed on each bucket independently, the server's CPU cost is still the same as if the client had requested a single tuple from the entire collection (as was the case in §3.3).

In the best case, since there are as many buckets as user queries ($B = k$), clients can retrieve all of their desired messages at once. However, this scenario presupposes that all tuples that the client wishes to retrieve have labels that fall in different buckets. But what if a client wished to retrieve ρ tuples ($1 < \rho \leq k$) from the same bucket?

Unfortunately this cannot be done privately as it would require the client to interact with the same bucket ρ times, leaking information about the requested labels. Instead, the entire protocol must be rerun ρ times, allowing the user to retrieve one message from the contested bucket on each run. There is one caveat: the number of times that the protocol is rerun during a round must not depend on the user's choice of labels; this too would leak information. Instead, the number of reruns must be set a priori.

But how common is it for clients to want to retrieve multiple tuples from the same bucket? This is a standard balls-and-bins scenario, since the client's labels are generated from a pseudorandom function, and the buckets' range is statically and independently partitioned. We can thus bound the number of tuples that fall in any bucket by $\rho \leq \frac{3\ln(k)}{\ln(\ln(k))}$ [95, Lemma 5.1]; this bound fails to hold with probability $\leq \frac{1}{k}$. Unfortunately, this is a fairly large number (9–11, for $k \leq 512$), especially since we require rerunning the entire protocol ρ times to guarantee that clients can retrieve k messages with high probability.

Below we describe how Pung reduces the bound on ρ exponentially by reaping the load balancing benefits of giving clients multiple choices to retrieve tuples [94].

4.3 Fewer reruns with the power of two choices

Azar et al. [14] show that in a k balls and k bins scenario, if each ball maps to d random bins ($d > 1$), and balls are placed in the bin least full, the highest load in any bin is bounded by $\frac{ln(ln(k))}{ln(d)} + \Theta(1)$ with high probability.

We observe that if clients had multiple buckets from which to retrieve a message, we could apply this result to decrease the bound on ρ, and consequently the number of reruns that clients must perform during multi-retrieval (§4.2). However, this kind of load balancing is typically applied from the *producer's* perspective (e.g., choosing which server to issue a request, or on which queue to place a packet); in our case, we are interested in enabling the *consumer* (i.e., the recipient of a message).

This raises the following question: how can we enable a client to be able to retrieve a message under two labels? We propose a seemingly bad idea: have senders derive *two* labels for each message, and have the server store messages under both labels. This of course doubles the already large number of messages in the system (n). Considering that all PIR costs scale linearly with n, and the BST retrieval scheme (§3.3) adds a multiplicative $\log(n)$ factor to network costs, this is a cause for concern. However, the exponential decrease in the number of reruns that clients will have to perform (i.e., ρ), far outweighs the costs associated with doubling all messages. Ultimately, this simple approach results in significant savings.

We implement the above scheme by extending Pung's send and retrieve procedures (§3.2). Recall that clients derive two keys from their shared secret, and use one of them (with a PRF) to generate a label under which to store a message. Under the modified protocol, clients derive a third key that they use in combination with a second PRF to generate the extra label.[5] Clients can then send (L_1, L_2, c) to the server, which then stores c under two different $(label, c)$-tuples. During retrieval, clients generate both labels for each message they wish to retrieve (§3.3) and follow the lookup scheme (§4.2) using the label that leads to fewer bucket collisions. Note that collisions are defined with respect to a client's other labels. They are independent of the actions of other clients or the server; they are therefore a notion local to each client.

[5] Clients need to ensure that both labels do not map to the same bucket. This can be done by using a counter as a nonce to the PRF, incrementing it until both labels map to different buckets.

4.4 Probabilistic multi-retrieval with batch codes

The above bucket-based scheme makes progress toward lowering CPU and network costs, but still requires the protocol to be rerun ρ times. In this section we further refine the scheme by composing it with batch codes, discussed next, to achieve a hybrid scheme that has lower CPU costs than either mechanism, fewer round trips than the bucket-based scheme, and lower network costs than applying existing batch codes in isolation.

Batch codes. A (n, N, k, m)-batch code [69] takes as input a collection of n items and the number of desired retrievals k ($k > 1$), and outputs N items ($n < N < n \cdot k$) distributed across m subcollections ($m > 2$) that have a useful load-balancing property: any k items from the original collection can be retrieved by querying each of the subcollections at most once. In our context, this means that a Pung server that encodes n $(label, c)$-tuples with a batch code can process k simultaneous queries from the same client, while only paying the processing cost required to answer one query to a collection of N tuples.

We now give an example of a $(n, \frac{3}{2}n, 2, 3)$-batch code scheme that supports $k = 2$ retrievals. A collection DB of n items, is split into 3 subcollections db_1, db_2, db_3, such that db_1 has the first half of the items, db_2 has the second half of the items, and db_3 has $db_1 \oplus db_2$ (where $\oplus$ is the element-wise XOR operator). A single PIR query to each subcollection is thus sufficient to privately retrieve any two items from DB (we provide details later in this section). Furthermore, the CPU cost of answering all three queries (one for each subcollection) is the same as that of processing one PIR query over a collection of $N = \frac{3}{2}n$ items. Therefore, this scheme is 25% cheaper than running PIR twice on DB to retrieve 2 items (since that would require processing $2n$ items).

Subcube batch codes [69] are a generalization of this scheme and allow clients to retrieve any k items at once by recursively performing the above encoding (e.g., to support $k = 4$, one encodes each of db_1, db_2, db_3 to obtain a total of $m = 9$ subcollections). Consequently, large values of k significantly amortize the CPU cost of retrieving k items. A disadvantage is that clients always have to retrieve an element from each of the m subcollections, where $m = 3^{log(k)}$ in the above scheme. This is acceptable for small k, but for large k the network overheads are enormous: for $k = 128$, clients retrieve 17× more elements than running 128 instances of the scheme in Section 3.3.[6]

On the other hand, our probabilistic bucket-based scheme allows clients to retrieve k messages at once with lower CPU and network overhead, but requires ρ reruns of the protocol (ρ is roughly 3–4 with our refinement in §4.3). The rationale behind rerunning the protocol is that clients might need to retrieve up to ρ items from the same bucket. Observe that retrieving a few items (e.g., $k \approx$ 2–4) is a strength of subcube batch codes. It therefore makes sense to hybridize the two techniques. However, subcube batch codes are not compatible with BST-based retrieval (which reduces network costs for large buckets as discussed in §3.3). We address this with the following technique, which might be of independent interest.

BST retrieval with subcube batch codes. We now adapt BST-RETRIEVAL (Fig. 4) to work on encoded collections. We focus on the $(n, \frac{3}{2}n, 2, 3)$-subcube batch code described earlier but our approach generalizes.

Server setup. The server starts with a collection of n $(label, c)$-tuples, which it sorts based on labels. Analogous to the batch code scheme described earlier, the server splits the collection into two halves, and stores them as two complete BSTs, b_1 and b_2. Finally, the server creates a third binary tree, b_3, from b_1 and b_2 as follows: for every level i and index j, $b_3(i,j) = b_1(i,j) \oplus b_2(i,j)$. The server then indicates to clients the collection size (n) and the lowest label in b_2, L_{mid}; tuples with labels lower than L_{mid}, if they exist, would be found in b_1.

Client lookup. A client wishing to retrieve two tuples labeled L_1 and L_2 can do so as follows. Assume without loss of generality that $L_1 < L_2$. There are two cases:

- If $L_1 < L_{mid}$ and $L_2 \geq L_{mid}$: the client calls BST-RETRIEVAL(L, $\frac{n}{2}$) on each tree independently, passing L_1 for b_1, L_2 for b_2, and a random label for b_3.
- If $L_1 < L_{mid}$ and $L_2 < L_{mid}$, the client calls BST-RETRIEVAL(L_1, $\frac{n}{2}$) on b_1, and performs a *joint tree traversal* on b_2 and b_3 to retrieve L_2 (the case where both $L_1 \geq L_{mid}$ and $L_2 \geq L_{mid}$ is symmetric and simply requires exchanging the role of b_1 and b_2).

Joint tree traversal. Since b_3 is not a BST (i.e., the order of its elements does not respect BST semantics), it cannot be used directly for search. However, it can be jointly traversed with the help of another tree. We describe this for the case where $L_1 < L_{mid}$ and $L_2 < L_{mid}$. A client starts by retrieving the tuples at level 0 and index 0 for both b_2 and b_3 in parallel. This is equivalent to lines 10–12 in Figure 4 (during the first iteration of the loop). The result of the two separate calls (one for each tree) to the DECODE procedure in line 12 is the pair of tuples t_2 and t_3. While the label of t_3 is unintelligible (since it is encoded) and the label of t_2 is irrelevant to the client's search, they can be combined to compute $(L, c) = t_1 = t_2 \oplus t_3$, which is the corresponding tuple in b_1. This yields a way to jointly traverse the trees: the client can compare L_2 to L and choose whether to go left or right on both b_2 and b_3 for the next level. If $L_2 = L$, the client can save c (as this is the desired ciphertext), and continue with random indices for the remaining levels. The above steps are analogous to lines 14–25 in Figure 4 when one replaces $L^\star$ with L_2.

[6] Other batch codes exist [69, 104, 111, 123], but their concrete costs are significantly higher than those of subcube batch codes in all our cases.

A hybrid scheme. As before, the server partitions the label space into B buckets. For each bucket b, the server encodes all the corresponding tuples with a (n_b, N_b, ρ, m)-subcube batch code. Here, n_b is the initial number of tuples in b, ρ is the number of reruns required after deriving two labels per tuple (§4.3), N_b is the total number of tuples in b after encoding, and m is the number of subcollections per bucket ($m = 3^{log(\rho)}$). If n_b is large enough, the server uses the BST-aware batch code presented above so clients can benefit from the lower network cost of BST-based retrieval. The upshot is that combining batch codes with probabilistic multi-retrieval lets clients retrieve up to ρ tuples from each bucket, without rerunning the protocol.

5 Operational challenges

A key challenge in any communication system is managing user connections. In particular, how do clients determine when and for how long to communicate? In Pung, the answer depends on the type of pre-existing relationship that users have: *symmetric*, where users already know each other and have already derived a shared secret (§3.1), and *asymmetric*, where one user wishes to "cold call" another for the first time. We now describe both cases.

Managing symmetric connections. Client applications of users who already know each other can exchange *control messages* through Pung. Control messages have a special structure that client applications can recognize and automatically act upon, so they are transparent to actual users. Control messages are sent over Pung like any other message—so they too are private—and include statements like "`END`" to indicate that a conversation is over, or "`START [round]`" to indicate the round when a conversation should start. These messages are sent periodically (e.g., every 20 rounds), but can also be sent during an active communication in response to events (e.g., `END` is sent when the application is placed in the background or when the user stops typing for a few minutes).

The frequency of control messages is initially configured the first time that two users communicate with each other, but it can be adjusted dynamically with the "`FREQ [rounds]`" control statement. Higher frequency leads to smoother operation (e.g., client applications can agree on a round to start a conversation faster), but like any other message, they count toward the send and retrieve rate limit chosen by the user (§3.1). Pung's multi-retrieval optimizations (§4) make sending and receiving control messages more efficient, and enable clients to fetch control messages from several known peers at once.

Initiating asymmetric connections. The exchange of control messages described above presupposes an established relationship between clients. But how does Pung bootstrap this interaction in the first place? One option is for clients to use control messages to introduce their peers to others. A more realistic alternative is for clients to use a *dialing* protocol, as proposed by Vuvuzela [130] and Alpenhorn [80]. In a dialing protocol, clients send *invitations* (messages stating the desire of a user to start a conversation, and information about a round on which to do so) to mailboxes with labels derived from users' email addresses [80] or public keys [130]. Clients can then periodically check their corresponding mailboxes for invitations, without leaking metadata in the process.

Unfortunately, Pung does not currently support an efficient dialing protocol. We attempted to adapt Vuvuzela's dialing scheme, but due to Pung's threat model and architecture, we found that it degenerates into each client having to download the invitations sent by all users. The precise issue is that Pung does not provide sender anonymity [105]. Incidentally, all existing systems that provide sender anonymity without trusted infrastructure are fully peer-to-peer and broadcast messages to everyone [33, 42, 60, 66, 131]. This makes dialing gratuitous since all users already know each other (i.e., relationships are symmetric), and they actively communicate with everyone in every round. Designing an efficient dialing scheme under our setting (§2)—or proving that it cannot exist—remains an open question.

6 Implementation

We implement Pung in 5,800 lines of Rust and C++ bindings. We express the server-side computation of Pung in Naiad's timely dataflow model [97], and use the Timely Dataflow library [89] written in Rust, to create, run, and coordinate dataflow workers. Each worker processes send and retrieve requests issued by clients, encodes the tuple collections, and invokes the PIR procedures exposed by XPIR [11]. Finally, we derive keys from secrets with HKDF [76], generate labels with HMAC-SHA256, and encrypt messages with ChaCha20-Poly1305. All of these operations are supported by the Rust-Crypto library [8].

Additional features. Our prototype supports:

• *Long-lived messages.* The Pung cluster maintains a sliding window of messages, regardless of the number of rounds over which they were sent. This allows users to retrieve messages sent to them during past rounds. This requires dataflow workers to mix new and existing messages, garbage collect the messages that outlive the sliding window, and reconstruct buckets and BSTs.

• *Group communication.* Pung provides privacy to groups if all users in the group follow the protocol. Suppose a group G has derived a shared key k_L, then: (1) user $i \in G$ can send its message to G under label $\text{PRF}_{k_L}(r \mid\mid uid_i)$ during round r; (2) users in G can simultaneously retrieve all messages sent in round r using a multi-retrieval query with labels $\text{PRF}_{k_L}(r \mid\mid uid_j)$ for all $j \in G$.

• *Directory service.* If users know each others' public

keys pk_i (e.g., RSA keys), they can derive a shared secret through a standard Diffie-Hellman key exchange [49] via Pung. User i can send the tuple $(\text{PRF}_0(pk_i), \{pub_i, \sigma_i\})$ to the server, where pub_i corresponds to i's public Diffie-Hellman parameters $(g, p, g^a \bmod p)$, and σ_i is a signature of pub_i under i's private key. Notice that the tuple's label depends only on pk_i; anyone with access to pk_i can derive the label and retrieve the tuple. Clients can retrieve each other's public components (pub_i), verify their authenticity, and derive the shared secret independently. Clients send these tuples to Pung servers when they first register, or via a special message that flags them so they are not garbage collected by dataflow workers. Pung stores these tuples in the same collection as other messages, so their access is kept private. If the tuples are larger than regular messages, they are split into chunks; clients can retrieve these chunks over several rounds or with multi-retrieval.

Compressing explicit label mappings. Recall that for large collections BST retrieval incurs less network costs than explicitly downloading the label-to-index mappings and performing PIR with a known index (§3.3). We now describe how to delay the *breakeven point* (i.e., the collection size at which BST retrieval is better than explicitly downloading labels) by using a *Bloom filter* [24]. A Bloom filter is a probabilistic data structure that encodes a compressed representation of a set, and is widely used to reduce network costs in many settings, including private communication [80, 108] (although our use case is different). It exposes a *check* procedure that allows anyone to check whether some element is in the set (false positives are possible and occur with small probability).

In our implementation, the Pung server adds to a Bloom filter the element $index||label$ for each tuple in the collection, and sends it to clients. Clients can then find the index of their desired label $L^\star$ by testing for set membership locally while varying the index until a match is found: $check(0||L^\star), \ldots, check(n-1||L^\star)$. While standard Bloom filters require computing a large number of hash functions for each add and check operation, there exist constructions that require only two [74]. Thus, with little computation, clients can locally derive their desired index while saving network resources. For larger collections, retrieval via BST (Fig. 4) remains more efficient.

7 Experimental evaluation

Our evaluation answers four main questions. First, what is the cost of the cryptographic primitives used in Pung (§7.1)? Second, what is the concrete performance of Pung, and how does it compare to prior systems (§7.2)? Third, what are the benefits of multi-retrieval (§7.3)? Last, what are the costs that Pung imposes on clients (§7.4)?

Setup and metrics. We deploy Pung's server logic on timely dataflow workers running on Microsoft Azure H16 instances (16-core Intel Xeon E5-2667 with 112 GB RAM) with Ubuntu 16.04. Our performance metrics are throughput (in messages/minute) and end-to-end latency (in seconds). Note that all entities run on the same data center, so our results do not capture the effects of wide area networking. In all cases we report the mean over 10 trials; standard deviations are less than 10% of the means.

We run clients and dataflow workers in a closed loop and let round duration be as low as possible: a new round starts as soon as all current requests are fulfilled. To keep the number of messages constant across rounds, we configure Pung's garbage collection window to be the number of messages sent in one round (§6).

Baselines. We compare Pung to two prior systems: Dissent [42] and Vuvuzela [130]. They represent the state-of-the-art in private communication under the anytrust[7] (Vuvuzela) and no-trust (Dissent) models. We want to emphasize that our comparison to Dissent is not apples-to-apples: Dissent achieves an additional privacy property—sender anonymity (§2, §9)—that Pung does not provide. However, we are not aware of a system with the same guarantees as Pung under our threat model.

7.1 Microbenchmarks

To understand the costs of Pung we start with a series of microbenchmarks. The network and CPU costs of many of Pung's operations depend on the size of the collection (n = # of tuples) held by the Pung cluster and the size of each $(label, c)$-tuple. We report the results for several collection sizes, and tuple sizes (288 bytes, 1 KB). We choose these tuple sizes to match our baselines: Vuvuzela clients exchange 256-byte encrypted messages (Pung's 32-byte labels account for the difference), while Dissent targets larger messages ($\geq$ 1 KB). The costs of PIR operations depend on two parameters: aggregation (α) and dimension (d) [10]. They control the number of ciphertexts that make up a PIR query and answer (higher α and d lead to smaller queries but larger answers). For each collection and tuple size, Pung dynamically chooses the parameters that minimize total network costs.

Figure 5 tabulates our results. We find that client-side operations incur little CPU costs aside from generating a PIR query. This operation is performed once by clients when retrieving a message, or several times (on smaller collections) when traversing a BST (§3.3). The network and CPU cost of generating and sending a PIR query depend on the number and the size of the ciphertexts that make up the query; for the PIR parameters that Pung uses (last two rows of Figure 5), these costs are sublinear in the size of the collection (i.e., $\sqrt{n}$). We discuss more about client-to-server network costs in Section 7.4.

[7] The anytrust model [137] states that out of a set of servers one is assumed to be correct; clients need not know which is the correct one.

	# tuples in Pung cluster (n)		
	2,048	8,192	32,768
client-side CPU costs			
Key derivation	6.05 μs	6.05 μs	6.05 μs
Label generation	1.60 μs	1.60 μs	1.60 μs
Message encryption	1.56 μs	1.56 μs	1.56 μs
Message decryption	1.37 μs	1.37 μs	1.37 μs
Bloom filter lookup	0.15 ms	0.47 ms	2.02 ms
PIR query (288 B tuples)	0.86 ms	1.91 ms	3.35 ms
PIR query (1 KB tuples)	1.68 ms	3.36 ms	5.02 ms
PIR decode (288 B tuples)	0.62 ms	0.69 ms	0.70 ms
PIR decode (1 KB tuples)	0.68 ms	0.69 ms	1.35 ms
server-side CPU costs			
PIR setup (288 B tuples)	4.52 ms	16.01 ms	68.73 ms
PIR setup (1 KB tuples)	15.64 ms	63.86 ms	255.38 ms
PIR answer (288 B tuples)	6.05 ms	14.91 ms	36.81 ms
PIR answer (1 KB tuples)	14.72 ms	37.87 ms	143.38 ms
network costs			
PIR query (288 B tuples)	256 KB	512 KB	1024 KB
PIR query (1 KB tuples)	512 KB	1,024 KB	1,536 KB
PIR answer (288 B tuples)	432 KB	464 KB	464 KB
PIR answer (1 KB tuples)	464 KB	464 KB	912 KB
PIR parameters (α, d) **[10]**			
288 B tuples	$(32, 2)$	$(32, 2)$	$(32, 2)$
1 KB tuples	$(8, 2)$	$(8, 2)$	$(16, 2)$

FIGURE 5—Microbenchmarks for Pung's operations under varying collection sizes (n), and tuple sizes (288 bytes and 1 KB).

Unlike clients' CPU costs, the server's costs are significant. One of the most expensive operation is the one-time setup of a PIR collection. In Pung, this procedure needs to be performed once at the beginning of every round following the send phase (§3.3). The other major source of overhead is answering PIR queries. In general, this cost scales linearly with n, though fixed costs make processing several small collections ($n < 8$K) relatively more expensive than processing a single large one. We return to this point in Section 7.3 when we discuss the theoretical versus actual benefits of our optimizations.

7.2 End-to-end performance of single retrievals

We focus on two end-to-end metrics: latency observed by a client and throughput achieved by Pung servers. Here we test the version of Pung that we describe in Section 3 without any of the multi-retrieval optimizations (§4).

Latency. To measure the end-to-end latency perceived by clients in Pung, we set up a single dataflow worker that is under-utilized and that can immediately handle a user's request. We then have a single client send its message and perform a retrieval. To experiment with large collection sizes we populate the server with up to 1 million 288-byte tuples. We experiment with three different methods that clients can use to retrieve their desired tuples from the server. The first has the client explicitly download all the label-to-index mappings prior to retrieval, look up the index of the corresponding label locally, and perform PIR with this index. The second downloads a Bloom filter that succinctly encodes the label-to-index mappings (§6), and performs the same steps as above. The last performs the BST retrieval procedure listed in Figure 4.

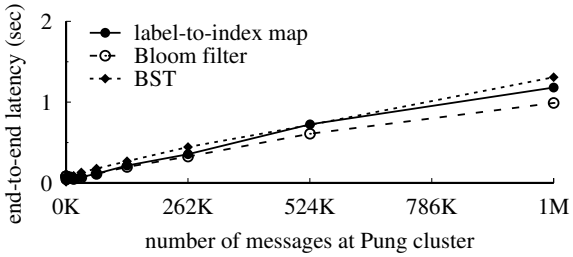

FIGURE 6—The end-to-end latency of sending and retrieving one message when the Pung cluster is under-utilized is up to 1.3 seconds (when the server stores 1 million tuples).

Figure 6 depicts the results. As we expect from our microbenchmarks, the client latency grows linearly with the number of messages at the server. Also, our low-latency network allows us to confirm that the server-side CPU costs associated with BST retrieval are negligibly higher than explicitly fetching the label-to-index mapping. However, in wide area networks we expect to see added latency due to $\log(n)$ round trips. The Bloom filter's checks (§6) also incur little CPU overhead, and its size is up to 10.4$\times$ smaller than the associated label-to-index mapping. Finally, note that our prototype performs request-level—rather than data-level—parallelism, so these latencies could be reduced further by having dataflow workers process fractions of a request. However, current latencies are already comparable to those achieved by Vuvuzela, where even a two-client scenario requires 20-second rounds due to the addition and serial processing of cover traffic.

Throughput. To measure Pung's peak throughput, we run experiments where clients send and retrieve a 256-byte message per round, for a total of 10 rounds. We then vary the number of clients (n) and measure the number of messages processed per minute. We distribute 64 timely dataflow workers across 4 VMs to run Pung's server-side computation. Since we cannot run tens of thousands of clients in our infrastructure, we employ a combination of real and simulated clients. We configure 512 real clients across 8 VMs (4 clients per core). We then have each client send a single message and instruct dataflow workers to make up the difference by injecting the remaining messages ($n-512$) at the end of the send phase, simulating additional clients. Finally, during the retrieve phase, each real client fetches a message from a random mailbox.

We also run both baselines in our cluster, with 256-byte messages. Since Dissent is a peer-to-peer system and does not use servers, we spread out its peers across our VMs. We run only its shuffle protocol as that is more efficient than full Dissent for small fixed-sized messages [42, §3].

For Vuvuzela, we set up a 3-server chain in addition to

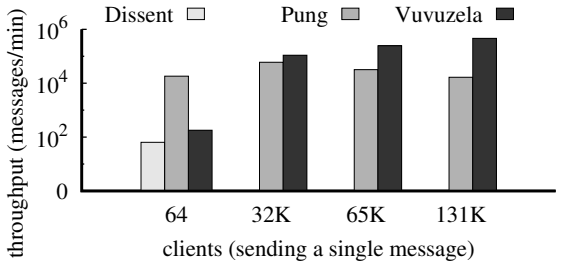

FIGURE 7—Pung can handle significantly more messages and clients than Dissent but its throughput at 131K clients is 27.8× lower than Vuvuzela's. We do not report Dissent's throughput past 64 users (see text for details).

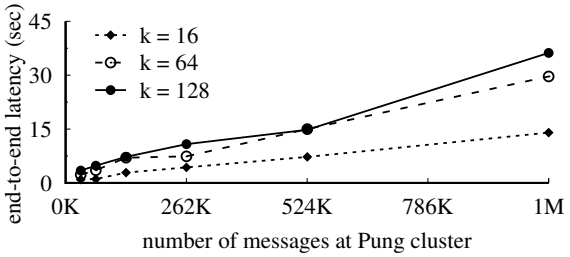

FIGURE 8—The end-to-end latency of sending one message and retrieving k using Pung's multi-retrieval. It takes 36.2 seconds with $k = 128$ and 1M tuples. This is 4.6× faster than retrieving 128 messages using Pung's single-retrieval (Fig. 6).

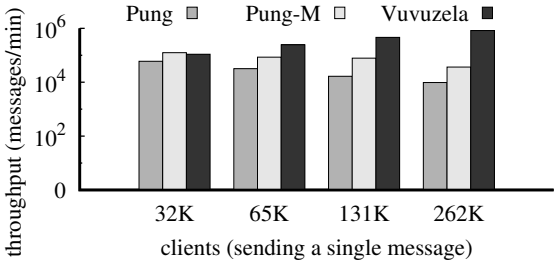

FIGURE 9—Pung's multi-retrieval optimizations increase its throughput by up to 5.2×. Pung-M represents a version of Pung where clients retrieve k=64 messages simultaneously using our hybrid scheme (§4.4). At 262K clients, Vuvuzela handles 84.9× and 22.6× more messages than Pung and Pung-M, respectively.

the entry server that proxies client requests, which mirrors the arrangement evaluated by its authors [130, §7]. A caveat is that our VMs have fewer CPU cores. We also use the same parameters that characterize the distribution from which Vuvuzela servers draw noise ($\mu = 300{,}000$ and $b = 13{,}800$). We run 512 Vuvuzela clients and modify the entry server [9] to make up for the remaining messages (similar to how Pung's dataflow workers inject messages).

Figure 7 depicts our results for 64, 32K, 65K, and 131K clients. We show Dissent's throughput only with 64 clients because at higher peer counts it is less than one message per minute with the prototype we use [6].

Pung and Vuvuzela achieve relatively low throughput—far below their capacity—at very low client counts. This is due to lack of work, since only 64 clients are sending and retrieving messages in a given round. As a result, Pung workers sit idle most of the time, while Vuvuzela servers continue to generate and process significant cover traffic, delaying the start of the next round. However, at higher (and more realistic) client counts, there is enough work to make long rounds a non-issue for Vuvuzela. Indeed, Vuvuzela's throughput is 27.8× higher than Pung at 131K clients, and this gap grows even larger with more clients.

7.3 What are the benefits of multi-retrieval?

We now discuss how our techniques (§4) impact the performance of Pung in terms of latency and throughput. In both cases, we run the same experiments described in Section 7.2, but configure clients to use the hybrid scheme (§4.4) to retrieve multiple messages at once.

Latency. As with the single retrieval case, client latency grows linearly with the number of messages at the server. This is depicted in Figure 8. However, with one million tuples, the multi-retrieval latency is 1.5×, 2.8×, and 4.6× lower than running the single retrieval protocol (§7.2) k times when retrieving $k = 16$, 64, and 128 messages respectively. Note that in this experiment we have a single dataflow worker respond to all of the client's queries (recall that there is a query for each subcollection). However, this is an embarrassingly parallel task since subcollections are independent; different workers could be assigned to each of them. Given enough workers, it is possible to drive down the end-to-end latency of processing all k requests to the level of processing a single request.

Throughput. We depict the throughput benefits of having clients retrieve a batch of $k = 64$ messages in Figure 9. We find that Pung's hybrid scheme offers a throughput boost of up to 5.2× over single retrieval. Based on our cost model (available in our extended report [12, Appendix B]), the maximum gain that we can expect from using our hybrid scheme over retrieving messages one by one is 14.2× for $k = 64$. This large disagreement (over 2×) with our experimental results comes from two main sources. First, our end-to-end throughput measures not only message retrieval but also Pung's send phase—including the expensive PIR setup step (§7.1) and the encoding of buckets using batch codes (§4.4)—which lowers our potential gains. Second, as we discuss in Section 7.1, smaller collections are disproportionately more expensive to serve than larger ones, owing to fixed costs.

Nevertheless, Pung's multi-retrieval throughput is high enough (5.9× lower than Vuvuzela's at 131K clients) that it can accommodate thousands of users and tens of thousands of messages with sub-minute latencies. This performance is sufficient to support many existing applications (§8). We also experiment with values of k ranging from 4 to 128, and find gains between 1.52×–11×.

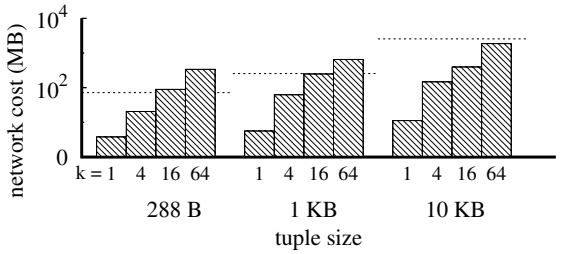

FIGURE 10—Pung's network costs (upload and download) for $n = 262$K with varying k and tuple sizes. The dashed line represents the cost of naively downloading the entire collection, which provides information-theoretic privacy. Pung's single retrieval is cheaper than naively downloading the entire collection. For $k>1$, Pung performs better than naive download only when messages are large, or when k is moderate (see text for details).

7.4 What costs does Pung impose on clients?

Pung's clients have to participate in every round to ensure unobservability (§3.1). Clients thus pay fixed CPU and network costs regardless of their actions. Our microbenchmarks (§7.1) show that many of these costs are small. Indeed, clients incur tens of milliseconds of CPU time per round for the experiments in Sections 7.2 and 7.3.

Network costs. To better understand the network costs incurred by clients, we run a set of experiments in which we vary the collection sizes (n), the number of messages retrieved by a client (k), and the size of tuples in the collection. Figure 10 summarizes the results for $n = 262$K tuples with varying k and the size of tuples.

We find that for single retrievals ($k = 1$), clients incur 3.8–11 MB of network costs for sending and receiving a message, depending on the tuple size. This cost is 3–4 orders of magnitude higher than retrieving the tuple from the server non-privately. However, compared to downloading the entire collection (which would also meet our privacy goals), it is 19× lower for 288 byte tuples, 45× lower for 1 KB tuples, and 230× lower for 10 KB tuples.

For $k>1$, we find that clients incur 4.5–36 MB per message depending on k and tuple size. Perhaps surprisingly, we find that under certain regimes (e.g., small tuple sizes, high k), it is beneficial for clients to simply download the entire collection instead of using Pung's multi-retrieval. The reason is that clients have to retrieve tuples from many subcollections—the number of which depends on k (§4.4)—by sending PIR queries and receiving PIR answers (several ciphertexts). With the PIR construction that we employ (i.e., XPIR [10]), ciphertexts are rather large (128 Kbits), so these overheads are more than the size of the collection for smaller tuple sizes and large k. While we can use a different cryptosystem with smaller ciphertexts (e.g., Paillier [101]) to reduce network costs by orders of magnitude, it incurs much higher server-side CPU costs [10]. We are investigating ways to resolve this conflict between network and CPU costs.

Admittedly, this is the primary limitation of Pung's current design. However, there are certain regimes in which Pung's multi-retrieval outperforms downloading the entire collection: larger messages (e.g., ≥1 KB), or medium k (e.g., ≤64). For example, with $k = 16$ and 10 KB messages, the total network cost is 7× lower than downloading the entire collection. Finally, while these costs may be considered modest for well-connected devices, they remain high for many settings (e.g., mobile devices).

8 Applicable scenarios

Section 7.3 demonstrates that Pung's optimizations can substantially increase its throughput, but they incur additional network resources and require clients to retrieve many messages at once. We now discuss applications that can benefit from Pung's privacy guarantees as well as its multi-retrieval—high network costs remain an issue.

First, participants in a dark pool (a private stock exchange) could hide their orders using Pung, preventing market speculation and predatory tactics by high-frequency traders [85, 103]. Second, email, group chats, and collaboration tools such as Slack [4] are all a natural fit for Pung: they use larger messages (>1 KB), and require (or benefit from) multi-retrieval.

Finally, several applications with many-to-one communication can use Pung. For instance, health/embedded devices can send diagnostic information to medical providers using Pung, preserving the privacy of the communication. Similarly, Pung enables private collection of data from sensors (e.g., Internet of things), or corporate software (e.g., bug reports). While these devices have limited resources (e.g., power, bandwidth) they can still use Pung, since they can choose (a priori) how often to participate (e.g., every 5 rounds). They can then leverage Pung's multi-retrieval to "catch up" by simultaneously retrieving all messages sent to them during the last 5 rounds. Of course, if a client rarely participates, its messages might be garbage collected before it can catch up (§6).

9 Related work

This section discusses related systems, and their comparison to Pung. (Danezis et al. [45] provide a more thorough discussion of many of these systems.)

Mix networks. The earliest private messaging systems employ *mix networks* [22, 23, 31, 32, 47, 65, 72, 81, 82]: they rely on a set of servers (called mixes) to shuffle messages before delivering them to recipients. This shuffling is often accompanied by encryption, batching, and chaffing (the addition of dummy traffic) to prevent traffic analysis. Since all operations are relatively lightweight, these systems enjoy lower latency and higher throughput than many other works in the literature—including Pung. However, malicious mixes can replay, duplicate, or drop

messages, violating these systems' guarantees via known attacks [84, 87, 100, 106, 107, 112, 122, 134]. Indeed, Kesdogan et al. [71] show that many of these attacks are fundamental. Consequently, systems like Aqua [82] and Herd [81] sidestep these attacks by targeting scenarios where particular mixes with critical roles are trusted. The use of such trusted mixes contradict our goals (§2).

There are works with a decentralized architecture: peer-to-peer mix networks [114, 138] and peer-to-peer routing [17, 37, 46, 55, 56, 98, 113, 121]. These systems have high network costs, and rely on a threshold of peers being correct. Furthermore, they are susceptible to strong adversaries [54, 91, 124] and Sybil attacks [52]. Salsa [98] combats these issues by making an additional assumption: fewer than 20% of all nodes are malicious. Blindspot [56] and `Drac` [46] suggest peering only with contacts from existing social networks, but this leaks information about users' relationships and results in small anonymity sets.

Onion routing. Works based on onion routing [51, 92, 93, 127], especially Tor [51], are widely adopted due to their relative low latency and ability to support millions of users. However, these systems are unable to resist traffic analysis attacks [68, 96, 112], even those performed by local adversaries [29, 78, 102, 132]. While future Internet architectures may address many of these shortcomings [34], we target a system that is deployable today.

DC networks. Another line of work is based on Dining Cryptographers (DC) networks [33, 42, 66]. They provide stronger guarantees than Pung under the same threat model, but they are peer-to-peer (requiring all users to know each other) and are based on all-to-all broadcast of messages. This results in high costs. Consequently, these systems typically accommodate only dozens of users. Verdict [43] and Dissent's successor [136] make great strides to reduce these costs and support thousands of users, but in the process introduce trusted infrastructure (under the anytrust model) which differs from our goals (§2).

Mailbox systems. Finally, there are a number of systems [25, 40, 41, 75, 79, 119, 130] that employ an architecture and techniques similar to Pung's (clients retrieve messages from per-round mailboxes kept at third-party servers). The key differences between these works and Pung is their reliance on at least one correct server, and the mechanisms that follow from that assumption. We elaborate on the most related ones below.

P^3 [75], like Pung, employs a key-value store from which users can privately pull messages. While P^3's focus is a retrieval mechanism that supports general queries when fetching a message (e.g., prefix search), Pung's primary goal is to drive down the cost of retrieval by introducing several batching optimizations (§4).

Riposte [41] targets a setting more fitting for whistleblowers and informants where the sender wishes to remain anonymous from everyone (including all recipients). In contrast, Pung's goal is hiding the communication pattern between users who already know each other's identities. The Pynchon Gate [119] provides anonymity by composing a mix network with an IT-PIR scheme (§3.3). However, these guarantees hold only for passive adversaries who do not compromise mixes; under our threat model several attacks exist [100, 106, 107, 134]. Riffle [79] addresses this limitation by enhancing mixes with a verifiable shuffle, but retains the IT-PIR substrate and the anytrust model, which requires at least one correct server.

Vuvuzela [130] provides privacy through request shuffling and the careful addition of cover traffic rather than through PIR. Vuvuzela achieves significantly better performance than Pung (§7.2, §7.3), and it proposes an efficient dialing protocol, which Alpenhorn [80] enhances further. In contrast, Pung is not compatible with either dialing scheme, and we have not yet identified a suitable substitute (§5). However, Pung does introduce some benefits. In Vuvuzela, messages are ephemeral and can only be accessed during a single round; Pung supports long-lived messages that can be retrieved anytime prior to garbage collection (§6). Vuvuzela does not support group communications since it is based on point-to-point exchanges. Finally, the guarantees of a Vuvuzela deployment are based on differential privacy and are valid only for a certain number of rounds (based on a privacy budget). Pung's guarantees hold for any number of rounds.

10 Summary and conclusion

Our goal was to eliminate trust assumptions in private communication. To accomplish this goal, we leverage powerful cryptography and build Pung. Pung supports $10^3\times$ more users than prior systems in a similar threat model but falls short of systems that make trust assumptions. To improve performance, Pung targets a setting where clients retrieve multiple messages at once (§8). In this regime, Pung introduces new techniques that heavily amortize the costs of its cryptographic machinery. Our evaluation confirms that Pung reduces computational costs by up to $11\times$, at the expense of higher network costs. With these improvements, Pung presents an attractive design point for private communication systems.

Acknowledgments

Careful comments from Trinabh Gupta, Michael Lee, Josh Leners, Jay Lorch, Manos Kapritsos, Bryan Parno, Riad Wahby, the anonymous reviewers, and our shepherd Gernot Heiser made this paper better. We thank Michael Walfish for his thorough comments, which greatly improved this work. We also thank Carlos Aguilar Melchor for help with XPIR, and Frank McSherry for help with dataflow operators that Pung uses. Sebastian Angel was supported by NSF grants CNS-1055057 and CNS-1514422.

References

[1] Bleep. http://www.bleep.pm.

[2] ChatSecure. https://chatsecure.org.

[3] Open Whisper Systems. https://whispersystems.org.

[4] Slack: Be less busy. https://slack.com/.

[5] Telegram. https://telegram.org.

[6] Dissent: Provably anonymous overlay. https://github.com/dedis/Dissent/tree/95f73, Apr. 2010.

[7] Google says anything flowing across open WiFi is fair game. https://goo.gl/fjOW2A, Jan. 2014. Privacy SOS.

[8] Rust-crypto. https://github.com/dagenix/rust-crypto/, 2016.

[9] Vuvuzela: Private messaging system that hides metadata. https://github.com/davidlazar/vuvuzela, Sept. 2016.

[10] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[11] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. https://github.com/xpir-team/xpir/, 2016.

[12] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure (extended version). Technical Report TR-16-16, The University of Texas at Austin, Oct. 2016.

[13] J. Angwin, C. Savage, J. Larson, H. Moltke, L. Poitras, and J. Risen. AT&T helped U.S. spy on Internet on a vast scale. http://goo.gl/Jfsm18, Aug. 2015. The New York Times.

[14] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1994.

[15] J. Ball. GCHQ captured emails of journalists from top international media. http://goo.gl/YzXnYK, Jan. 2015. The Guardian.

[16] J. Bamford. Shady companies with ties to Israel wiretap the U.S. for the NSA. http://goo.gl/bdi7w4, Apr. 2012. Wired.

[17] A. Beimel and S. Dolev. Buses for anonymous message delivery. *Journal of Cryptology*, 16(1), Jan. 2003.

[18] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2000.

[19] M. Bellare, R. Canetti, and H. Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1996.

[20] M. Bellare and A. Lysyanskaya. Symmetric and dual PRFs from standard assumptions: A generic validation of an HMAC assumption. Cryptology ePrint Archive, Report 2015/1198, Dec. 2015. http://eprint.iacr.org/2015/1198.pdf.

[21] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Dec. 2000.

[22] O. Berthold, H. Federrath, and S. Köpsell. Web MIXes: A system for anonymous and unobservable Internet access. In *Proceedings of the International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobserbability*, July 2000.

[23] O. Berthold and H. Langos. Dummy traffic against long term intersection attacks. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, Mar. 2002.

[24] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), July 1970.

[25] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, June 2015.

[26] J. Brickell and V. Shmatikov. Efficient anonymity-preserving data collection. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, Aug. 2006.

[27] S. Buttar. Dragnet NSA spying survives: 2015 in review. https://goo.gl/JsNgS7, Dec. 2015. Electronic Frontier Foundantion.

[28] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1999.

[29] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008.

[30] Y.-C. Chang. Single database private information retrieval with logarithmic communication. In *Proceedings of the Australasian Conference on Information Security and Privacy*, July 2004.

[31] D. Chaum, F. Javani, A. Kate, A. Krasnova, J. de Ruiter, and A. T. Sherman. cMix: Anonymization by high-performance scalable mixing. Cryptology ePrint Archive, Report 2016/008, Jan. 2016. http://eprint.iacr.org/2016/008.pdf.

[32] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), Feb. 1981.

[33] D. L. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1), 1988.

[34] C. Chen, D. E. Asoni, D. Barrera, G. Danezis, and A. Perrig. HORNET: High-speed onion routing at the network layer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2015.

[35] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, Feb. 1998. http://eprint.iacr.org/1998/003.

[36] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1995.

[37] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobserbability*, July 2000.

[38] D. Cole. We kill people based on metadata. http://goo.gl/LWKQLx, May 2014. The New York Review of Books.

[39] T. Cook. A message to our customers. http://www.apple.com/customer-letter/, Feb. 2016.

[40] D. A. Cooper and K. P. Birman. Preserving privacy in a network of mobile computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1995.

[41] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[42] H. Corrigan-Gibbs and B. Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2010.

[43] H. Corrigan-Gibbs, D. I. Wolinsky, and B. Ford. Proactively accountable anonymous messaging in Verdict. In *Proceedings of the USENIX Security Symposium*, Aug. 2013.

[44] Council of Europe. European Convention on Human Rights: Article 8. http://www.echr.coe.int/Documents/Convention_ENG.pdf, Nov. 1950.

[45] G. Danezis, C. Diaz, and P. Syverson. Systems for anonymous communication. https://securewww.esat.kuleuven.be/cosic/publications/article-1335.pdf, Aug. 2009.

[46] G. Danezis, C. Diaz, C. Troncoso, and B. Laurie. Drac: An architecture for anonymous low-volume communications. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2010.

[47] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2003.

[48] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[49] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), Nov. 1976.

[50] R. Dingledine. Did the FBI pay a university to attack Tor users? https://goo.gl/NB3hSR, Nov. 2015. Tor Project.

[51] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium*, Aug. 2004.

[52] J. R. Douceur. The sybil attack. In *Proceedings of the International Workshop on Peer-to-Peer Systems*, Mar. 2002.

[53] M. Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. Technical Report SP 800-38D, National Institute of Standards and Technology, Nov. 2007.

[54] C. Egger, J. Schlumberger, C. Kruegel, and G. Vigna. Practical attacks against the I2P network. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Nov. 2013.

[55] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2002.

[56] J. Gardiner and S. Nagaraja. Blindspot: Indistinguishable anonymous communications. arXiv:1408/0784v2, Aug. 2014. http://arxiv.org/abs/1408.0784.

[57] W. Gasarch and A. Yerukhimovich. Computationally inexpensive cPIR. https://www.cs.umd.edu/~arkady/papers/pirlattice.pdf, 2006.

[58] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, July 2005.

[59] I. Goldberg. Improving the robustness of private information retrieval. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[60] P. Golle and A. Juels. Dining cryptographers revisited. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2004.

[61] A. Greenberg. Whatsapp just switched on end-to-end encryption for hundreds of millions of users. http://www.wired.com/2014/11/whatsapp-encrypted-messaging/, Nov. 2014.

[62] G. Greenwald and R. Gallagher. New Zealand launched mass surveillance project while publicly denying it. https://goo.gl/UwNpwV, Sept. 2014. The Intercept.

[63] G. Greenwald and E. MacAskill. NSA Prism program taps in to user data of Apple, Google and others. http://goo.gl/qETWUq, June 2013. The Guardian.

[64] J. Groth, A. Kiayias, and H. Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, May 2010.

[65] C. Gülcü and G. Tsudik. Mixing E-mail with Babel. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 1996.

[66] E. Gün Sirer, S. Goel, M. Robson, and D. Engin. Eluding carnivores: File sharing with strong anonymity. In *Proceedings of the ACM SIGOPS European Workshop*, Sept. 2004.

[67] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2016.

[68] N. Hopper, E. Y. Vasserman, and E. Chan-Tin. How much anonymity does network latency leak? In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007.

[69] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Batch codes and their applications. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, June 2004.

[70] J. Katz and M. Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In *Proceedings of the Fast Software Encryption Workshop (FSE)*, Apr. 2000.

[71] D. Kesdogan, D. Agrawal, V. Pham, and D. Rautenbach. Fundamental limits on the anonymity provided by the MIX technique. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[72] D. Kesdogan, J. Egner, and R. Büschkes. Stop-And-Go-MIXes providing probabilistic anonymity in an open system. In *Proceedings of the International Workshop on Information Hiding*, Apr. 1998.

[73] A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, and Q. Tang. Optimal rate private information retrieval from homomorphic encryption. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2015.

[74] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. *Journal of Random Structures and Algorithms*, 33(2), Sept. 2008.

[75] L. Kissner, A. Oprea, M. K. Reiter, D. Song, and K. Yang. Private keyword-based push and pull with applications to anonymous communication. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, June 2004.

[76] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2010.

[77] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1997.

[78] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas. Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[79] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[80] D. Lazar and N. Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[81] S. Le Blond, D. Choffnes, W. Caldwell, P. Druschel, and N. Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2015.

[82] S. Le Blond, D. Choffnes, W. Zhou, P. Druschel, H. Ballani, and P. Francis. Towards efficient traffic-analysis resistant anonymity networks. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2013.

[83] R. Lenzner. ATT, Verizon, Sprint are paid cash by NSA for your private communications. `http://goo.gl/x7Cz1m`, Sept. 2013. Forbes.

[84] B. N. Levine, M. K. Reiter, C. Wang, and M. Wright. Timing attacks in low-latency mix systems. In *Proceedings of the International Financial Cryptography Conference*, Feb. 2004.

[85] M. Lewis. *Flash Boys: A Wall Street Revolt*. W.W. Norton & Company, Mar. 2014.

[86] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *Proceedings of the International Financial Cryptography and Data Security Conference*, Jan. 2015.

[87] N. Mathewson and R. Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, May 2004.

[88] J. Mayer, P. Mutchler, and J. C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 113(20), May 2016.

[89] F. McSherry. Timely dataflow. `https://github.com/frankmcsherry/timely-dataflow/`, 2016.

[90] J. Menn. Yahoo secretly scanned customer emails for U.S. intelligence. `https://goo.gl/KZuUYo`, Oct. 2016. Reuters.

[91] P. Mittal and N. Borisov. Information leaks in structured peer-to-peer anonymous communication systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2008.

[92] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable anonymous communication using private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2011.

[93] P. Mittal, M. Wright, and N. Borisov. Pisces: Anonymous communication using social networks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2013.

[94] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), Oct. 2001.

[95] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, Jan. 2005.

[96] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[97] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow

system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[98] A. Nambiar and M. Wright. Salsa: A structured approach to large-scale anonymity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2006.

[99] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1990.

[100] L. Nguyen and R. Safavi-Naini. Breaking and mending resilient mix-nets. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, Mar. 2003.

[101] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1999.

[102] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Oct. 2011.

[103] D. C. Parkes, C. Thorpe, and W. Li. Achieving trust without disclosure: Dark pools and a role for secrecy-preserving verification. In *Proceedings of the Conference on Auctions, Market Mechanisms and Their Applications (AMMA)*, Aug. 2015.

[104] M. B. Paterson, D. R. Stinson, and R. Wei. Combinatorial batch codes. *Advances in Mathematics of Communications (AMC)*, 3(1), Feb. 2009.

[105] A. Pfitzmann and M. Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. `http://dud.inf.tu-dresden.de/literatur/Anon_Terminology_v0.34.pdf`, Aug. 2010.

[106] B. Pfitzmann. Breaking an efficient anonymous channel. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1995.

[107] B. Pfitzmann and A. Pfitzmann. How to break the direct RSA-implementation of mixes. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Apr. 1989.

[108] A. Piotrowska, J. Hayes, N. Gelernter, G. Danezis, and A. Herzberg. AnoNotify: A private notification service. Cryptology ePrint Archive, Report 2016/466, May 2016. `http://eprint.iacr.org/2016/466.pdf`.

[109] E. Protalinski. Facebook scans chats and posts for criminal activity. `http://goo.gl/pfV9XE`, July 2012. CNET.

[110] C. Rackoff and D. R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 1991.

[111] A. S. Rawat, Z. Song, A. G. Dimakis, and A. Gál. Batch codes through dense graphs without short cycles. *IEEE Transactions on Information Theory*, 62(4), Apr. 2016.

[112] J.-F. Raymond. Traffic analysis: Protocols, attacks, design issues and open problems. In *Proceedings of the Workshop on Privacy Enhancing Technologies (PET)*, May 2001.

[113] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1), Nov. 1998.

[114] M. Rennhard and B. Plattner. Introducing MorphMix: Peer-to-peer based anonymous Internet usage with collusion detection. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Nov. 2002.

[115] R. L. Rivest. Chaffing and winnowing: Confidentiality without encryption. *CryptoBytes Technical Newsletter (RSA Laboratories)*, 4(1), July 1998.

[116] P. Rogaway. The moral character of cryptographic work. Cryptology ePrint Archive, Report 2015/1162, Dec. 2015. `http://eprint.iacr.org/2015/1162.pdf`.

[117] A. Rusbridger. The Snowden leaks and the public. `http://goo.gl/VOQL86`, Nov. 2013. The New York Review of Books.

[118] D. Rushe. Yahoo $250,000 daily fine over NSA data refusal was set to double 'every week'. `http://goo.gl/FZGfTT`, Sept. 2014. The Guardian.

[119] L. Sassaman, B. Cohen, and N. Mathewson. The Pynchon Gate: A secure method of pseudonymous mail retrieval. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Nov. 2005.

[120] B. Schneier. *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World*. W.W. Norton & Company, Mar. 2015.

[121] R. Sherwood, B. Bhattacharjee, and A. Srinivasan. P5: A protocol for scalable anonymous communication. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002.

[122] V. Shmatikov and M.-H. Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Sept. 2006.

[123] N. Silberstein and A. Gál. Optimal combinatorial batch codes based on block designs. *Designs, Codes and Cryptography*, 78(2), Feb. 2016.

[124] A. Singh, T.-W. Ngan, P. Druschel, and D. S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Apr. 2006.

[125] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2007.

[126] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4), Aug. 1969.

[127] P. F. Syverson, D. M. Goldschlag, and M. G. Reed. Anonymous connections and onion routing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1997.

[128] United Nations General Assembly. The Universal

Declaration of Human Rights: Article 12. http://www.un.org/en/universal-declaration-human-rights/, Dec. 1948.

[129] United States Congress. Electronic Communications Privacy Act of 1986 (ECPA). https://it.ojp.gov/privacyliberty/authorities/statutes/1285, Oct. 1986.

[130] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[131] M. Waidner and B. Pfitzmann. The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Apr. 1989.

[132] T. Wang and I. Goldberg. Improved website fingerprinting on Tor. In *Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES)*, Nov. 2013.

[133] S. Warren and L. Brandeis. The right to privacy. *Harvard Law Review*, 4(5), Dec. 1890.

[134] D. Wikström. Five practical attacks for "optimistic mixing for exit-polls". In *Proceedings of the Conference on Selected Areas in Cryptography (SAC)*, Aug. 2003.

[135] P. Williams and R. Sion. Usable PIR. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2008.

[136] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[137] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Scalable anonymous group communication in the anytrust model. In *Proceedings of the European Workshop on System Security (EUROSEC)*, Apr. 2012.

[138] B. Zantout and R. A. Haraty. I2P data communication system. In *Proceedings of the International Conference on Networks*, Jan. 2011.